Low Latency Workstation Cluster Communications Using Sender-Based Protocols [1]

Mark R. Swanson
Leigh B. Stoller
E-mail: {swanson,stoller}@cs.utah.edu

UUCS-96-001

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

January 24, 1996

Abstract

The use of workstations on a local area network to form scalable multicomputers has become quite common. A serious performance bottleneck in such "carpet clusters" is the communication protocol that is used to send data between nodes. We report on the design and implementation of a class of communication protocols, known as sender-based, in which the sender specifies the locations at which messages are placed in the receiver's address space. The protocols are shown to deliver near-link latency and near-link bandwidth using Medusa FDDI controllers, within the BSD 4.3 and HP-UX 9.01 operating systems. The protocols are also shown to be flexible and powerful enough to support common distributed programming models, including but not limited to RPC, while maintaining expected standards of system and application security and integrity.

[1] This work was supported by a grant from Hewlett-Packard, and by the Space and Naval Warfare Systems Command (SPAWAR) and Advanced Research Projects Agency (ARPA), Communication and Memory Architectures for Scalable Parallel Computing, ARPA order #B990 under SPAWAR contract #N00039-95-C-0018

Contents

1 Introduction

The use of workstations connected by a local area network to form low-priced, incrementally scalable multicomputers has become common. One factor driving this development is the increasing availability of low latency, high bandwidth interconnection fabrics. The latency of these fabrics is sufficiently low that the time spent even in the control processing portion of standard general purpose protocols, notably IP-based protocols such as TCP and UDP, will soon dwarf the transmission time[2]. Examples of the fabrics we consider potentially viable are Fibre Channel[6] and R2[5]. Our target systems, clusters of commodity workstations running essentially standard operating systems, rules out approaches such as those taken by Alewife[8], Typhoon[12], *T[11], or MDP[3], which rely on custom processors and/or non-standard operating systems.

Continued reliance on standard protocols can impose unnecessary communication costs. The services required by applications on these clusters are often far more modest than those provided by the standard protocols. For example, consider a client-server application. As each client issues a request to the server, it waits for an explicit reply. It does not need a separate acknowledgment of its request, as the server's reply will implicitly provide that acknowledgment. The problem is not the additional network traffic, but rather the software overhead of generating and handling the unneeded acknowledgments.

Another way that the standard protocols impose unnecessary costs is in loss of information across the protocol layers. For example, message-based applications are often implemented on top of TCP, a stream protocol. It is common to see code that explicitly inserts the message length in the data stream at the start of each message to allow the receiver to re-impose a message structure on the received stream. It is also common for such receivers to embed their receive operations in a loop to ensure acquisition of an entire message, since the stream may well provide the message to the receiver in parts rather than as a whole. The added cost of extra system calls and potential context switches can directly impact latency.

A third area of potential waste is insensitivity of the standard protocols to the characteristics of the given interconnect. For example, protocol software should not need to perform costly operations to ensure in-order delivery when using a medium that provides that guarantee. Similarly, the protocol software should not generate acknowledgments if hardware acknowledgments are implemented and visible to the software.

Regardless of the semantics of a given protocol, its performance is generally more strongly influenced by its specific implementation than the semantics it imposes. Even relatively complex protocols can be made efficient through careful implementation[13, 10]. One approach to reducing protocol handling overhead is to allow the application to perform high frequency operations, usually sending and receiving messages, directly, without involving the OS[9]. While this approach can produce good results, it is often difficult to ensure security and fairness within a general purpose computing environment. Another approach, which we have pursued, is to leave these functions in the OS, but to engineer efficient implementations of them. Specifically, we are investigating "sender-based" protocols and their implementation within production operating systems.

The sender-based model relies on the ability to specify the location within a receiver's address space, where the packets of a message will be placed. In this manner, messages are guaranteed to fit, thus avoiding the need for costly buffering operations. Further, the semantics of sender-based protocols result in extremely low control processing overhead. An initial prototype using an FDDI ring has shown that an efficient implementation of sender-based protocols can result in near-link latencies of 47 microseconds for a minimal remote procedure call (RPC), and near-link bandwidth of 10.2 megabytes per second, between user processes.

In Section 2, we discuss sender-based protocols in detail. Then, in Section 3, we describe an efficient implementation of these protocols within the operating system. Low-level implementation concerns are addressed in Section 4. In Section 5, the use of sample applications to validate the protocols is discussed. Section 6 presents timings of a prototype implementation.

2 Sender-Based Protocols

The core concept of sender-based[14] protocols lies in the specification, by the sender, of the location at which a message is to be placed in the receiver's memory. This specification enables the receiving end to place the (packets comprising a) message directly into a user-space buffer. Others[4, 2] have shown that avoiding copying of large messages is of crucial importance given the limited memory bandwidths in modern workstations. At the device driver level, the cost of buffer allocation is avoided, and at the kernel level the cost of copying from a kernel buffer into user space is saved.

The address specification provided by the sender implies that the sender manages a buffer within the receiver's memory. Each packet of a message that it sends is tagged with a connection identifier. Each connection has an associated receive buffer that is part of the user's address space. Each packet also contains an offset within the receive buffer at which to place the message. The receiving entity, either a "smart" hardware interface or the device driver software, need only ensure that the message lies within the specified buffer, which involves a couple of simple range checks. A correct sender, i.e., one that always specifies correct offsets and sizes, will never have a message rejected by the receiver due to buffering difficulties.

Direct copying of the message into the user's space requires that the receiving entity (hardware interface or software device driver) be able to identify the receiving process, or at least its buffer, from the packets. We accomplish this with the connection identifier. A modest extension to this knowledge about the receiving process also allows the receiving entity to post notifications of message arrival directly to a *notification queue* that the receiving process may poll or wait upon.

Another potential benefit of using pre-allocated buffers for message receipt is that relatively simple hardware can be implemented to perform reassembly of messages from their constituent packets. With the small size of packets in many emerging communications fabrics, this may prove to be crucial to economical communication by avoiding the need for operating system intervention on arrival of the many small packets comprising large messages.

Because the receiving entity deposits packets directly into user buffers, that entity must provide a measure of security for the receiving process. As stated above, bounds checks are performed to prevent messages from being deposited in locations not explicitly specified as buffers. A further level of protection is needed to ensure that only qualified senders are allowed to direct messages to a given endpoint. Barring the availability of a complex device interface to impose such security, restricting device access to trusted operating system routines is the most expeditious method of restricting senders to authorized connections. Separate header checksums are provided since the body of the packet will be moved directly into memory using the header information. An error in the header that was only identified when the entire body had been processed would thus be unacceptable.

2.1 A Realization of a Sender-Based Protocol

Going from the concept of sender-based protocols to a working protocol requires specification of many details:

- connection establishment and teardown;
- buffer management;
- notification of message arrival;
- system integrity and security;
- exploiting interconnect characteristics;
- exceptional conditions.

All of these will be addressed in this section in the full paper.

3 Engineering Efficient Operating System Services

The prototype was implemented within two versions of Unix, the BSD 4.3 kernel ported to the PA-RISC at Utah and a production kernel for PA-RISC machines distributed by Hewlett-Packard, HP-UX 9.01. The two kernels are structurally very similar, since they share common roots. They have diverged significantly at the detailed level, however. In both cases, those architectural features of the PA-RISC[7] that facilitated increased efficiency were employed. We assume any serious attempt to achieve low latency communication will need to be sensitive to the capabilities of the host system.

Two performance-enhancing modifications to normal operating system facilities are described here. We shall argue, in the complete paper, why these are reasonable and safe things to do for systems that remain essentially general-purpose machines.

3.1 Lightweight System Calls

One major contributor to communication latency, on both send and receive sides, is the overhead of performing a system call. As noted above, we have chosen not to rely on user-mode send and receive operations, and thereby avoid system call overhead. Instead we have attacked the problem of expensive system calls directly. Figure 1 lists the time to perform an "empty" normal system call in each of the systems.

We have implemented send and receive as *lightweight* system calls; in the same systems, the time for an "empty" lightweight system call is 18 cycles. The major savings come in two areas:

Operating System	BSD		HP-UX	
Component	Cycles	Usecs.	Cycles	Usecs.
entry	357	5.4	523	7.9
exit	208	3.2	334	5.1
Total	565	8.6	857	13.0

Figure 1: System call entry/exit times for an empty system call.

1. The amount of state (registers, context) that must be saved/restored when switching between user and kernel mode is significant in a full system call. The state goes onto a kernel stack and is later restored involving a large number of memory references and associated cache misses.

2. The generic system call interface has to decode the system call number, arrange for the arguments to be moved into the kernel's address space, and must otherwise handle scheduling and signal delivery, when appropriate.

A lightweight system call has state-saving requirements similar to those of a procedure call. The primary cost lies in instantiating the kernel's addressing context, which on the PA-RISC architecture is accomplished by a "gate" instruction, which sets the value of a single control register. The user's addressing context remains in force, facilitating direct access to the system call arguments, as well as the message data, which can be copied directly into the network device.

This system call mechanism is appropriate for the common case of simple operations. In the event that a more complex kernel service is required, e.g., the process needs to `sleep` in the event that there is no data to receive, the process simply proceeds on through the full system call interface.

3.2 Fast Path Interrupt Handling

On the receive end, a major portion of communication cost lies in interrupt service. On the PA-RISC, a network device posts an external interrupt on packet arrival, which is handled by the generic interrupt handling code in the kernel. The time to process such an interrupt is between 20 and 25 microseconds (for a one word message), depending on the state of the cache when the context for the currently running process is saved.

We have re-engineered the low-level external interrupt handler to deal directly with most network interrupts, reducing the interrupt handling time to just over seven microseconds for an incoming message consisting of one word of data.

4 Efficient Micro-level Implementation

On a workstation such as those used by our prototype, a cache miss is very expensive in terms of processor cycles (30 cycles for the HP 730, nearly .5 microseconds). The high cost of cache misses argues for protocols to be designed so that they can be implemented with small data structures occupying as few cache lines as possible. Keeping the per-connection data structures small also allows us to allocate connection descriptors in a linear array and index directly into it. In software, this saves search time and memory accesses compared to approaches that must use more complex lookup schemes (e.g., TCP with its very long connection identifiers).

A second influence that has led to the same economical implementation of the protocols is the desire to implement some part of the protocol in hardware. The expected "budget" for such hardware is limited, and as a result, any data structures that it would need to cache or store onboard must be small.

The primary protocol data structures will be discussed in the full paper, with an emphasis on their effect on cache and bus transaction costs.

5 Example Applications

As an initial test of the correctness of our prototype and as a measure of the adequacy of the protocols, we performed a trivial port of PVM to our prototype. In so doing, we observed some of the expected effects of pushing some transport responsibilities up to user level. These effects and those arising in further applications testing will be described in the full paper. Our next application is a port, currently underway, of the HP-UX X11 server and library to the sender-based protocols.

6 Experimental Results

We present some experimental results based on our prototype implementation. We used a cluster of from 2 to 5 HP730 workstations, which employ a 66 MHz PA-RISC 1.1 cpu with off-chip, single level 128KB instruction and 256KB data caches. The operating systems were our modified versions of HP-UX 9.01 and BSD 4.3. The workstations were connected to an FDDI ring using Medusa[1] interfaces; no other machines were on the ring. We will present two sets of measurements:

- basic timings of the interconnect;
- micro-benchmarks of our protocols.

6.1 Basic Interconnect Characteristics

At the device level, our prototype is currently implemented for the Medusa FDDI controller. The interesting characteristics of the Medusa for this discussion are:

- the controller has 1 megabyte of on-board memory which can be divided into 128 buffers;
- the controller has two queues, one for transmitting and the other for receiving;
- the controller (and its queues and buffers) are accessible to the processor via load and store operations to IO space addresses.

We have measured the round-trip packet time on a two node ring with this controller as 19.5 microseconds (1286 cycles @ 66MHz). This includes only the controller and transmission time, and

Activity	Instructions	Cycles (HP-UX)	Cycles BSD	CPI BSD
Lightweight system call entry	9 (0)	14	14	1.5
Decode and process arguments	25 (3)	33	33	1.5
Read Medusa transmit queue	5 (2)	37	39	12.3
Read/Write 6 word FDDI header	18 (2)	195	101	6.2
Write 1 word message body	12 (4)	29	28	3.0
Write 7 word protocol header	33 (4)	62	61	1.9
Write Medusa transmit queue	3 (2)	7	7	5.0
Total	105 (17)	377	283	3.0

Figure 2: Cost of sending a 4 byte message.

provides a base figure for the best possible round-trip RPC time, and a lower bound on the number of instructions needed to send and receive a packet.

6.2 Performance of Protocol Primitives

Figure 2 shows the costs of sending a message containing one word of data, with checksumming of the header and body. Of the total instruction count of 105 instructions, 17 instructions are measurement related (given in parentheses) and 25 are Medusa or FDDI specific. The total of non-device specific, non-timing instructions is thus 63.

A non-checksumming implementation uses 14 fewer instructions. Since this packet had a payload of only 1 word, only 4 of the additional instructions are for checksumming the data. The rest are used to checksum the header. Separate header checksumming is performed so that the header information can be reliably used for packet delivery before the entire packet has been processed.

Figure 3 shows the cost of sending and receiving a message in one direction (one half of an RPC). All times were measured using an interval timer[2] except for "Interrupt Handler (control)", where instruction counts and an estimated Cycles per Instruction were used.

The send path time of 4.2 microseconds includes the system call entry code and the user system call stub. As indicated in Figure 2, the send path proper includes validating arguments from the user, obtaining a transmit buffer identifier from the Medusa, formatting the packet header, performing checksums, copying the payload to the Medusa buffer, and queuing the transmit buffer identifier for sending.

The receive path is the time spent after the lightweight system call (receive) detects the arrival of a message, via a *notification* posted in memory by the interrupt handler, to the time it jumps back to user mode. This time is spent accessing and updating a connection state block and writing a connection identifier, message address, and message size into locations provided in the send call.

The total time for an RPC is thus 44 microseconds, though this is not user-to-user. Our measured round-trip RPC time, user-to-user, is 47 microseconds. The additional 3 microseconds is spent

[2]The PA-RISC contains a control register that is incremented, in the case of the HP 730, at each CPU cycle.

Component	Time (Usecs.)
Send Path	4.2
Receive Path	1.0
Interrupt Handler	5.3
Interrupt Handler (control)	2.0
Controller (and on the wire)	9.5
Total	22.0

Figure 3: Break down of times for a one-way message.

Type	Throughput (MB/second)
Memory Copy	10.2
Filesystem Copy	7.8

Figure 4: Measurements of network throughput.

in entering/leaving lightweight system calls and in user-mode system call stubs. Measurements of an earlier version of RPC using standard interrupt paths for the Medusa gave an RPC time of 85 microseconds. The change in interrupt handling alone resulted in a savings of 38 microseconds or 44% total RPC time.

Figure 4 shows the measured throughput for two test cases, each of which involved sending a large (64 megabyte) block of data from a sender to a receiver. In the first test, the data was simply moved from the sender's address space to the receiver's address space. In the second test, the copy was performed through the filesystem (but not to disk) by reading from a /dev/zero pseudo device on the sender side, and writing to /dev/null on the receiver side. In both cases, a significant percentage of the Medusa's 12.5 megabyte per second bandwidth was achieved.

7 Conclusions

A realization of a sender-based protocol has been developed and prototyped. As hoped, it has delivered communications latencies that are a reasonable match to the physical layer latencies of emerging mass-production interconnects such as FDDI, Fibre Channel, and R2. The achieved latencies are as much a product of an aggressive engineering of the software, especially the operating system components, as they are due to the protocol design. With the availability of low-latency interconnects, we expect to see round-trip times of 30-35 microseconds. Following that, specification of appropriate communications interfaces for these interconnects, should provide even smaller total end-to-end latencies, in the range of 20 microseconds or less. At this level, the contribution of communications to latency for any but the most trivial interactions will be negligible compared to user-level processing.

References

[1] Banks, D., and Prudence, M. A High-Performance Network Architecture for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communications 11*, 2 (February 1993), 191–202.

[2] Clark, D., Jacobson, V., Romkey, J., and Salwen, H. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine 11*, 2 (June 1989), 23–29.

[3] Dally, W., et al. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro 12*, 2 (April 1992), 23–39.

[4] Dalton, C., Watson, G., Banks, D., Calamvokis, C., Edwards, A., and Lumley, J. Afterburner: A Network-Independent Card Provides Architectural Support for High-Performance Protocols. *IEEE Network* (July 1993), 36–43.

[5] Davis, A., Cherkasova, L., Kotov, V., Robinson, I., and Rokicki, T. R2 - A Damped Adaptive Multiprocessor Interconnection Component. In *Proceedings of the University of Washington Conference on Multiprocessor Interconnects* (May 1994).

[6] Fibre Channel Association. *Fibre Channel: Connection to the Future*, 1994.

[7] Hewlett-Packard Co. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, February 1994.

[8] Kubiatowicz, J., and Agarwal, A. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 7th ACM Internatonal Conference on Supercomputing* (July 1993).

[9] Maeda, C., and Bershad, B. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles* (December 1993), pp. 244–255.

[10] Mckenney, P. E., and Dove, K. F. Efficient Demultiplexing of Incoming TCP Packets. In *Proceedings of the 1992 Conference on Communications Architectures, Protocols, and Applications* (August 1992), pp. 269–279.

[11] Nikhil, R. S., Papdopoulous, G., and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture* (May 1992), pp. 156–167.

[12] Reinhardt, S., Larus, J., and Wood, D. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture* (April 1994), pp. 325–336.

[13] Thekkath, C., and Levy, H. Limits to Low-Latency Communications on High-Speed Networks. *ACM Transactions on Computer Systems 11*, 2 (May 1993), 179–203.

[14] Wilkes, J. Hamlyn - an interface for sender-based communication. Tech. Rep. HPL-OSR-92-13, Hewlett-Packard Research Laboratory, November 1992.