Possibilities for Active Messaging in PVM

Philip J. Mucci
mucci@cs.utk.edu
and
Jack Dongarra
dongarra@cs.utk.edu

December 1994

Abstract

Active messaging is a communications model designed around the interaction of a network interface and its driving software in an operating system. By utilizing this model, the user can design applications that make better use of the available computing and communication resources. Currently, successful implementations exist only for a certain subset of workstations and network adapters. This paper is an exploration into a portable implementation of active messaging for possible inclusion to the PVM suite, a generalized framework for distributed computing.

1 Introduction

Active messaging was first introduced at the University of California at Berkeley as a means to reduce the communication latency inherent in distributed computing. The idea was driven by the discrepancy in performance between traditional multiphase communication protocols and the actual access time of the network device. The concept of AM is as elegant as it is simple. When an AM arrives, the address of a user-defined handler is read out of the message's header. This handler is then invoked with a the pointer to the message as its argument. This handler is small and compact. It performs some finite amount of work, which may include sending a reply AM, and then returns. The user's program can then continue. (This process is similar to that of an interrupt-driven device driver, the critical difference being that the *user's* code is executed upon reception of an interrupt from the device.) The role of the user-defined handler is merely to fill or empty the data from the network's buffers. Usually this consists of a simple `read()` or `memcpy()` and the setting of a few flags that are polled during computation. Active messages are intended to be the primitives upon which all other communication operations are based. It has been shown that both the two phase protocol used in most networking stacks and the less common three phase deadlock-free protocol map quite naturally to the AM paradigm[17]. This mapping implies that AM much more closely resembles the underlying communication model than do traditional communication protocols. Distributed applications can thus be redesigned with AM and can realize significant improvement in effective bandwidth. Active messages are not the same as remote procedure calls (RPCs). RPCs are heavyweight procedures usually consisting of many system calls. RPCs are often executed in a different process from the one that services the network. Moreover, RPCs are designed to perform a significant number of computations in the user's program only upon reception of a message. A program using active messaging, on the other hand, is working all the time and is interrupted only when there is data to be exchanged. The job of the handler specified in the message is simply to extract the data from the network and integrate it into the compute loop. In this way, computation and communication can be overlapped—hiding much of the latency encountered when using today's networks and their associated access methods.

In this paper we explore the feasibility of developing a PVM-AM layer. Any addition to the PVM suite must be portable to a wide range of platforms. Unfortunately, the portability of a piece of software is often inversely proportional to its performance in driving the hardware efficiently. Here, we equate good performance with achieving a high percentage of the hardware's theoretical bandwidth. For an effective implementation of active messaging in PVM, then, a tradeoff must be made: minimizing network access latency while maintaining a significant level of portability.

2 The Network Interface

Active messages were designed to be as close to the hardware model as possible. In this way, protocol processing is minimized, even when the network is somewhat unreliable. All implementations assume that a very small percentage of packets will be dropped (as a result of collisions, propagation delay, media quality, etc…). The objective is to pay a performance penalty only in the infrequent event that a packet is lost. Hence, a very simple request-and-reply protocol will suffice to get the data through reliably. This protocol will be covered later in the paper.

In theory, the AM layer has direct access to the network device's FIFOs and receives interrupts upon packet reception. On the CM5 and similar architectures where network interrupts are not available, the AM layer polls the interface for available data. Unfortunately, direct access to the network is often very difficult to achieve. In fact, many AM implementations consist of large kernel patches to make such access easier. When using an unreliable transport, additional problems must be solved. These include flow control, error detection, packet timeouts, and fragmentation. All of these problems, as well as that of portability, can be solved by using the operating system's networking protocols. These, however, incur significant overhead and thus limit the potential bandwidth. Higher performance interfaces are available. Unfortunately one must have root privileges to access them in all commercial Unix implementations.

In this paper we explore the various alternatives available in designing a portable AM layer to be incorporated into the PVM suite. We conclude with specific suggestions to assist the developer in achieving a higher network throughput.

2.1 Unrestricted Access

The privileged user has basically five options to consider in terms of the underlying communication mechanism used for active message transmission. Three of the five communication options are related in that they directly control the network device while the other two are abstracted from the hardware.

2.1.1 Direct Control of the Network

In the first option, the programmer may work directly with the network interface's device driver. The small overhead required to send and receive a message is of great benefit. Usually this consists of a system call and the time to copy the data from user space into kernel mbufs. However, a device driver's `ioctl()s` can present quite a challenge to the developer. These calls are often not portable and poorly documented. The network interface would be have to be polled for data because most operating systems will not dispatch hardware interrupts to a user process. The network would have to be dealt with on a packet-by-packet basis, with fragmentation being done by the user of the AM library. Most important, code would have to be written and rewritten for each operating system, networking board, and packet format. Given the plethora of connectivity options available today, an implementation of this type is just not practical.

Another option is to add a new device driver into the kernel. This module could consist of a vector of stub functions referencing a real network device's vector of file operations. A distributed application could then be linked with an AM library that consisted of the AM formatting routines and a few customized kernel traps. Once such a driver was installed, the user would *not* need to have root privileges to use the network device. Obviously, this situation presents a significant security risk. To overcome such a risk, the AM library and the new driver would have to incorporate some sort of authentication and protection mechanism. All of the problems of the previous approach apply here as well. The difficulty of kernel hacking and widely varying formats for loadable device drivers would further complicate the developer's job. However, it has been shown that this approach can be very successful if the target machines are limited to the same architecture, operating system, and networking interface[15].

The third option in this category of direct device control is to `mmap()` the network interface into the user's address space. Dealing with the network would require detailed hardware documentation and complete network specifications. Control would be accomplished through register-based operations and could possibly DMA transfers. In addition to having to deal with all the problems of the previous two approaches, the interface would have to be timeshared among multiple processes performing network I/O. Nevertheless, a successful implementation of this approach has been achieved in completely homogeneous environment[9].

2.1.2 An ATM Implementation

Asynchronous Transfer Mode (ATM) consists of a specification for packet format and network switching. Each ATM packet, or cell, is fifty-six bytes, forty of which are available for data. ATM networks have been praised for their promise to deliver high network bandwidth upon demand. It should be noted that these networks do not provide any form of reliability or flow control.

Active messaging has been implemented on an ATM network by using the Fore Systems SPARCstation interface and 140 Mb/sec TAXI fiber. The work was done at Cornell University primarily by Thorsten von Eicken, one of the pioneers of active messaging. Von Eicken and his colleagues achieved a maximum bandwidth of 5.5 MB/sec using the remote memory access primitives outlined in Figure 1.2. It should be noted, however, that this implementation was highly optimized. Through cooperation with Sun and Fore, the Cornell group patched SunOS and installed a device driver that allowed direct access to the network device's FIFOs. Transmitting a message consisted of formatting a cell, writing it to the card's FIFO, and trapping to the kernel. For comparison with less optimized strategies, the group wrote a traditional device driver and achieved 4.5 MB/sec. This result agreed quite well with Fore Systems' own API, which averaged around 4 MB/sec. It was found through exhaustive profiling that throughput was limited almost entirely by the trap to the kernel. This result is not surprising, given the small payload of ATM packets. To achieve anything close to maximum throughput would require a DMA card with an onboard processor that automated cell formatting and transmission. Also, software would be required that would allow the user to initiate these events without suffering the overhead of a trap to the kernel. Despite these difficulties, the performance achieved by the Cornell group is quite impressive, especially when compared with traditional twisted pair networks[15].

2.1.3 A FDDI Implementation

Another protocol gaining in popularity is the fiber optic distributed data interface (FDDI). The current specification dictates a maximum throughput of 100 Mb/sec, although progress is being made toward a 200 Mb/sec synchronous standard. Each FDDI packet can be up to 4,352 bytes long, approximately 20 of which are for control. As with ATM and Ethernet, FDDI makes no provisions for guaranteed delivery or data integrity. Unlike Ethernet and ATM, however, FDDI is token based. Any machine wishing to transmit a message must first acquire the token through a broadcast operation. Also, messages are explicitly forwarded from node to node, instead of being allowed to propagate along the entire medium. This strategy has the positive effect of eliminating collisions, but it can cause higher than normal latencies for networks with many nodes.

Richard Martin of the University of California at Berkeley implemented active messaging on an FDDI network of four HP PA-RISC workstations. The FDDI card in each workstation was memory mapped into each communicating process and a special process called the scheduler. Through mutual exclusion constructs, the scheduler allowed only one user process to access the board at any given time. The "active" user process forwarded any packets not destined for it to the scheduler, which distributed them accordingly. Outgoing packets from "excluded" processes were handled in a similar manner. Mr. Martin achieved a sustained transfer rate of 12 MB/sec with large packets and measured round-trip latencies as low as 29 μsec for packets with no payload. This is certainly quite an accomplishment, especially when compared with the performance of interconnection networks on today's massively parallel processing systems. Even though the code is virtually unportable, the algorithms and protocols are universal in their applicability. These will be outlined later in the paper.

2.1.4 Indirect Control of the Network

Recently, OS engineers have started to address the problem of writing portable code that deals directly with the network interface. To this end, the Data Link Provider Interface (DLPI) was introduced in the System V release of Unix. Using DLPI consists of specifying an option either to the `socket()` call or through an API of its own. DLPI does not require that the operating system provide for asynchronous or signal-driven operation; however, nonblocking routines are part of its specification. Unfortunately, DLPI was not adopted by those companies marketing a OS that was not a System V derivative. Instead, these companies chose to develop their own interface. Nevertheless, the ideas of offering the user simple and complete access to the media as well as ability to easily switch from one packet format to another seem to have survived. If PVM does adopt a system daemon, DLPI and its non-System V analogues may be the best solution.

An even further abstraction from the hardware is the IP layer. IP offers a number of advantages. For instance, it performs a header checksum that is essentially free. Its sockets can be set up to provide both asynchronous and nonblocking operation. And most important, any layer written using IP would very portable. On the other hand, many of IP's extra features come with a hefty price tag. For example, fragmentation and reassembly mean that buffering is performed in the kernel. This in turn means that we must incur the cost of allocating and copying blocks of memory during each operation. Another feature, flow control, can affect performance by having *too* sophisticated an algorithm, as well as by flooding the network with explicit control messages. Furthermore, these two features can cause a process to block, thus further affecting performance through unnecessary context switching and kernel overhead.

2.2 Restricted Network Access

Traditionally, any user-level application that needed network access had two choices: either the TCP or the UDP protocol stacks. Both are a standard part of all Unix networking packages, and their interfaces follow strict guidelines. TCP and UDP are both implemented on top of the IP layer. This means that once a packet is encapsulated (and possibly fragmented), it is passed to the IP stack for further processing. Both TCP and UDP support nonblocking operation and asynchronous notification upon delivery. These characteristics are perhaps their only similarities, for these two protocols are targeted at different audiences.

The TCP stack presents a reliable, connection-oriented data stream to the application. It accomplishes complete reliability through various techniques, most of which are beyond the scope of this paper. Of particular interest to us here is the amount of overhead caused by each. Internal buffering, dynamic memory allocation, timeouts for both packets and buffers, state tables, and explicit control messages all require extra cycles to manage and are responsible for a significant amount of delay. Each send performed using a deadlock-free protocol requires three trips over the network before the reader can continue (¡- request to send, -¿ ready-to-receive, ¡- data and -¿ acknowledge). Also, each trip requires a message of different composition. Connections to each host must established in advance and subsequently kept active, placing further demand on the kernel and its resources. TCP was designed for communicating data streams and does not preserve message boundaries, whereas distributed computing often involves infrequent and irregular communication of discrete messages. Explicit state of the interface would have to be maintained to prevent the user from receiving control messages. Otherwise, incoming messages would have to be scanned for headers and processed piecemeal. On many PCs and workstations, using TCP can severely limit an application's network throughput. The performance one achieves is frequently limited by the power of the CPU and not by the networking device or the speed of the medium. For our purposes, its level of functionality is largely unnecessary, especially considering our assumption of a low rate of packet loss.

The UDP stack is a connectionless, unreliable, datagram-oriented service. In fact, UDP adds very little to IP's functionality. Its only additions are an optional checksum of the data, a larger packet size (often 4k), and the ability to be used by non-root users. UDP's overhead is higher than IP's, primarily due to the computation of the checksum and the subsequent formation of the header. For the situation where portability is of the utmost importance and root access is not practical, an AM implementation using a UDP-like protocol offers the best chance of success.

BSD Unix was the first to make an addition to these two protocols. Called the Transport Layer Interface (TLI), it is now a part of most commercial operating systems. The TLI is another level of abstraction from the Internet protocols and their addressing formats. It provides the ability to implement another protocol on top of any of the existing stacks. For example, a new protocol could be defined that could function on any network where hosts were speaking either TCP or IPX. Unfortunately, TLI programming involves a lot of code—particularly in the area of state control and maintenance. The user must formulate a state machine capable of handling any condition raised from the lower layers. In truth, the two layers available to the nonroot user already handle this, and much work must be duplicated. Given its coding difficulty and the limited number of protocols with which to work, TLI does not appear to be a viable option.

2.3 Protocol Performance

As processor speeds have increased, the time spent processing communication protocols has become less significant in terms of affecting network throughput. It has been shown that for higher speed networks like FDDI, most of the time is spent in the the kernel performing mbuf management, data movement and checksum computation. Kay and Pasquale demonstrated that UDP performs only ten to fifteen percent better than TCP on a DecStation 5000[7]. The authors own experiments using `netperf` confirmed this fact. This seems to indicate a fundamental problem with the way data is exchanged between the user, the kernel and the communication device. Most operating systems' networking codes utilize the 4.3 BSD release of Unix as a reference point, and thus this is not suprising.

As communication bandwidth increases, the industry is slowly starting to realize this shortcoming and is rectifying it through various optimizations.[1] Until these changes are made public and applied across a wide variety of machines, the user has but one choice to guarantee the efficient use of the machines resources. That is, to use specialized APIs that allow the non-root user to eliminate much of the kernel from the processing loop by controlling the network device directly.

3 An Active Messaging Protocol

For our AM implementation, an N-way request-and-reply protocol will be used as outlined in Richard Martin's paper. He has shown that this model is capable of achieving full network bandwidth when using efficient network access methods [9]. Initially, a one-way model will be presented for clarity. This model will subsequently be extended to N ways to realize higher performance.

The request-and-reply protocol dictates that two messages be exchanged for every transaction: a request and a reply. All requests *must* send a reply, but no action is required of the user: the AM library automatically sends an empty acknowledgment if none is specified. The protocol also prevents any reply handler from accessing the network, effectively eliminating race conditions.

A traditional one-way request-and-reply protocol works as follows. Given V nodes, each node allocates two tables of length V for outgoing messages. The first table stores request AMs, and the second stores replies to requests made from the other nodes. Each position in each table corresponds to a fixed, unique destination address. When any message is to be sent, the buffer that corresponds to its type (a request or reply) and its destination address is examined. If that buffer is free, the message is stamped with an instance number (either 0 or 1), copied into the buffer, and injected into the network. Otherwise, the message will stall and wait for the buffer to become available. The request buffer is freed when a reply with a matching instance number is received. The reply buffer is freed when a request with a new instance number is received from the associated node. This new request means that the previous reply was received and processed by the sender; hence it is no longer of any use. Note that this protocol allows only one outstanding request and reply per node in the table. Thus, the instance number need be only a single bit.

One advantage to using a split-phase protocol is that it completely eliminates livelock and deadlock among communicating processes. Every request is automatically paired with a reply. Doing so avoids the overhead caused by code associated with the detection of such conditions.

Another advantage is that this model provides a simple and efficient form of flow control. An occupied request buffer means that the receiving node has not yet processed the request and sent the reply. Any attempt to send a request to this node will stall until a reply is received. Sophisticated buffer management and the explicit exchange of control messages are unnecessary. The benefits of this approach will become apparent upon the extension of the model.

Reliability is a problem only when communicating over local area networks (LANs), since all MPPs guarantee packet delivery and data integrity. A key item to remember is that very low rates of packet loss and corruption are being assumed. Reliability problems can be solved by using two different methods. Packet truncation and data corruption can be detected by using two fields in the message's header: a packet length and a checksum. Packet losses can be detected by timestamping the outgoing request. Upon each subsequent operation or poll of the network interface, one entry of the request table is examined. If any stale requests are found, they are retransmitted. Reply AMs are not stamped. If a reply is dropped, the resulting action is to retransmit the request. This presents a problem because the corresponding request handler could be executed twice. The solution is as follows: when an invalid request (its instance number matches that of reply buffer) is received, the corresponding reply is simply retransmitted without invoking the handler[9].

Clearly, this algorithm allows only one outstanding request and reply per node. As a result, the protocol is not capable of achieving maximum bandwidth. The goal of any communication protocol is to hide the latency of accessing the media by keeping multiple messages *in the network.* This is accomplished here by replicating the one-way protocol N times for each node, where N corresponds to the network depth. This value usually needs to be discovered by experimentation for each networking card, operating system, and access method. For example, memory-mapped register operations to the network device are much more efficient than traps to a kernel networking stack. Since processing an active message requires matching requests with replies and vice versa, a multibit instance number is introduced to the message. Now, for each node in the host pool, N buffers are allocated, and each is associated with one instance of the one-way protocol.

4 Implementation

This section addresses several issues involved in implementing the AM protocol efficiently.

4.1 The Transport

Before any code is written, the matter of the underlying communication mechanism must be resolved. Since further experimentation is necessary to determine which of the above approaches is most efficient and portable, a transport will not be explicitly named. Instead, two requirements will be made of any method chosen as the basis for an AM implementation. First, the transport must be connectionless. One that is connection-oriented does not map well to the all-to-all communications model found in distributed computing. Personalized communication typically requires many system resources and has longer startup times. A connectionless model will thus be assumed—where data and the sender's address appear only at one "interface" to the network. The second requirement is that the communication layer provide for nonblocking operation. Overlapping computation and communication are central in active messaging and would simply not be possible with blocking communication primitives. Note that asynchronous notification of message delivery is not mandatory. This can be accomplished, sometimes much more efficiently, through explicit polling of the interface.

4.2 Differing Address Space

In PVM and similar heterogeneous computing environments, a process's address space is not necessarily the same from machine to machine. Given this, the active messaging library must have some machine-independant way of specifying a location in either text or data memory. A common solution to this problem is to introduce the concept of a binding. A machine-independent identifier can be "bound" to a specific address a process participating in communication. In order to avoid the time to search a table, the AM library can provide a call to allocate these identifiers from a linear progression corresponding to the indices of an array. Before any communication is done, the user will have to store the desired address into the array position pointed to by the identifier. In this way, the address will be bound. Any incoming message wishing to write in the local hosts memory must decode this identifier into the corresponding virtual address. Extending this concept, three separate tables can be maintained: one for the addresses of request handlers, one for reply handlers, and one for data addresses. Thus, one address identifier can be associated with three different addresses and resolved by the context in which it is used. C macros can be provided to reference each of these tables, reducing unnecessary subroutine calls.

4.3 Handler Execution

In the traditional networking stacks, asynchronous notification is accomplished by using signal driven I/O. Upon arrival of a message, the OS can deliver a previously defined signal to designated process. In the AM implementation, this process will register an active message dispatch handler. This handler will first check whether data is waiting to be read, since anyone could have delivered the signal. Next, it will look up the address of the corresponding request handler, using the address identifier as an offset into the array of request handler addresses. If the tag is not bound to a request handler function, a "sink" function should be executed that simply removes the packet from the network. If the tag is bound, the dispatcher invokes the request handler with any arguments optionally packed into the message and a pointer to the message itself. The user's handler is then allowed to run to completion, optionally sending a reply AM to the originator.

It is conceivable that the user would wish to have option of declaring critical sections. This is especially important to consider when the user's process is performing system calls, most of which can be interrupted by a signal. Guarding against this situation usually requires watching for a special return code. If this code is returned, the call must be invoked until a different code returned. Systems conforming to the 4.2 BSD release of Unix do this automatically. However, given the wide variety of platforms we wish to support, this cannot be relied upon. If the user does choose to make system calls while communication is taking place, those calls must not reference any buffer whose contents are suspect. Aside from the nuisance to the user, repeated system calls can cause unnecessary process suspension and context switching delays. Two solutions to this problem are available. The first is a true lockout, where the kernel actually prevents the signal from being delivered until the user indicates that it is ok to do so. The second solution, recommended for debugging and I/O-intensive applications, is to do away with asynchronous notification completely and poll for all messages. Currently, all active messaging implementations used polled I/O, mainly because the polling function also performs the check for stale requests. Where polling is chosen over interrupts, the request and reply calls should automatically poll[1] the network at the beginning

[1] To poll in this context has two meanings: to check the request table for stale entries and

and end of every call. However, the user must be careful not to ignore the network in computation-only loops. Hence, an explicit call to poll the interface is provided. To achieve widespread acceptance, the burden on the user of AMs must be minimal. Thus, a function could be provided that can register an alarm handler that performs this polling on a regular interval.

Theoretically, active messages are executed immediately upon arrival. As mentioned before, these messages simply remove the data from the network and integrate it into the ongoing computation. An arbitrary time limit could be imposed on the handlers to prevent deadlock and aid in debugging. However, this should be available only as a compile time switch to the library, since the management of many alarms is quite expensive.

4.4 Reliability

To realize maximum bandwidth with a low-level transport demands a simple and efficient implementation of reliable data delivery. When using a completely unreliable transport, timers and data verification fields must be used in addition to those for protocol management. The above protocol requires only two fields in the header: the message type and the protocol instance number. The type actually requires only one bit indicating whether the message is a request or a reply. The protocol instance number is necessary in order to match a reply with the corresponding request. Remember that with the N-way protocol, multiple requests can be outstanding for the same node. Explicit alarms can be avoided by timestamping the copy of each transmitted request. Note that the above protocol requires that only outgoing requests be stamped, since they are the only messages being monitored for loss. Timestamps should be obtained through the most efficient means possible. This is often very different from platform to platform. On some platforms for example, interval timers are less expensive to read compared to the `gettimeofday()` system call [9]. The interval between successive retransmission of stale requests can be exponential with an arbitrary limit signifying node failure.

To maintain data integrity requires the insertion of the length and a packet-wide checksum into the message's header. Upon reception, if the computed values do not match, the message is discarded. This approach may not be necessary, however, since many network transports provide this service automatically.

4.5 Buffering

Buffering is often the cause of large delays in network computing. There are two reasons for such delays. The first is that the data from the network is not processed immediately upon reception. The data is read from the device and held in kernel space until the user's process issues a read trap.

to check the network interface for data.

The second reason stems from the problems of guaranteed data delivery. Any process using an unreliable transport must maintain a copy of the outgoing packet until an acknowledgement is received. Current implementations of AM over LANs solve these problems by and by preallocating packet buffers of maximum permissible size on the device itself. It is left to the user, to fragment and order the data accordingly. This strategy places a fairly large burden upon the user. The transport chosen for an AM layer may allow an unlimited packet length but still not guarantee delivery. A true solution to this problem does not currently exist. The only option the implementer has in this situation is to provide the user with a call to specify the maximum buffer size before any communication is allowed. Dynamic memory allocation upon message reception would introduce too much overhead.

4.6 MPP Portability

Writing an AM layer is a trivial task on most massive parallel machines. All machines supported by PVM include a reliable delivery interconnection network and provide a nonblocking send-and-receive API. This makes for a very simple and fast port of the AM library, mainly consisting of `ifdef`ing the code for reliability. As a bonus, many of the supported architectures come furnished with an optimized AM library. In this case, the task of the AM implementer would simply be to write stubs calling the existing AM API.

5 Optimization

In order to reduce the cost of context switching and signal delivery delays in the kernel, the AM dispatch handler can be optimized. Once the user's handler has executed and returned to the dispatcher, the network interface is again checked for data. If another message is received, the user's handler will again be called and the entire process repeated. If no data is available, the user's process is allowed to continue processing where it left off.

A nice optimization can be made regarding the search of the request table for stale timestamps. Instead of scanning the entire table during each operation, only one entry need be examined. The major motivation is that the pattern of reference to this table is cyclic during normal operation. As long as the user is communicating or periodically checking this table during computation, the data is guaranteed to be delivered.

One can easily imagine a situation where a cluster of nodes is performing some computation and then pushing data to each of their neighbors. This type of operation requires only that data be sent during the request phase. Even on a small network, this could easily result in considerable duplicate processing in the form of a reply-request loop, with each requiring the construction of a separate message. A simple optimization is to "piggyback" replies whenever possible. Thus, with the addition of one field to the request message—the reply instance number—we have reduced the message-formatting work by 50 percent.

6 Conclusion

Overall, active messaging could prove to be a worthwhile addition to the PVM suite if done carefully. However, an AM implementation *will not* provide an overwhelming "out-of-the-box" improvement over PVM's direct routed `psend` and `precv` APIs. AM *will* provide the users the tools to redesign their applications to communicate more efficiently. The asynchronous nature of AM combined with overlapped communication and computation can lead to orders of magnitude speedup, especially on algorithms that interleave massive data movement and computation (FFT for example). The most likely option for a first implementation of AM would be to use UDP or some relatively simple device-specific API. This would allow detailed study and suggest further optimizations to be made to the model and its algorithms. Successive experiments could then be performed using different network access methods until an acceptable balance of portability and performance is obtained. If it can be shown that the AM model can support PVM's level of functionality and that active messages routinely generate more robust and efficient distributed applications, then the issue of the conversion of the PVM suite to use this paradigm should be seriously considered.

References

[1] Chran-Ham Chang, Richard Flower, John Forecast, Heather Grey, William R. Hawe, K. K. Ramakrishnan, Ashok P. Nadharni, Uttam N. Shikarpur, and Kathleen M. Wilde. High Performance TCP/IP and UDP/IP Networking in DEC OSF/1 for Alpha AXP. *Digital Technical Journal*, 5(1), Winter 1993.

[2] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP*, volume II. Prentice Hall, Englewood Cliffs, NJ 07632, 2nd edition, 1994.

[3] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *Proceedings of the Fourth ACM Symposium on Principles and Practice of Parallel Programming*, May 1993.

[4] David E. Culler, Kim Keeton, Cedric Krumbein, Lok T. Liu, Alan Mainwaring, Richard P. Martin, Kristin Wright, and Chad Yoshikawa. Generic Active Message Interface Specification. Technical report, Department of Computer Science, University of California, Berkeley, November 1994.

[5] Digital Equipment Corporation. *OSF/1 Guide to the Data Link Interface*, revision 1.2 edition, 1993.

[6] Information Networks Division, Hewlett Packard Company. *Netperf: A Network Performance Benchmark*, revision 1.9alpha edition, August 1994.

[7] Jonathan Kay and Joseph Pasquale. A Performance Analysis of TCP/IP and UDP/IP Networking Software for the DECstation 5000. Technical report, University of California, San Diego, 1992.

[8] Lok T. Liu, Alan Mainwaring, and Chad Yoshikawa. White Paper on Building TCP/IP Active Messages. Technical report, Department of Computer Science, University of California, Berkeley, November 1994.

[9] Richard P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. Technical report, Department of Computer Science, University of California, Berkeley, 1994.

[10] Neal Nuckolls. *How to Use DLPI.* Internet Engineering, June 1992.

[11] *Proceedings of the 19th International Symposium of Computer Architecture*, May 1992.

[12] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, Englewood Cliffs, NJ 07632, 1990.

[13] UNIX International, OSI Work Group. *Data Link Provider Interface Specification*, revision 2.0.0 edition, August 1991.

[14] Thorsten von Eicken. Building Parallel Programming Models using Active Messages. Technical report, Department of Computer Science, Cornell University, 1994.

[15] Thorsten von Eicken, Veena Avula, Anindya Basu, and Vineet Buch. Low-Latency Communication over ATM Networks using Active Messages. *Proceedings of Hot Interconnects II*, August 1994.

[16] Thorsten von Eicken and David E. Culler. Building Communication Paradigms with the CM-5 Active Message layer (CMAM). Technical report, Department of Computer Science, University of California, Berkeley, July 1992.

[17] Thorsten von Eicken, David E. Cullerand Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In pica [11].

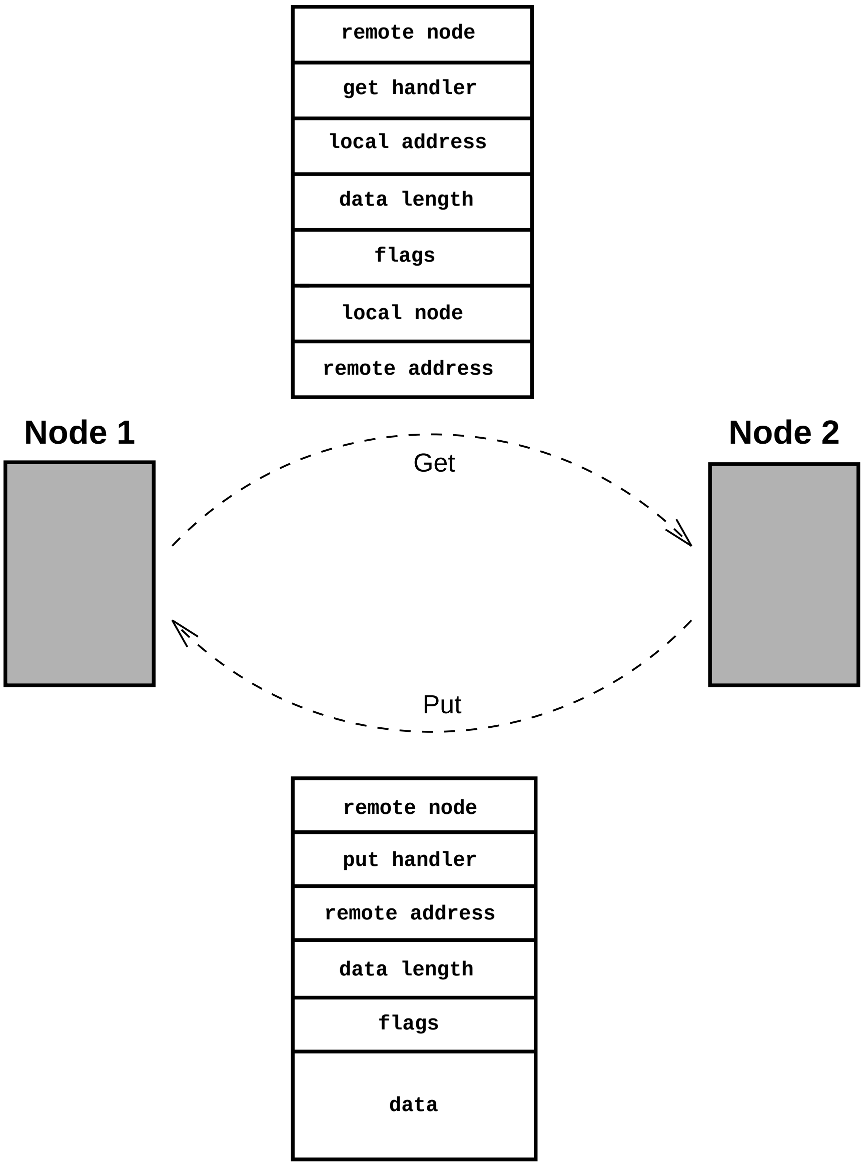

Figure 1: Using active messages to implement remote memory access [3]

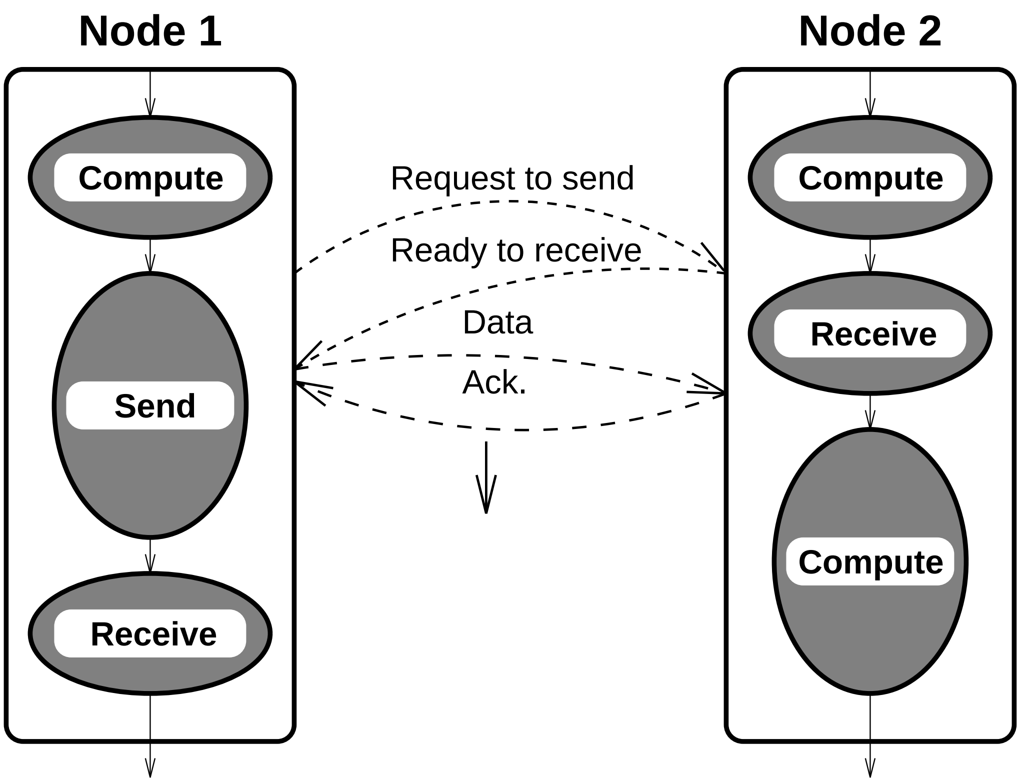

Figure 2: A deadlock-free blocking three-phase communication protocol [3]