PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcast Domains

Jehoshua Bruck* Danny Dolev† Ching-Tien Ho‡ Rimon Orni§ Ray Strong‡

*California Institute of Technology
Mail Code 116-81
Pasadena, CA 91125
bruck@systems.caltech.edu

†Institute of CS
Hebrew University
Jerusalem, Israel
dolev@cs.huji.ac.il

‡IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
{ho,strong}@almaden.ibm.com

§University of Maryland
Institute of Advanced Computer Studies
College Park, MD 20742
rimon@umiacs.umd.edu

Abstract

Existing programming environments for clusters are typically built on top of a point-to-point communication layer (send and receive) over local area networks (LANs) and, as a result, suffer from poor performance in the collective communication part. For example, a broadcast that is implemented using a TCP/IP protocol (which is a point-to-point protocol) over a LAN is obviously inefficient as it is not utilizing the fact that the LAN is a broadcast medium. We have observed that the main difference between a distributed computing paradigm and a message passing parallel computing paradigm is that, in a distributed environment the activity of every processor is independent while in a parallel environment the collection of the user-communication layers in the processors can be modeled as a single global program. *We have formalized the requirements by defining the notion of a* correct global program. *This notion provides a precise specification of the interface between the transport layer and the user-communication layer. We have developed PCODE, a new communication protocol that is driven by a* global program, *and proved its correctness.*

We have implemented the PCODE protocol on a collection of IBM RS/6000 workstations and on a collection of Silicon Graphics Indigo workstations, both communicating via UDP broadcast. The experimental results we obtained indicate that the performance advantage of PCODE over the current point-to-point approach (TCP) can be as high as an order of magnitude on a cluster of 16 workstations.

*Supported in part by the NSF Young Investigator Award CCR-9457811, by a grant from the IBM Almaden Research Center, San Jose, California and by a grant from the AT&T Foundation.

1 Introduction

Parallel computing on clusters of workstations and personal computers has very high potential, since it leverages existing hardware and software. In fact, there are a number of existing commercial parallel programming environments that can run on top of clusters of workstations [3, 11, 15, 18].

Parallel programming environments offer the user a convenient way to express parallel computation and communication. The communication part consists of the usual point-to-point communication as well as collective communication. Examples of collective communication operations include one-to-all broadcast, all-to-all broadcast, global combine operation, scatter and gather.

The need for collective communication arises frequently in parallel computation. Collective communication operations simplify the programming of applications for parallel computers, facilitate the implementation of efficient communication schemes on various machines, promote the portability of applications across different architectures, and reflect conceptual grouping of processes. In particular, collective communication is extensively used in many scientific applications for which the interleaving of stages of local computations with stages of global communication is possible (see [10]). Collective communication routines can operate over the entire set of processes that are created at the beginning of an application or over user-specified groups of processes [4, 14].

However, existing programming environments for clusters are built on top of a point-to-point communication layer (send and receive) over local area networks (LANs) and, as a result, suffer from poor communication performance. For example, a broadcast that is implemented using a TCP/IP protocol (which is a

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

"reliable" point-to-point protocol) over a LAN is obviously inefficient as it is not utilizing the fact that the LAN is a broadcast medium.

The system model that we consider in this paper consists of a set of processors that communicate via asynchronous and unreliable broadcast messages. A processor has three logical layers of software (see Figure 1). The lowest layer is a LAN-communication layer, typically a User Datagram Protocol (UDP), that interfaces the LAN. The second layer is the transport layer (this is where our new protocol fits). The upper layer is the user-communication layer, which in our case is a set of collective communication routines of a parallel programming environment.

Our goal is to create a transport layer which utilizes the fact that a LAN is a broadcast domain and to make the collective communication part of a parallel programming environment more efficient. The challenge in achieving this goal is that the LAN-communication facility within a broadcast domain, typically a User Datagram Protocol (UDP), is unreliable. We make use of special properties of the parallel programming environment in order to save in communication cost, in code complexity, and in CPU overhead.

Reliable broadcast in distributed systems is a topic that has been studied extensively for more than a decade [12]. In fact, there are a number of existing projects and systems that provide a reliable transport layer as well as other services for distributed computing. Examples are the V system [8], ISIS [5], Psync [16], Amoeba [19], Trans [13], Transis [1] and Totem [2]. However, we have observed that the properties required from the user-communication layer associated with reliable broadcast protocols for *distributed systems* are different from the properties of the user-communication layer associated with *parallel systems*.

The main contributions of the paper are:

- We have studied the requirements associated with collective communication for parallel computing. We have observed that the main difference between a distributed computing paradigm and a message passing parallel computing paradigm is that, in a distributed environment the activity of every processor is independent while in a parallel environment the collection of the user-communication layers in the processors can be modeled as *a single global program*. Also, the typical fault model in parallel computing (which is the fault model we will be assuming) is that if a single processor fails then the execution stops and the recovery is handled by global techniques (such as check-pointing) at the application layer, and not at the communication layer. In distributed computing environments a message that is received from the network by the transport layer is *delivered* to the user-communication layer. The notion of *delivered* has a different meaning in parallel computing environments, where a message is expected by the receivers. Namely, a message is not just *delivered to* but also *requested by* the user-communication layers at the receivers.

- We have formalized the requirements by defining the notion of a *correct global program*. This notion provides a precise specification of the interface between the transport layer and the user-communication layer. We also formally defined the interface between the transport layer and the LAN-communication layer. We note here that the notion of a global program fits well with the notion of a Single Program Multiple Data (SPMD) in parallel computing. It allows concurrent execution of point-to-point communication as well as communication over groups of processors.

- We have developed a new communication protocol that is driven by a *global program*, and proved its correctness. The protocol has a number of new ingredients that take advantage of the fact that (i) we have a global program and (ii) that the communication layer is a broadcast domain. We call this new protocol PCODE, for Parallel Computing On Distributed Environments.

- We have implemented the PCODE protocol on a collection of IBM RS/6000 workstations and on a collection of Silicon Graphics Indigo workstations, both communicating via UDP broadcast. The experimental results we obtained indicate that the performance advantage of PCODE over the current point-to-point approach (TCP) can be as high as an order of magnitude on a cluster of 16 workstations.

The paper is organized as follows. In the next section, we present the properties and requirements associated with parallel computing, define the notion of a *correct global program* and specify the properties of the LAN-communication layer. The description of PCODE is done in three steps. First, in Section 3 we describe a simple protocol (that assumes infinite buffers). Next, in Section 4 we extend the simple protocol to a more practical protocol (it uses finite buffers). The proofs of correctness of both protocols are omitted here due to space limitation. In Section 5 we indicate additional extensions and ideas in PCODE that facilitate performance improvements. In Section 6 we describe the environment we have created to conduct performance evaluation of PCODE and present experimental results. Finally, Section 7 concludes the paper.

2 Formalization of the Model

In this section we will formally describe the computation/communication model of the distributed/parallel system that we are interested in. We then, in the next two sections, will use the model to prove the correctness of our protocols. The system consists of processors that communicate via asynchronous and unreliable broadcast messages. Although we expect that some messages might be lost, we assume that the content of a received message is not corrupted. A processor has three logical layers of software (see Figure 1).

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

The lowest layer is a LAN-communication layer (typically UDP) that interfaces the LAN. The second layer is the transport layer (this is where our new protocol fits). The upper layer is the user-communication layer, which in our case is a set of collective communication routines of a parallel programming environment. We will describe the upper and lower interfaces to the transport layer and then specify the properties of the transport layer.

2.1 The Global Program

The calls that the user-communication layer at each node can make to the transport layer are either *multicast* or *request*. We model the collection of calls to the transport layer made by all the processors in the system as a *single global program*. The function *Program* maps $Processors \times PositiveIntegers$ into a set *Calls* consisting of all possible transport layer calls from the user-communication layers and the null operation that we call *skip*. More specifically, for each processor p and each positive integer i, we assume that $Program(p, i)$ has one of the following three forms:

- $multicast_p(m, T)$ where m is a message and T is a nonempty set of processors (not including p) to receive the multicast message m from p.
- $request_p(q, T)$ where $p \neq q$ is a processor to receive a multicast message from q, and T is a set of processors including p but not q.
- *skip*.

Each processor p is executing the same program and making the calls $Program(p, i)$, starting with $i = 1$ and incrementing i by 1 after each call. Thus, the index i identifies the execution order within each processor.

A *global Program* is said to be *correct* if the following three assumptions are satisfied:

Assumption 1 (Matched Calls) *if* $Program(p, i) = multicast_p(m, T)$ *then* "$q \in T \Rightarrow Program(q, i) = request_q(p, T)$" *and* "$q \notin (T \cup \{p\}) \Rightarrow Program(q, i) = skip$"; *and if* $Program(q, i) = request_q(p, T)$ *then there is a processor* p *such that* $Program(p, i) = multicast_p(m, T)$.

While the first assumption relates to the syntax of a correct program the next two assumptions relate to the execution of a correct program.

Assumption 2 (Iteration) *Each processor* p *issues the call* $Program(p, 1)$ *and issues* $Program(p, i + 1)$ *after* $Program(p, i)$ *returns.*

Assumption 3 (Maximum Message Size) *There is a maximum message size and each finite buffer has room for at least one message.*

2.2 LAN-Communication Layer

The LAN communication layer (typically a UDP) has a broadcast capability. The calls made by the transport layer to the LAN-communication layer are *broadcast* of a message and *receive* of a message. Although some messages may be lost, we assume that if a message is broadcast infinitely many times it will be received infinitely many times by all other processors. Formally, we have the two following assumptions.

Assumption 4 (Eventual Receipt) *If the same message is broadcast infinitely many times from one processor then it will be received infinitely many times at every other processor.*

Assumption 5 (Sane Receipt) *If a message is received at a target processor then it was previously sent by the source processor claimed.*

2.3 A Correct Transport Layer and Its Properties

In this subsection we specify the notion of a correct transport layer and specify its main properties.

A transport layer is correct if any correct *Program* that runs over it with a set P of processors, without any processor failure, satisfies the following propositions.

Proposition 1 (Progress) $(\forall p \in P)$ $Program(p, i)$ *is eventually issued.*

Proposition 2 (Correct delivery) $(\forall p, q \in P)$ *If* $Program(p, i) = multicast_p(m, T)$ *and* $Program(q, i) = request_q(p, T)$, *then at* p *the multicast returns and at* q *the message* m *is delivered and the request returns.*

It is easy to see that a correct transport layer has the following properties:

Property 1 (Delivery by a Request) *Messages are delivered once to the user-communication layer only in response to requests. The request returns after the correct message is delivered.*

Property 2 (FIFO Delivery) *Messages from the same source processor are delivered to the target set in the order in which they were multicast.*

Note that throughout the paper we use the term "deliver" as delivering a message from the transport layer up to the user-communication layer in the same processor. In the next two sections, we will describe protocols that implement a correct transport layer assuming that Assumptions 4 and 5 on the LAN-communication layer are satisfied. The proof of correctness of the protocols is accomplished by proving that the foregoing two propositions are true, namely, that we get progress and correct delivery.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

3 A Simple Protocol (P0)

The P0 protocol is described in Figures 2 to 5. This protocol includes only what is essential for correctness. It assumes unbounded memory space for keeping a copy of the messages that have been sent out. Therefore it is obviously unrealistic and is also inefficient. It is presented here to provide a basic idea of the general protocol. The main idea in P0 is that we guarantee reliable delivery by using the fact that every *multicast* has a matching *request*. In particular, a NACK is generated infinitely many times (using a timer) upon a *request* which can not be satisfied, until the *request* is satisfied.

We make the following remarks for clarification of the protocol described in Figures 2 to 5.

- We present the protocol which will be executed at each processor. We assume each processor is preassigned some unique pid which is stored in the variable *myid*. Furthermore, at the initialization of the protocol each processor receives the pids of all other processors. We also use P_{myid} to denote the executing (current) processor.
- Each processor has a set of input buffers, one for each sender. Each input buffer can hold at least one message. This idea helps in providing the property of non-interference between messages from different sources. We note that this is an implementation choice and other solutions are possible.
- An important data structure is the Personal Counter Vector, denoted as *pcv*. It is an array of the size of the processor set. It reflects the highest consecutive personal counter that has been seen by P_{myid} from each processor. When the *PersonalCount* on an incoming message is not consecutive with the *pcv* for the sender, the message must be either too early (for instance, due to message losses) or too late (for instance, when this message is a resend, and the original one has already been received).
- The target set T is specified as part of every message m that is targeted to T. The *target* field of the message is overloaded for convenience of the presentation. It is sometimes referred to as a set, and other times as a single processor. The meaning should be clear from the context.
- To *broadcast* m means to send a message m over the broadcast medium of the communication network. Every processor on the network can then receive it.
- To *resend* a message is to broadcast it exactly as was done the first time it was sent out, with the same counter.
- When a message is received from the communication layer (*receive*(m)) it can be found in a place reserved for the incoming messages until the procedure *handle*(m) is called. When we want to specifically keep this message for further use, we clearly state that the message is being kept (e.g. added to a buffer).

4 A Practical Protocol (P1)

Due to space limitation, the pseudocode of the P1 protocol is omitted here and can be found in [6]. This protocol is different from P0 in that it does not assume unbounded memory space. Hence it is practical but can still be optimized. In P0 a sender kept a copy of every message it sent. The new ingredient in P1 is a mechanism to deal with the discarding of messages at the sender after it has been verified that all the processors in the target set handled them. The *status* mechanism provides a means by which to know when a sent message can safely be discarded.

5 The PCODE Protocol

The PCODE protocol is an expansion of P0 and P1. It includes many additional features which make it efficient, but do not change the basic properties of P0 and P1. The pseudocode of the PCODE protocol can be found in [6].

5.1 The Global Counter

In addition to the personal counters, each processor keeps a *GlobalCounter* which roughly counts the number of messages sent on the whole system. The processor adds this counter to every message it sends out. Each processor will increment its *GlobalCounter* whenever it sends out a message or delivers up one carrying a higher *GlobalCounter* than it already has. Notice that, unlike the *PersonalCounter*, the *GlobalCounter* is not necessarily unique for different messages since two processors broadcasting concurrently may use the same *GlobalCounter*. However, the *GlobalCounter* provides a method to control the flow of messages on the network as well as a method for possible early detection of lost messages as described below.

A message can be safely delivered if it is carrying a consecutive *PersonalCounter*, even if there is a gap in the *GlobalCounter* it is carrying, but the receiving process should not update its own *GlobalCounter* if there is such a gap. To avoid unnecessary delay in the delivery of a message, we deliver messages immediately, even if they have a gap in the *GlobalCounter*, but in that case we remember the counter for later updating. We keep a list of the counters which created such a gap and were attached to a message that has already been delivered. Whenever the *GlobalCounter* is updated we will check this list to see if any of its counters can now be updated. This is necessary in order to keep track of the *GlobalCounters* we received from each processor, for the purpose of flow control, as described in the next subsection.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

5.2 Flow Control

In addition to the *GlobalCounter*, each processor keeps a vector *gcv*, the size of the processor set. Here, *gcv*[*i*] holds the last *GlobalCounter* that processor P_{myid} has seen on a message from processor *i*. If the minimum on *gcv* is too far from the maximum (which is the current *GlobalCounter*) this means that too many messages have not yet been delivered, they might be lost, and there may be a lot of message traffic on the network. Therefore when the difference between the minimum and the maximum on *gcv* is greater than a FLOW_WINDOW size, P_{myid} will stop sending, until the difference decreases. FLOW_WINDOW is a tunable size (See [1] regarding flow windows). Note that the difference between the minimum and maximum will also be large when the user communication layer on a processor is not initiating the sending of messages. For this reason every processor will send out PROGRESS messages whenever it sees that the difference is too large, or when it is requested to do so by another processor. A PROGRESS message is simply a message that holds the current *GlobalCounter* of the sender. This will enable the other processors to update the sender's *gcv*. See Subsection 5.5 for the actual sending of the progress messages. Notice that the *GlobalCounter* on a PROGRESS message is not "original" in the sense that it was not necessarily incremented to its current size by the sending process (which is the case for a *GlobalCounter* on a REGULAR message). Therefore a process receiving a PROGRESS message may not increment its own *GlobalCounter* according to the one on the message. The *GlobalCounter* should be incremented only when a message is received from the sender which originated the counter. The PROGRESS messages may be used only to increment the *gcv* of their sender. Therefore when we update the *GlobalCounter* and when we save counters (as described in Subsection 5.1 above) we distinguish between "original" and "non-original" counters.

5.3 Early Detection of Message Loss

In P0 and P1 we know that a message is missing only if it is requested by the user-communication layer. When messages are actually lost by the UDP layer, we can often identify the loss even before the message is requested. If we receive a message with *PersonalCount* not consecutive to *pcv* of the sender, we have probably lost some message(s) (though they may still arrive later). In this case we can send a NACK to the sender, requesting the missing message(s), thus improving our chances of having the message ready when it is requested. When using the *GlobalCounter* described in the previous subsection, we can also check for gaps in this counter, which indicate possible message loss in the same way as the gaps in the *PersonalCount*. We cannot identify the sender of the messages that we lost, but if we broadcast a NACK the sender will be able to identify its own messages and resend them. A benefit of detecting message loss by the *GlobalCounter* is that even if the sender has stopped sending new messages, as long as some other processor has seen the lost message and sent a new one after that, we may see a gap in the *GlobalCounter*. Since the *GlobalCounter* is not precise, it may not help detect a loss when more than one message was given the same *GlobalCounter*. The *PersonalCount* will detect every loss, as long as we receive further messages from the same sender.

5.4 Saving Early Messages

In P0 and P1, if a message is too early (i.e., there is a gap in the *PersonalCount*) we ignore the message. In PCODE we maintain a buffer of waiting messages, in which we keep messages which have arrived too early, until they can be accepted. Whenever we accept a message we can check this buffer to see if the next message we expect is already there. When the buffer is full - we will ignore the "too early" message as in P0 and P1. This mechanism can reduce the number of messages that have to be resent when a message loss occurs.

5.5 Periodic Status Messages

In P1, messages will be discarded from the buffer of sent messages only when the processor has a message to multicast and has found the buffer full. It would obviously be better to try and discard messages before the buffer is full, so as not to slow down the user's application. Therefore each processor should send STATUS messages periodically (the same STATUS messages used in P1). A processor can determine when to send a STATUS message by looking at the *GlobalCounter*, and remembering the *GlobalCounter* that was sent on the last STATUS message. If the *GlobalCounter* has grown more than STATUS_WINDOW (a tunable size) since the last STATUS message was sent, the processor will send out a new one. The STATUS messages can also be used as PROGRESS messages, which are described in Subsection 5.2 above. Since both STATUS and PROGRESS messages are usually useful to the protocol, we combine the two. The messages of type STATUS will in fact include the PROGRESS information as well (which is simply the *GlobalCounter*). When handling these messages both issues will be taken care of. Whether we are required to send STATUS or PROGRESS information, we will always send the "augmented" STATUS message.

5.6 Sending Point-to-Point Messages

Though our goal is to make use of the broadcast medium, in some cases the message is intended only for a small number of processors, or even one recipient, namely, a point-to-point message. In this case it would be undesirable to broadcast the message, thus forcing all the processors on the network to read it and process it. This occurs either when the user-communication layer specifies a target group of size one, or for certain control messages—e.g., a NACK message indicating message loss from a known sender. In these cases we can send the messages by UDP, using the specific host's address instead of the broadcast address. The recovery of the point-to-point messages

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

cannot be done by the mechanism used for the recovery of broadcast messages, since this mechanism relies on the fact that all processors can receive all the messages. Therefore a separate mechanism must be supported to deal with recovery of point-to-point messages. Such a mechanism is simple to construct.

5.7 Timeouts

In P0 and P1 we had only one type of NACK, which is issued when a *request* is issued from the user-communication layer, if the requested message is not ready. In PCODE we have two more types of NACKs, which are issued when gaps are found in the PERSONAL or in the GLOBAL counters. P0 through P1 set a periodic timer for resending a NACK only in the case of a REQUEST NACK, which is the only case in which the protocol may deadlock if the NACK is not resent. In PCODE we set a timer for all types of NACKs. Every NACK will be periodically resent until it has been satisfied with the required messages. In the case of an unsatisfied *request*, PCODE does not issue a NACK immediately, but waits for an initial timeout in order to give the message a chance to arrive. If the message does not arrive within this timeout, a REQUEST NACK is issued, and a periodic timer is set. The length of the timeouts for the different NACKs may be tuned, as described in Section 6.5.

6 Implementation and Performance Evaluation

In this section we will present the implementation effort of PCODE and the environment that we have set up for performance evaluation. We will also present the results of our measurements, which clearly express the advantage of our approach.

6.1 The Environment

We have implemented a prototype of the PCODE protocol in C. The prototype was initially developed on a collection of RS/6000 workstations using the AIX operating system and communicating via UDP over a 10Mbit Ethernet LAN. The results in this paper were obtained on a collection of Silicon Graphics Indigo machines with R4000 processors, using the IRIX operating system and communicating via UDP over a 10Mbit Ethernet LAN.

The transport layer runs as a background daemon. This enables PCODE to treat the messages coming in from the LAN-communication layer while the user-communication layer is blocked, e.g., waiting for a *request* call to return. Therefore the PCODE protocol and the user-communication layer are implemented as two separate processes. The communication between them is done using TCP sockets. Ideally the two layers would be integrated into one multi-thread process, thus eliminating the time used for Inter-Process Communication (IPC).

6.2 The User-Communication Layer

In our initial experiments we have assumed that the global program (the user-communication layer) is performing an all-to-all broadcast in which each processor broadcasts a message to all other processors. For this we used two different drivers. One written in RAPID [9], the other is a simple C program for the user-communication layer which runs through the sequence of *multicast/request* that corresponds to an all-to-all broadcast. The driver runs through this sequence a large number of times and measures the average time it takes. We have observed certain variability in the times measured between individual communication events. As a result, we have developed techniques for obtaining an average time per call as a figure of merit for our protocol.

We tried implementing the all-to-all broadcast in two ways. In one implementation each processor broadcasts its message in turn. While one processor calls *multicast*, all the other processors call the corresponding *request*. In the second implementation each processor first calls *multicast*, and then calls a series of *requests*, one for each other processor. Our tests showed that the time for an all-to-all broadcast using PCODE is better when using the second implementation. When using TCP, On the other hand, it is better to use the first implementation. The results in the following section were obtained using for each system the implementation that gives better results.

In our discussions hereafter, we will refer to the "time per call". This time is obtained by dividing the average time measured for the all-to-all broadcast by the number of machines in the configuration. The term "time per call" is not accurate, since it is in fact an average of the time for one multicast and the time for $N-1$ requests, where N is the number of machines. This normalization enables us to compare the performance over a changing number of machines.

6.3 Optimizing TCP

In order to optimize broadcast time, protocols using TCP must usually be tailor made, considering the number of processes participating in the broadcast, and which process should receive which information. This is true for all point-to-point communication, and specifically for TCP, which performs differently for different patterns of communication on the connection.

In order to compare PCODE to TCP, we implemented a TCP program which implements the *multicast* and *request* calls, as defined in this paper, simply by using TCP point to point connections, which are reliable. Unlike PCODE, the TCP program does not run separately from the driver. It is linked with the driver and run as one process. This fact gives TCP the advantage that it does not need IPC communication. Since we were using the same atomic calls as in PCODE, namely *multicast* and *request*, we could not fully optimize TCP. E.g., we could not parallelize the multicasts - a multicast must be completed before the next one can begin. The TCP *multicast* was implemented as a series of sends, one to each target processor.

6.4 Results

We tested the protocol on up to 16 machines. The machines were not dedicated to the tests, but the load

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

apart from the tests themselves was not high. The messages were of sizes of up to 1Kbyte, since at the PCODE level we are interested only in sending UDP packets.

To obtain a measure of "ms per call" for a certain configuration and message size, we ran a number of tests, each one of 1000 rounds of all-to-all broadcast. For each test we obtained the average time per round, and divided it by the number of machines. We then took an average over the results on each of the machines for each of the tests and this final average is the "ms per call" for this configuration and message size. The variance of the results is usually under 10%.

In the next 2 subsections we compare PCODE to TCP and to distributed transport layers.

6.4.1 Comparing to TCP

In Figures 6 and 7 we show the time per call plotted against the number of machines in the configuration with message size 20 bytes and 1 Kbyte, respectively. The figures compare the PCODE curve to the TCP curve. With message size 20 PCODE is not faster than TCP on up to 8 machines. With larger configurations and larger message sizes PCODE is always faster, up to an order of magnitude faster with 16 machines and message size 1K. It is clear that while the TCP time grows linearly with the number of machines, the PCODE time hardly grows at all.

Figures 8 and 9 show the time per call against the message size, for a configuration of 3 and 16 machines, respectively. Each plot compares the PCODE time with the TCP time for the same configuration. On 3 machines we see again that PCODE is not faster than TCP when the message size is less than 1K. Nevertheless it is evident that the PCODE curve almost stops growing towards the 1K message size, while the TCP curve is growing steadily. On 16 machines the TCP time grows very fast with the message size. This shows that the performance of PCODE scales much better compared with the solution based on TCP.

6.4.2 Comparing to Distributed Broadcast Layers

In comparing our results to those previously published for distributed transport layers, like Transis, one has to notice that parallel protocols have a built-in synchronization which influences the performance. For example, each all-to-all requires all machines to synchronize. Moreover, in a typical distributed broadcast layer a slow machine hardly influences the throughput measured, whereas in a synchronous mode it slows down every other machine.

In Transis the reported measurements are for maximum flooding of the network, and do not measure latency. In Horus [17] the results refer to packing several short messages on a single UDP packet. We tried to bring the measurements to a common ground, for that we performed a few experiments in which we imitated the transmission patterns of MPI [14] over Transis and Horus.

Our experiments show that PCODE's performance is comparable to that of the other distributed broadcast layers. In Figures 10 and 11 we show the results of running repeated all-to-all broadcast calls in an MPI mode on different systems, with a message size of 20 bytes and 1 Kbytes, respectively, over a changing number of machines. The all-to-all broadcast is implemented in the second version (see previous section). Note that the PCODE timings in these two figures were measured in different runs from those presented in Figures 6 through 9.

We compared to Transis running over Lansis as well as Transis running over the Token Ring protocol for message recovery and ordering. Note that all but PCODE are protocols which have been tuned and optimized over some period of time now, while PCODE is a newly developed protocol. It is evident that PCODE performs better than Transis using the Ring, but the same as Transis using Lansis. Horus performs better than all the tested systems. We note here that Horus is implemented as one multi-threaded process, as ideally we would like to implement PCODE. We believe that with such an implementation and with some further tuning PCODE should eventually perform better than any general distributed broadcast layer, since its requirements are more lenient.

6.5 Tuning the Constants

In the previous 3 sections describing the protocols, we mentioned that several parameters of the algorithm are tunable. As an example of what can be accomplished by such tuning, we experimented with the size of one of the timeout delays. The specific delay was the length of time to wait between the arrival of a *request* for a message not yet received and the sending of a NACK to the source. The longer the delay, the longer it would take to deliver a message that was actually lost; however, the shorter the delay, the more likely that the NACK and its response would be wasted because the required message was actually in transit. In our experiment, as we raised the delay, we observed a significant increase in the number of NACKs sent and a slight rise in the overall time per call. The best timing obviously depends on the reliability of the network as well as the speed of the machines, but it is clear that a real improvement in time can be achieved by appropriately tuning the constants.

7 Concluding Remarks

We have studied the requirements associated with collective communication for parallel computing. We have observed that the main difference between a distributed computing paradigm and a message passing parallel computing paradigm is that, in a distributed environment the activity of every processor is independent while in a parallel environment the collection of the user-communication layers in the processors can be modeled as *a single global program.* We have formalized the requirements by defining the notion of a

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

correct global program. This notion provides a precise specification of the interface between the transport layer and the user-communication layer. We have developed PCODE, a new communication protocol that is driven by a *global program*, and proved its correctness. We have implemented the PCODE protocol and run it over a collection of up to 16 IBM RS/6000 workstations, using the AIX operating system as well as over Silicon Graphics Indigo machines with R4000 processors, using the IRIX operating system. In both cases the workstations were communicating via UDP over a 10Mbit Ethernet LAN. The experimental results indicate that an improvement in performance of roughly an order of magnitude (in the case of 16 workstations) can be obtained using our approach compared to current approaches. Initial results also show that PCODE's performance is comparable to other distributed broadcast layers.

We note here that PCODE is just one possible implementation of the transport layer as formally defined. Recently, we have developed another new protocol, called User-level Reliable Transport Protocol (URTP), for this purpose [7]. The URTP protocol, which extends the AIX kernel, runs on LAN of IBM RS/6000 workstations. Note that the ideas presented in this paper can be easily extended to any Network of Workstations that provides an unreliable broadcast transport protocol (e.g. ATM).

Acknowledgements

We would like to thank especially Dalia Malki for her invaluable advice, coding ideas and trouble shooting. Thanks to Yair Amir for his coding ideas and advice on IPC and to Jim Wiley for useful help and advice on AIX.

References

[1] Y. Amir, D. Dolev, S. Kramer and D. Malki, "Transis: A communication subs-system for high availability," *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, IEEE, pp. 76–84, 1992.

[2] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal and P. Ciarfella, "Fast message ordering and membership using a logical token-passing ring", *Proceedings of the 13th International Conference on Distributed Computing Systems*, pp. 551–560, May 1993

[3] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C.T. Ho, G. Irwin, S. Kipnis, R. Lawrence and M. Snir, "The IBM External User Interface for Scalable Parallel Systems", Parallel Computing, Vol. 20, No. 4, pp. 445–462, April 1994.

[4] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.T. Ho, S. Kipnis, and M. Snir, "CCL: A portable and tunable collective communication library for scalable parallel computers", International Parallel Processing Symposium, pp. 835–844, Cancun, Mexico, April 1994. To appear in IEEE Trans. on Parallel and Distributed Computing, February 1995.

[5] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood, *The ISIS System Manual*, Dept. of Computer Science, Cornell University, September 1990.

[6] J. Bruck, D. Dolev, C.T. Ho, R. Orni and R. Strong, *PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcast Domains*, IBM Research Report RJ 9895, September 1994.

[7] J. Bruck, D. Dolev, C.T. Ho, M. Rosu and R. Strong, *Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations*, IBM Research Report RJ 9925, December 1994.

[8] D. R. Cheriton and W. Zwaenepoel, "Distributed process groups in the V kernel," *ACM Trans. on Computer Systems*, 2(3), pp. 77-107, May 1985.

[9] D. Dolev, R. Strong, and E. Wimmers, "Experience with RAPID prototypes", *Proceedings of the IEEE International Workshop on Rapid Systems Prototyping*, Grenoble, June 1994.

[10] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[11] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, "A user's guide to PICL: a Portable Instrumented Communication Library", *ORNL Technical Report*, ORNL/TM-11616, October 1990.

[12] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems", *Chapter 5 in Distributed Systems, second edition*, Edited by S. Mullender, ACM Press New York, 1993.

[13] P. M. Melliar-Smith, L. E. Moser and V. Agrawala, "Broadcast protocols for distributed systems", *IEEE Trans. on Parallel and Distributed Systems*, January 1990.

[14] Message Passing Interface Forum, *Document for a Standard Message-Passing Interface*, University of Tennessee, Technical Report No. CS-93-214, November, 1993.

[15] J. F. Palmer, "The NCUBE family of parallel supercomputers", *Proceedings of the International Conference on Computer Design*, IEEE, 1986.

[16] L. L. Peterson, N. C. Bucholtz and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Trans. on Computer Systems*, 7 (3), pp. 217–246. 1989.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

[17] R. van Renesse, K. P. Birman, R. Cooper, B. Glade, and P. Stephenson, "Reliable Multicast between Microkernels", *Proceedings of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pp. 27–28, April 1992.

[18] A. Skjellum and A. P. Leung, "Zipcode: a portable multicomputer communication library atop the reactive kernel", *Proceedings of the 5th Distributed Memory Computing Conference*, IEEE, pp. 328–337, April 1990.

[19] A. S. Tanenbaum, M. F. Kaashoek and H. E. Bal, "Parallel programming using shared objects and broadcasting," *IEEE Computer*, vol. 25, 1992.

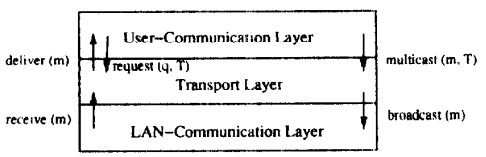

Figure 1: The three logical layers of software in a processor.

```
do forever {
  if there is a multicast (m, T) issued
                    from the layer above then
    communicate (m, T);
  if there is a request (q, T) for a message
                          to be delivered then
    if there is a ready message m from source q then
      deliver (m);
    else {
      IssueNack (q);
      denote there is a pending request (q, T); }
  if there is an incoming receive (m) then {
    handle (m);
    if there is a pending request
                    (m.sender, m.target) then
      if m is in the ready slot
                    from source m.sender then {
        deliver (m);
        inactivate the nack timer (if it was set);
        denote that there is no pending request; } }
  if the nack timer has expired then
    if there is a pending request (q, T) then
      IssueNack (q); }
```

Figure 2: P0: The main control loop.

```
handle (m) {
  q = m.sender;
  T = m.target;
  if (q == myid) then
    return;
  case (m.type) of
  REGULAR msg:
    if the ready slot for q is not free then
      return;
    if (m.PersonalCount ≠ pcv[q] + 1) then
      return;
    pcv[q] = m.PersonalCount;
    if (myid is a member of T) then
      put m into ready slot for q;
  NACK msg:
    if (T == myid) then
      resend m' from buffer of sent messages for
        which m'.PersonalCount > m.LastRcvd;
  return; }
```

Figure 3: P0: Procedure handle.

```
communicate (m, T) {
  increment pcv[myid];
  m.PersonalCount = pcv[myid];
  m.sender = myid;
  m.type = REGULAR msg;
  m.target = T;
  broadcast m;
  Keep m in buffer of sent messages; }
```

Figure 4: P0: Procedure communicate.

```
IssueNack (q) {
  m.sender = myid;
  m.type = NACK msg;
  m.target = q;
  m.LastRcvd = pcv[q];
  broadcast m;
  set the nack timer; }
```

Figure 5: P0: Procedure IssueNack.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE

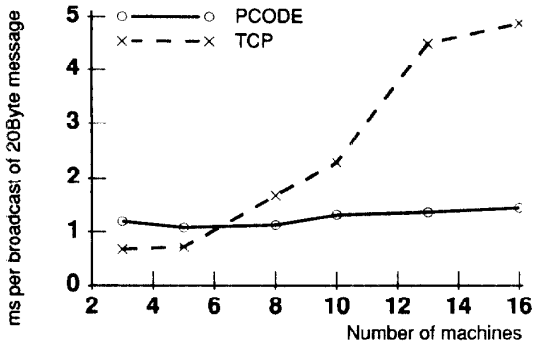

Figure 6: The time per broadcast of 20 byte message as a function of the number of machines.

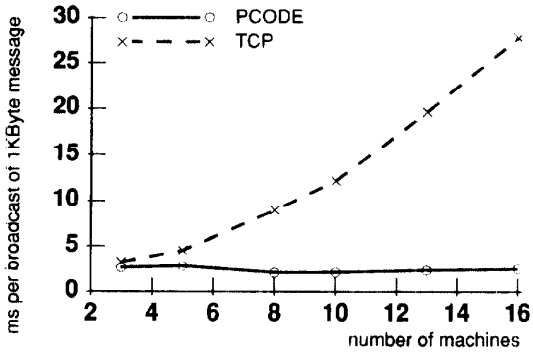

Figure 7: The time per broadcast of 1 Kbyte message as a function of the number of machines.

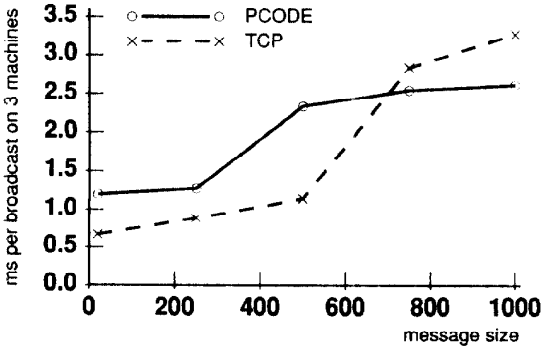

Figure 8: The time per broadcast call on 3 machines as a function of message sizes.

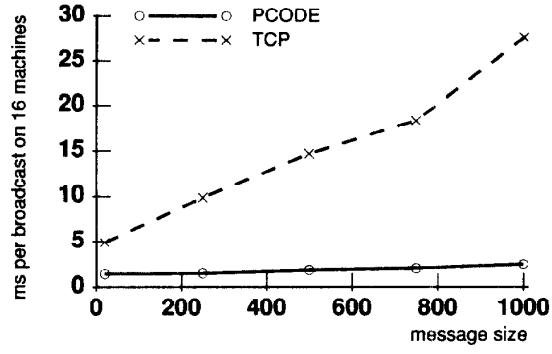

Figure 9: The time per broadcast call on 16 machines as a function of message sizes.

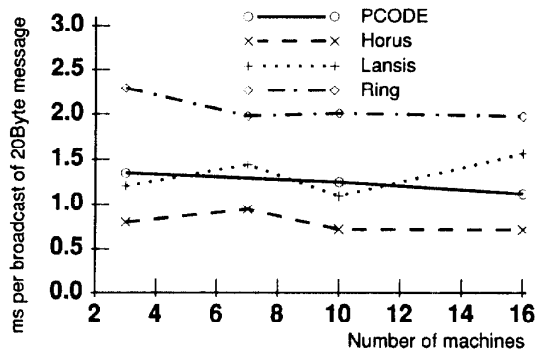

Figure 10: Comparison of the time per broadcast on PCODE and related protocols for 20 byte messages.

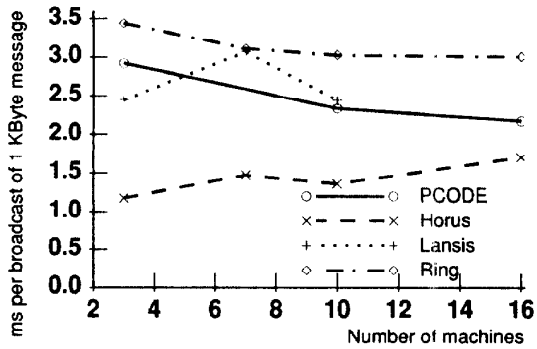

Figure 11: Comparison of the time per broadcast on PCODE and related protocols for 1 Kbyte messages.

Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)
1063-7133/95 $10.00 © 1995 IEEE