December 1980

Report No. STAN-CS-80-850

Also numbered:
CSL TR-207

Performing Remote Operations Efficiently on a Local Computer Network

by

Alfred Z. Spector

Department of Computer Science

Stanford University
Stanford. CA 94305

Performing Remote Operations Efficiently on a Local Computer Network .

Alfred Z. Spector
Department of Computer Science
a n d
Computer Systems Laboratory
Stanford University

Abstract

This paper presents a communication model for local networks, whereby processes execute generalized remote references that cause operations to be performed by remote processes. This remote reference/remote operation model provides a taxonomy of primitives that (1) are naturally useful in many applications and (2) can be efficiently implemented. The motivation for this work is our desire to develop systems architectures for local network based multiprocessors that support distributed applications requiring frequent interprocessor communication.

After a section containing a brief overview, Section 2 of this paper discusses the remote reference/remote operation model. In it, we derive a set of remote reference types that can be supported by a communication system carefully integrated with the local network interface. The third section exemplifies a communication system that provides one remote reference type. These references (i.e., remote load, store, compare-and-swap, enqueue, and dequeue) take about 150 microseconds, or 50 average instruction times, to perform on Xerox Alto computers connected by a 2.94 megabit Ethernet. The last section summarizes this work and proposes a complete implementation resulting in a highly efficient communication system.

Keywords: communications, distributed processing, local networks, multiprocessing, reliability.

This work was supported by a Fannie and John Hertz Foundation Graduate Fellowship. Some of this work came about as a result of the author's work at IBM's San Jose Research Laboratory. Firmware and software were developed on equipment given to Stanford University by the Xerox Corporation.

Table of Contents

1. Overview

This paper discusses an approach to communications that is capable of supporting distributed applications requiring *frequent* interprocessor communication on very high speed local networks. We believe it is important to study ways whereby the traditionally high software overhead of interprocessor communication [Peterson] can be reduced, particularly given recent hardware developments that make very high speed local networks feasible [Blauman] [Rawson-Metcalfe] [Ikeda et al]. We feel that if this goal can be achieved, some distributed applications exhibiting a fine granularity of parallelism [Jones-Schwarz] can be profitably executed on a local network based computer system. The approach to communication on local networks described here differs from recent internetworking research and relates as much to multiprocessors such as CM* [Swan *et al*] as it is does to usual local networking systems. See [Spector1980] and [Andler et *al*] for a more complete discussion of this approach.

Section 2 describes a model that is intended to serve as a basis for an efficient and flexible communication system for local networks. In this model, a process can execute a generalized *remote reference* that causes a *remote operation* to be performed. The remote operation may return a value to the caller.

Though remote reference semantics can be achieved with traditional high level communications protocols, careful study of the remote reference/remote operation model is useful for the following reasons:

(1) The execution of remote references that cause remote operations to be performed is naturally useful. Examples are remote memory references, remote subroutine calls [Nelson], and message passing operations. Specific examples include the SIGP operation on the IBM 370 [IBM] and "requests" in the Tandem operating system [Bartlett].

(2) This model suggests a general interface that includes types of remote references having differing levels of function, for example, with respect to reliability, flow control, or synchrony.

(3) A local network interface that provides useful primitives (such as remote references) can support more efficient communication than an interface that provides only raw message passing facilities. A direct implementation in microcode or hardware of the various reference types will be more efficient than an implementation that requires layering software on more primitive facilities.

(4) Many types of remote references can be implemented using simpler protocols than are required for more general and common communications mechanisms such as byte stream primitives. Using implementations tailored to the various reference types also contributes to efficiency.

In Section 3 we attempt to add substance to the discussion of the remote reference/remote operation model by describing an experiment in which a simple type of references was microcoded on Xerox Alto computers using the 2.94 megabit experimental Ethernet [Metcalfe-Boggs]. We show that these references execute quickly: they typically take about 150 microseconds on this hardware. This time is about two orders of magnitude faster than could be expected if they were implemented in a conventional way.

Section 4 summarizes the points made in Sections 2 and 3. Additionally, it suggests the possibility of implementing a full communication system for local networks based on the remote operation/remote reference model.

2. The Remote Reference/Remote Operation Model

Sections 2.1 and 2.2 contain definitions, assumptions, and a description of the proposed communications model. Because the concept of sessions (i.e., connections between processes) play such an important role in our model, they are introduced in Section 2.3, ahead of all other implementation concepts. Section 2.4 contains a taxonomy of reference types and is the focal point of this section. The protocol issues arising from this model are then described in Section 2.5. An example of one reference type is presented in Section 2.6. Finally, Section 2.7 discusses the benefits that will be realized if this model is used as the basis of a communication system.

2.1 Definitions and Assumptions

The remote reference/remote operation model provides a basis for communication on a system comprised of medium size *processors* connected by a very high speed local network, as shown in Figure 1. To discuss the model fully, we must first specify some aspects of the underlying system architecture.

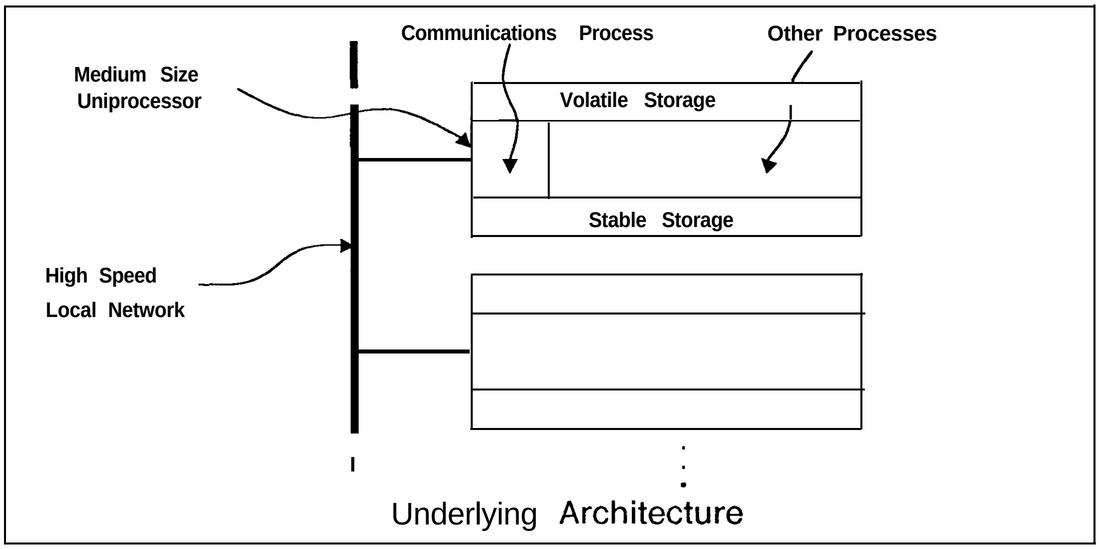


Figure 1

Data is exchanged among machines by the transmission of *packets* over a local network- a high bandwidth, low latency communication medium having high reliability and low cost. With respect to reliability, the local network provides four transmission properties: First, if a packet is transmitted enough times, it will reach its destination. Second, packets corrupted during transmission are automatically discarded. Third, packets are not duplicated by the network. Finally, packets arrive in the order in which they are sent. Thus, local network failures result only in lost packets.

Certain types of the remote references described below are practical only if transmission latencies are very small for short packets. In addition, a network with a high capacity is needed to support a distributed system comprised of many (say, 100) processors. A 50 to 100 megabit local network enforcing a small maximum packet length meets these needs.

In the model, processors occasionally' fail and are then restarted. (In [Spector1981], this work is extended to allow for failure of individual processes on a processor.) In addition, processors contain *processes* with names unique to that processor. All *regular processes* disappear after a processor failure. *Recoverable processes* are automatically reincarnated after a processor crash and are reset to one of several predetermined states. Clearly, recoverable processes require the availability of *stable storage*– storage that survives single failures [Lampson-Sturgis]. Processors are sufficiently reliable to allow recoverable processes to make progress.

We assume the existence of a distinguished process on each processor, called the *communications process*. This process is recoverable and is specially implemented-off-loaded or at interrupt level- so that it can be activated very rapidly. Though its implementation must permit the execution of simple functions quickly, it must also be able to do more complex operations requiring stable storage. (The need for this is described in Section 2.3.) The communications process does not have the full capabilities of other processes, but it can perform the following functions efficiently:

(1) Transmission and reception of packets on the local network. The communications process directly accesses the network-specific hardware.

(2) Manipulation of specialized communication state information. The communications process can quickly access the state tables described in Section 2.3. Some table entries may be saved in stable storage.

(3) The execution of communications primitives issued by other processes on the same processor. This process implements the communications interface seen by other processes.

(4) The execution of simple operations inititated by a remote communications process. Simple requests can be directed to a remote communications process for efficient processing.

2.2 **Model** Description

In our terminology, a *remote reference* is executed by a *master* process and causes a r-emote *operation* to be performed by a *slave* process. The slave process can return a value. Remote operations can vary greatly in complexity, from the simplicity of a memory access to the complexity of an asynchronous subroutine. For example, the message passing "send" operation causes a data block to be placed on the receiver's message queue and provides an indication of the success of that operation. In this case, the sender is the master, and the server (at the remote site) is the slave.

Figure 2 illustrates the remote reference/remote operation model in the absence of communications or processor failures. A reference is initiated by an action labeled Reference-Commit. At a later time, a request is received on the slave processor, and the communications process initiates the appropriate remote operation by issuing an action called Op-Begin-Commit. The remote operation is considered complete only when the action labeled Op-End-Commit has been executed. If the remote operation produces a value, the communications process on the slave issues a *response* to the master, which causes the action labeled Result-Commit to be executed. To allow for some reliability semantics, some of the four actions in Helvetica tvnc must be atomic and must commit some state information to stable storage.

In this model, short requests and responses are transmitted as single packets. Requests and responses that do not fit in a single packet are transmitted as *multipackets*, or sequences of packets. Individual packets in a multipacket are neither retransmitted nor acknowledged. Only the simplest underlying protocol is used to permit a receiver to determine if all packets in a multipacket are properly received; e.g., a packet count and multipacket identifier. The properties of the local network described in the previous section permit the use of this simple strategy for handling multipacket requests and responses.

There arc many posciblc protocols for implementing remote references. In the simplest case, upon executing a remote reference, a master issues a request packet to a slave and possibly awaits a subsequent response packet from that slave. In other cases, the protocol is much more complex. See Section 2.5 for a more detailed discussion of this.

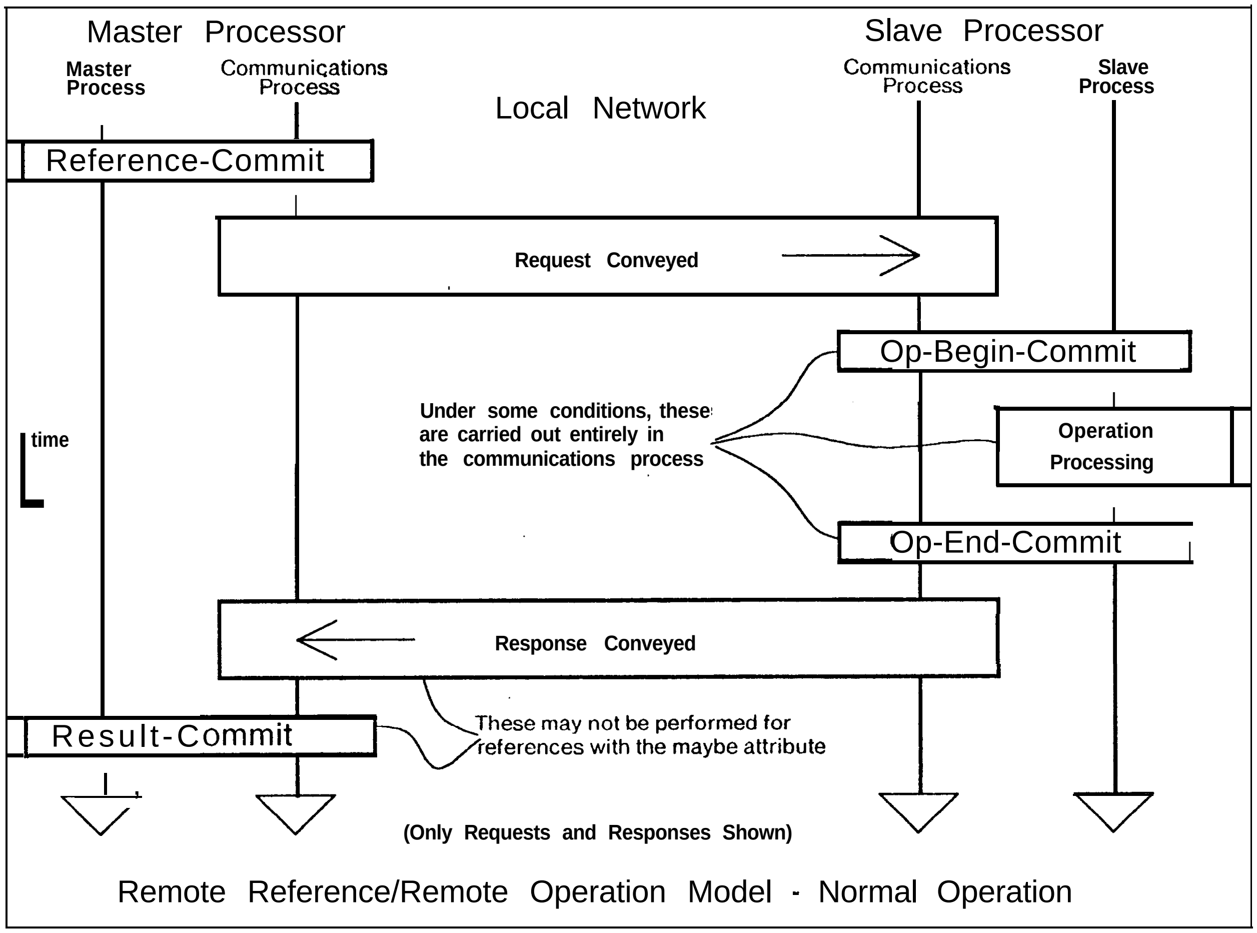

Master Processor
Slave Processor
Master Process
Communications Process
Local Network
Communications Process
Slave Process
Reference-Commit
Request Conveyed
Op-Begin-Commit
Under some conditions, these are carried out entirely in the communications process
time
Operation Processing
Op-End-Commit
Response Conveyed
Result-Commit
These may not be performed for references with the maybe attribute
(Only Requests and Responses Shown)
Remote Reference/Remote Operation Model - Normal Operation

Figure 2

2.3 Sessions and Sockets

As considered in this paper, a *session* is a logical connection over which a single master process can issue requests and a single slave process can issue responses. Processes are assumed to be located on fixed processor nodes for the life of the session. Sessions do not have multiple masters or slaves in this work.

When a session is established, one *socket* entry is created on each of the master and slave processors. These entries serve three main functions: First, packets associated with that session are addressed using references to these entries. Second, socket entries contain information that permits requests and responses to be mapped to individual processes. Third and most importantly, they contain state information that enable sessions to provide specific semantic attributes for references conveyed during that session. For some session types, considerable state information (for such purposes as flow control or duplicate detection) must be maintained. This information is used to support the various reference types described in Section 2.4. Sockets for active sessions are contained in a socket table that is accessed by the communications process and are implemented using both volatile and stable storage.

There are two types of sockets: type-1 sockets, which do not survive processor crashes and type-2 sockets, which use stable storage and do survive crashes. Type-2 sockets can be used by recoverable processes to ensure that their communications capabilities are not lost after processor failures. After a failed processor has been restarted, a recoverable process associated with type-2 sockets can continue executing references and/or receiving requests. No type-1 sockets are defined after a processor restart.

A session can be considered to be a distributed abstract data type that is manipulated by two cooperating communications processes via the two sockets. There are three major operations allowed during sessions.

> Issue-Reference permits a master process to initiate a remote operation on the slave and causes a Reference-Commit to occur locally.
>
> Receive-Response permits a master process to receive a response from a slave and causes a Result-Commit. Sometimes, Receive-Response is issued in combination with Issue-Reference.
>
> Return-Response allows a slave to return a result to its master and causes an Op-End-Commit to be locally executed.

Issue-Reference, Receive-Response, and Return-Response are generic names for primitives whose implementations are application dependent; in fact, their call syntax will usually be tailored to reduce overhead. For example, a normal memory reference may result in an Issue-Reference if a segmentation table specifies that the memory reference should be issued over a session.

Two sessions are maintained between each pair of communications processes to permit each communications process to act as both a master and slave to each other communications process. These sessions, called *systems-sessions*, allow other sessions between non-communications processes to be created and destroyed. Communications processes provide the following primitives:

> Register-Process-Name is processed locally and registers a server process name with the local communications process, and it specifies how requests for the server process will be conveyed to that process; e.g., via interrupt or queuing. It also specifies the type of session in which the slave will participate. This permits the communications process to respond to requests asking for sessions with this slave.
>
> Initiate-Session establishes a session with a slave process that has previously been registered. It requires as arguments the remote slave process name and address, the desired type of session (see Section 2.4), and an indication of the disposition for responses received from the slave. It returns a session number. Initiate-Session initiates a remote reference to the communications process on the slave. This remote reference serves the purpose of PUP's rendezvous protocol [Boggs *et* al]. Additionally, it allows the presetting of defaults for that session.
>
> Terminate-Session eliminates a session. It requires the session number as an argument and initiates a remote reference to the remote communications process. It can be executed by either a master or slave.

Figure 3 illustrates the initiatiation and termination of a session.

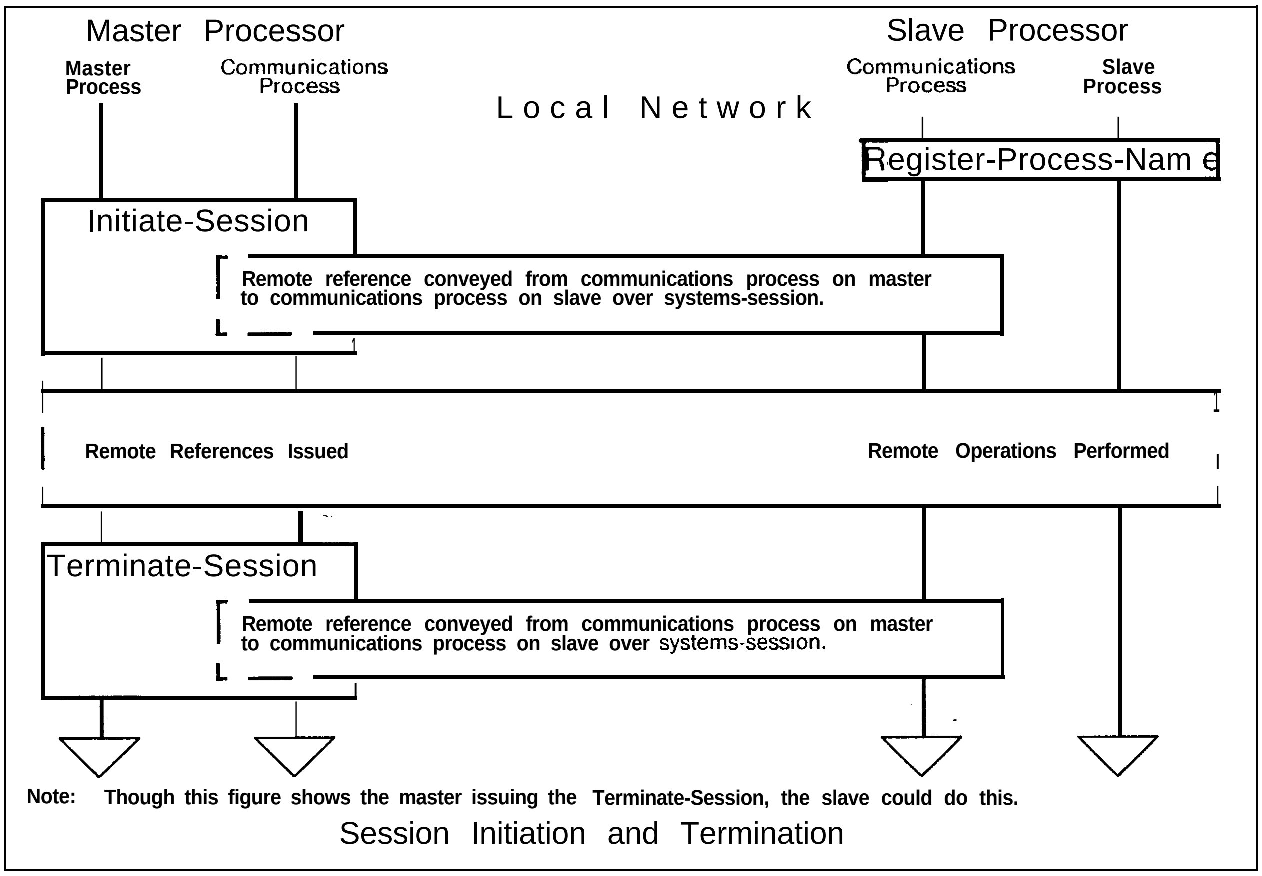

Master Processor
Slave Processor
Master Process
Communications Process
Local Network
Communications Process
Slave Process
Register-Process-Name
Initiate-Session
Remote reference conveyed from communications process on master to communications process on slave over systems-session.
Remote References Issued
Remote Operations Performed
Terminate-Session
Remote reference conveyed from communications process on master to communications process on slave over systems-session.
Note: Though this figure shows the master issuing the Terminate-Session, the slave could do this.
Session Initiation and Termination

Figure 3

In summary, the communications process performs a variety of functions: (1) It maintains systems-sessions, (2) It supervises the initiation and closing of sessions, (3) It accepts references from a master process, initiates their remote execution and possibly performs certain actions to inform the master of the result, and (4) It accepts remote requests from the network, awakens the slave process, if necessary, and possibly sends responses to the master.

The concepts of sessions and sockets are not unique to this work. For example, sockets are called half-sessions in SNA [Cypser] and TCB's in TCP [ISI], and sessions are implemented in Level 5 in the OSI protocol hierarchy [Folts]. However, in this work, there are many types of sessions, each fulfilling particular needs; the diversity of session types, in many instances, permits simpler protocols to be used. Additionally, the sessions in this model subsume the function of a few layers in general network hierarchies; this reduces the need for inefficient protocol layering.

2.4 Reference Taxonomy

In Sections 2.4.1 through 2.4.5, five different attributes that apply to sessions are presented. These attributes are associated with a session at the time the session is established and affect all references conveyed during that session. They are based on the following: (1) varying reliability requirements, (2) whether a value is returned, (3) how the remote operation occurs temporally with respect to the reference, (4) the need for flow control, and (5) for what kind of process the reference is intended. These attributes were selected to span the space of possible implementation strategies– and costs- as well as to provide a rich set of primitives with which to communicate.

2.4.1 Reliability

Careful specification of the performance of remote references under different failure conditions is important in distributed systems, because we often desire the system to tolerate failures. The attributes discussed in this section provide for various degrees of robustness under conditions of communications and processor failures. They arc summarized in Table 1 and described below. See [Liskov] for another characterization of reliability in high level communication systems.

Protocol Class	Reference Semantics Under Different Failure Conditions			
	No Failure	Lost Packets	Lost Packets & Slave Failure	Lost packets & Master Failure
Maybe	op performed: 1 result-commit: 1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0
At-Least-Once	op performed: 1 result-commit: 1	op performed: > = 1 result-commit: > = 1	op performed: > = 0 result-commit: > = 0	op performed: > = 0 result-commit: 0
Only-Once-Type- 1	op performed: 1 result-commit: 1	op performed: 1 result-commit: 1	op performed: 0,1 result-commit: 0,1	op performed: 0,1 result-commit: 0
Only-Once-Type-2	op performed: 1 result-commit: 1	op performed: 1 result-commit: 1	op performed: 1 result-commit: 1	normal master process op performed: 1 result-commit: 0 - - - - recoverable master process op performed: 1 result-commit: 1

Table 1 - Reliability Semantics Survey

Listed in increasing order of function complexity and implementation cost, the four reliability attributes are: maybe, at-least-once, and only-once-type-l and only-once-type-2. Their names arise from the semantics that these attributes provide under conditions of communication failures.

In the absence of communication or processor failures, all references initiate one Op-End-Commit and, if required, one Result-Commit. The performance of references under communication failure conditions (i.e., lost packets) is summarized below:

A reference having the maybe attribute will cause an Op-End-Commit to be performed zero or one times. If the Op-End-Commit is performed, the Result-Commit will occur zero or one times.

A reference having the at-least-once attribute will cause an Op-End-Commit to be performed one or more times. The Result-Commit will also occur one or more times.

A reference having the only-once-type-l or only-once-type-2 attribute causes exactly one Op-End-Commit and one Result-Commit to occur in light of all communication failures.

A slave processor failure may cause maybe, at-least-once, and only-once-type-l references to fail; that is, in addition to their normal semantics under communication 'failures, we must add the possibility that no Op-End-Commit and Result-Commit will occur. A failed master processor causes problems similar to those of a failed slave processor except that the Result-Commit is *guaranteed* not to occur. Table 1 summarizes these points.

The only-once-type'-2 attribute applies to references issued to recoverable slaves from both recoverable and non-recoverable master processes. Only-once-type-2 references cause exactly one Op-End-Commit to be executed, regardless of failures. The Result-Commit always occurs if the master process is recoverable.

These attributes have a major effect on the protocol that is needed to implement remote references. Protocol considerations.are described in Section 2.5.

2.4.2 **Value** and **No-Value References**

All references, except those with maybe semantics, explicitly return a value to the master process. These are called value references. In Figure 2, Result-Commit labels the time at which a value is returned. This value is either provided by the remote operation or by the communication system; in the latter case, it provides an indication that the remote operation has been performed. For only-once-type-2 references, which guarantee that a remote operation will be performed, this response allows the master process to know *when* the remote operation has occurred.

For efficiency considerations, we allow maybe references to not return a value. These are called no-value references. Maybe references do not require a response to achieve their reliability semantics; hence, it would be inefficient to require a response for operations that produce no value.

2.4.3 Synchrony

Remote operations that execute synchronously with respect to a calling processor are called processor-synchronous. Operations that execute synchronously with respect to the calling process only are called process-synchronous; in this case, the processor may execute another process while the remote operation is being performed. Operations that execute asynchronously with respect to only the calling process are called asynchronous. The order in which remote operations complete (i.e., execute Op-End-Commit) is independent from the order of the asynchronous references that initiate them. In summary, a master process can issue processor-synchronous, process-synchronous, or asynchronous references.

Asynchronous and process-synchronous references complicate response processing on the master, because many responses may be pending. Each pending response must be demultiplexed and queued to an individual process, and provision must be made for request-response correlation. Additionally, if there is to be any benefit from the asynchrony of these operations, process switches will likely ensue. These process switches may lead to I/O operations (due to working set changes) and cache invalidation, and may cause increased overhead. Asynchrony results in a more complex protocol for only-once references of both types, because slaves can not automatically discard saved state information when a new request arrives.

When errors occur, processor-synchronous references may require a slight modification of only-once-type-l, only-once-type-2 and at-least-once semantics. This modification is necessary, because it may be impossible to halt all processing on the master if the remote reference takes too long to complete (e.g., because of timeouts and retransmissions). The vast majority of the time, a processor-synchronous reference can be executed synchronously; however, when errors occur, the reference can be re-executed process-synchronously. This technique is used in the Medusa operating system on CM* [Ousterhout et *al*].

No-value references with maybe semantics can be performed efficiently. Processor-synchronous references can also be performed efficiently if requests and responses are short, and the specified remote operation can be quickly executed. They are useful when the cost of doing the remote operation is lower than the additional overhead that would be incurred with process-synchronous or asynchronous references. Process-synchronous references are next most efficient, because they do not require a process to explicitly account for multiple outstanding messages. Flow control, if required, adds to the complexity of asynchronous references.

2.4.4 Inter-Reference Flow Control

Flow control has many meanings, but we consider flow control as a resource reservation system that guarantees a resource is available on the slave. Usually, this is buffering space for requests. Thus, flow control ensures that a master issues requests to a slave below a predetermined rate. Flow control is not useful for process-synchronous or processor-synchronous references, because with these a process can not issue a new reference until the last reference has been acted upon. However, asynchronous references can be either flow-controlled or not-flow-controlled.

Flow control requires that additional state be saved by both master and slave. This has been called the *allocation* [McQuillan], the amount of storage space reserved for buffering additional requests on the slave. Additional packet transmissions may be required to maintain this allocation.

2.4.5 Operation Types: Primary and Secondary

Operations are primary if they are performed by a remote communications process and secondary if they are performed by a regular process. Primary operations can be executed rapidly on the remote processor, because the communications process can be activated without substantial overhead. Furthermore, requests do not cause scheduling or require additional queuing, because there is only one communications process per processor, and we assume that it can be run with low overhead. Finally, the caveat that primary operations must be simple (to avoid overloading the communications process) is a factor contributing to the high speed at which they run. Examples of primary references are causing data to be enqueued in a process' mailbox and initiating remote memory operations.

In comparison, secondary operations require request demultiplexing, request queuing, and more costly process switching on the remote side. Remote subroutine calls are typical examples of secondary references.

2.5. Implementation Considerations — Protocol

The reference attributes and the size of requests and responses affect the communication protocols that can be used to implement remote references. Due to space limitations, this section only addresses the major factors influencing the underlying protocols: the synchrony and reliability attributes, and the size of requests and responses.

Maybe, novalue references require only that the master issue a request. Maybe, value references require that requests contain a sequence number to permit responses, which are sent by the slave, to be correlated with requests. No retransmission mechanism is required for any maybe references.

For at-least-once semantics, (1) requests and responses must contain a sequence number to permit the correlation of requests and responses, and (2) requests must be retransmitted until either a valid response arrives or it can be determined that a processor failure has occurred. Only-once-type-l semantics additionally require that information must be saved by the communications process on the slave to permit the suppression of duplicate requests and allow response retransmission.

Implementations of only-once-type-2 references are similar to those of only-once-type-l, except that the session state must be maintained in type-2 socket table entries on both master and slave. The slave process must be recoverable, and the remote operations that it executes must be atomic: i.e., once a remote operation executes Op-Begin-Commit, it will either execute Op-End-Commit or fail and leave no trace. In addition, both Reference-Commit and Op-End-Commit must be atomic, and both must commit state to stable storage. Result-Commit and Op-Begin-Commit may be atomic and may commit state to stable storage in some instances.

We should also note that for all only-once-type-2 references, Result-commit can not occur until a valid response is received from the slave. This restriction requires that a back-up processor be available for the slave to minimize the duration of failures. Also, only-once-type-2 references require heavy use of stable storage. For efficiency considerations, traditional implementations such as mirrored disks may not be suitable.

At-least-once, only-once-type-l, and only-once-type-2, processor-synchronous references can be implemented using a request/response protocol. With this protocol, the master issues a request to a slave, and the slave returns a response (possibly containing a value) to the master confirming that the remote operation has been performed.

This protocol is useful for processor-synchronous references for two main reasons: First, the amount of space needed to store the last response issued to each master is not large, because responses are necessarily short. Additionally, the total number of socket entries used to save this information is small; it is equal to the number of remote processors containing at least one active master process in session with the slave.

A request/response/end-of-request protocol is useful for asynchronous and some process-synchronous references. In this protocol, the slave's response is acknowledged, allowing the slave to reclaim the space devoted to storing that response. This acknowledgement is required for asynchronous references, because a new request does not ensure that the last response has been received. In this protocol, attention must be paid to allow for recovery in the case of lost end-of-request packets.

Finally, the request/request-received/response/end-of-request protocol is useful if the remote operation will take a long time to execute and the request is long. The request-received packet allows the master to reclaim the space used for holding the request while it is awaiting a response. This protocol is useful for some process-synchronous or asynchronous references.

In all protocols using type-l sockets, the slave must be able to eliminate sockets associated with crashed masters. Two techniques are available: Upon recovery, the master communications process could issue primary references to processors with which it may have communicated, requesting that type-l sockets be eliminated. This reference would be issued over the systems-session. Alternatively, type- 1 sockets could timeout.

2.6 An Example: Only-Onx-Type-2, Asynchronous Rcfcrcnces

This section contains a brief discussion of one rather complex reference type: an only-once-type-2, asynchronous, flow-controlled, secondary, value reference. Its purpose is not to fully specify an implementation but to both demonstrate that this model includes rather complex primitives and exemplify the only-once-type-2 attribute. An example of such a reference type is primitive that reliably writes a page onto remote secondary storage, unlocks that page, and returns a version number.

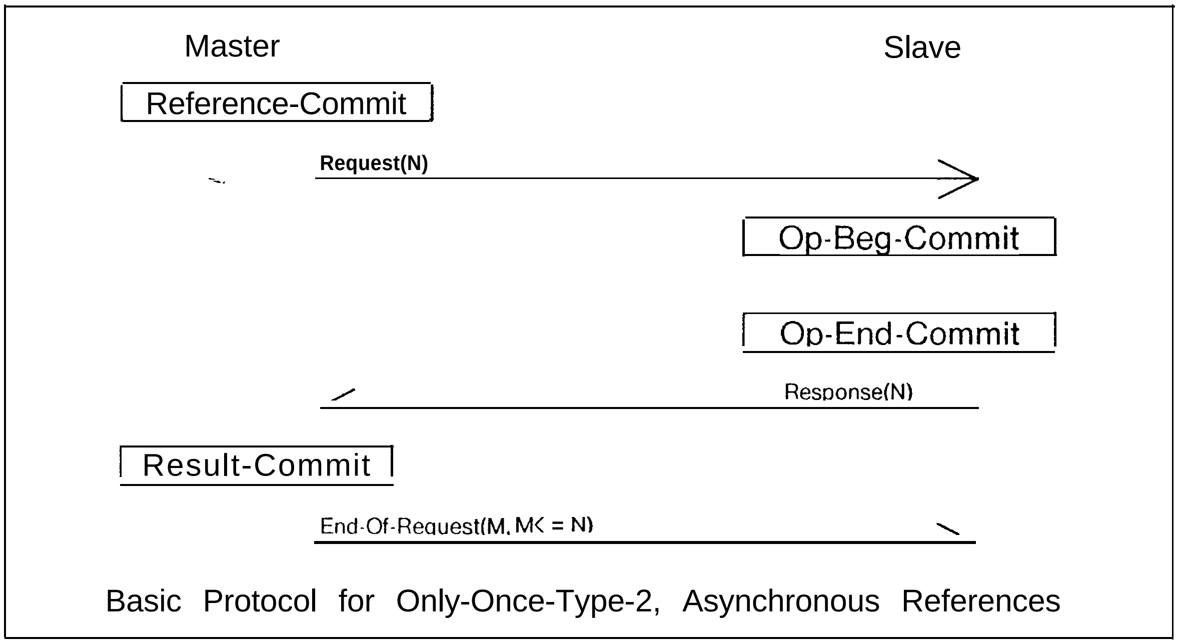


Figure 4

In one possible implementation, after Reference-Commit, a request containing a sequence number N is sent by the master to the slave. (Sequence numbers are ordered, and new remote references initiate transmission of requests with higher sequence numbers.) After the requested operation is performed and Gp-End-Commit is ‹.ecuted, the slave will issue a response, containing the result and the sequence number N, to the master. Normally, that response will be received by the master, and the master will execute Result-Commit and issue an end-of-request packet. The end-of-request packet contains a sequence number M, M<N, and indicates that the master has received all responses through sequence number M. In-this description, we have consciously ignored flow-control and addressing considerations.

To achieve only-once-type-2 semantics in this implementation, the request must be committed to stable storage on the master at Reference-Commit, and it must be retransmitted until Result-Commit occurs. On the slave, Op-End-Commit must be an atomic action that (1) ensures that the operation that is about to be committed is not a duplicate, and (2) if the operation is not a duplicate, commits the reference result, sequence number, and any operation-dependent data to stable storage. The slave then sends a response to the master, containing either the previous or the new result. The Result-Commit must atomically commit to stable storage both the reference's result and an indication that the reference has completed. The master can then reclaim stable storage used to maintain the request and transmit an end-of-request.

The end-of-request does not need to be sent reliably to the slave. This is because each future end-of-request will acknowledge at least as many responses as did the previous end-of-request.

2.7 Discussion

This communication model suggests a communication system with a well-specified and rich class of communication primitives. Table 2 summarizes the feasible reference types. Because many types of references are available, and they differ greatly in implementation costs, distributed systems need only "pay for what they use." Because each reference type provides only highly specific functions, implementations can be easily streamlined allowing highly efficient operation. It should be noted that even the reference types that are most efficient to implement can be directly used in distributed systems.

Choose One from Each Column Subject to Restrictions:

Synchrony Class	Op-Type Class	Flow-Control Class	Reliability Class	Value Returning Class
Processor-Synchronous (See 3)	Primary	Flow-Controlled (See 2)	Maybe	Value
Process-Synchronous	Secondary	Not-Flow-Controlled	At-Least-Once	No-Value (See 1)
Asynchronous			Only-Once-Type-1	
			Only-Once-Type-2 (See 4)	

1.) Only Maybe references can have No-Value.

2.) Only Asynchronous references may be Flow-Controlled.

3.) Processor-Synchronous references must be Primary.

4.) Only-Once-Type-2 references must be directed to recoverable slaves.

Table 2 - Reference Type Summary

Tho: ;h a communication system need not provide all of these reference types, many have direct uses in distributed systems. Simple processor-synchronous operations are useful for implementing remote shared memories, enqueuing small blocks of data, signalling remote processors, etc. Primary, process-synchronous and primary, asynchronous operations are useful for implementing message passing primitives. Secondary, process-synchronous operations are useful for implementing remote subroutine calls and cross-network paging. Finally, secondary, asynchronous operations have their place in the parallel execution of remote subroutine calls. Even the maybe reliability attribute is useful; an example is the transmission of packetized speech.

The operations provided by full-duplex byte stream mechanisms are costly to implement in the context of this model, because they require two asynchronous, primary, flow-controlled, only-once-type- 1 sessions. In this instance, better direct implementations can be constructed. We conjecture that little use would be made of byte stream protocols if a full range of remote references were provided.

Successful implementations of rcmotc references require that simple references bc done with great efficiency. This restriction requires a close integration of the network interface and processor. A memory reference or normal instruction must bc able to trigger the communications process into generating a request. The communications process should be implemented as a combination of software and microcode on the processor and by a specialized network interface, itself possibly microprocessor con trolled. Process switches to the communications process must be very inexpensive and it must bc capable of executing primary operations rapidly. Therefore, it must have fast access to main processor memory. Low latency, high bandwidth stable storage is necessary for efficient implementations of only-once-type-2 references.

3. A Case Study: **Only-Once-Type-1**, Primary, Processor-synchronous References

This section describes a case study that should clarify the above general discussion. Specifically, it describes two implementations of only-once-type-I, value, not-flow-controlled, primary, processor-synchronous references using a request/response protocol. One implementation was done in software and the other in microcode. Comparisons between the two demonstrate the advantages of specializing the interface.

Overall, this work shows that some remote operations can be performed so quickly that they may often bc executed synchronously with respect to the initiating master process. Thus, process switching is unnccessary, except in rare cases where errors or transmission delays arc encountered. This implementation helps to validate the utility of the model, because it shows that some references can be performed efficiently.

3.1. Background

The implementation was done on Xerox Alto computers, a microcoded 16 bit machine with an internal cycle time of 180 nanoseconds and a memory bandwidth of 29 megabits/second ['['hacker *et al*] [Xerox] connected with a 2.94 megabit Ethernet [Shoch-Hupp]. The macroinstruction set as well as peripheral device controllers are implemented on the micromachine through the operation of 16 microcoded tasks, each executing 32 bit microinstructions. Mechanisms exist to switch among tasks in one microcycle. Since I/O devices have the full processing capability and temporary storage of the micromachine at their disposal, as well as the ability to access main memory very efficiently, they can provide highly efficient hardware interfaces.

The macroinstruction set is similar to a Data General Nova [Data General]. For the purposes of this work, various new instructions were added using unimplemented operation codes, which transfer control to a 1K word microcode RAM. The emulator executes macroinstructions at about 3 3 0 KIPS. The Alto contains no protection or virtual memory facilities.

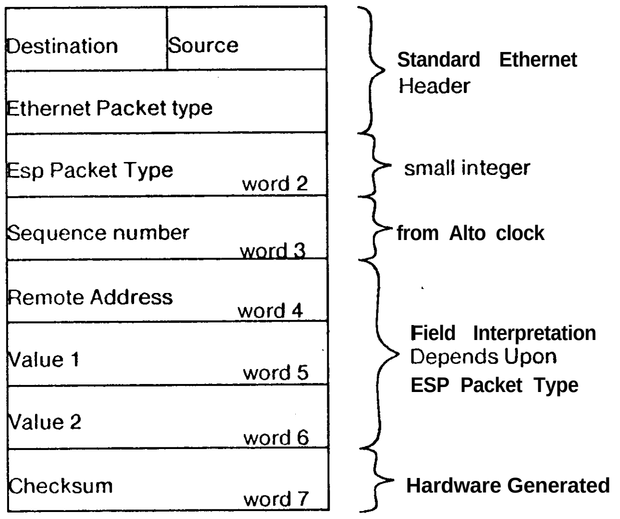
The Ethernet task is responsible for initiating packet transmission and reception, and performing underlying Ethernet protocol manipulation. It was not necessary to modify the Ethernet task although more generality and efficiency could result if this were done.

3.2. **Implementation** – Software Version

We first implemented a software package that provides five subroutines called RLDA, RSTA, RCS, RENQUEUE, and RDEQUEUE for remote load, remote store, remote compare and swap [IBM], remote enqueue, and remote dequeue. The exact semantics arc summarized in the appendix.

These subroutines cause a request packet to bc transmitted to a remote Alto and return control when a proper response packet is received or when an error condition is detected. We have called the message protocol "ESP" for Efficient Synchronous Protocol. See Figure 5 for the packet format.

The software is written entirely in BCPL [Curry et *al*] and uses the raw datagram facilities of PUP Level 0 [Boggs et *al*] for packet transport. Sessions are maintained between each pair of communicating processors. Duplicate elimination is handled by the sequence number field of the ESP packet and socket addressing is implicitly done using the Ethernet Packet Type field.



Esp Packet Format

Figure 5

3.3. **Implementation** -Microcode Version

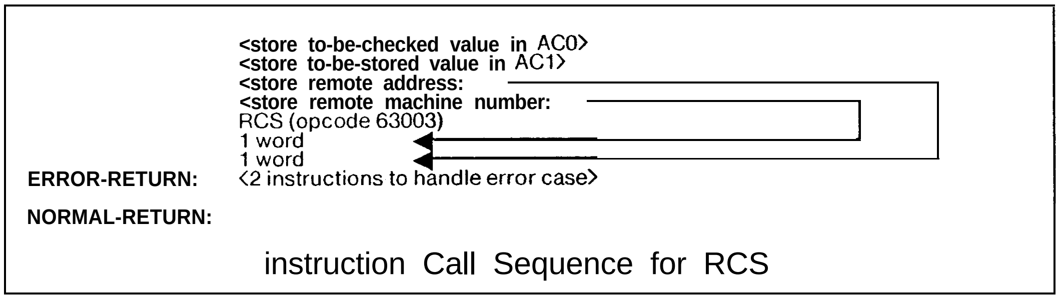
For the purposes of this study, it was sufficient to implement two separate sets of microcode: one allows an Alto to act as a shared memory that executes and responds to ESP request packets; the other implements the RLDA, RSTA, RCS, RENQUEUE and RDEQUEUE instructions, formatting request packets and awaiting responses. The two sets of microcode could be combined to provide exactly the same function as that provided by the software version, including compatibility with standard PUP communication. However, this would require time-consuming modifications to the Ethernet control task and is not necessary to prove the efficiency that can be achieved for only-once-type-I, primary, processor-synchronous references.

Though the microcode is quite similar to the software, it does differ in some respects. First, incoming requests are not queued, because queuing a request would require almost as much work as processing it. Second, the processing time of a request is small in comparison to the amount of time that the Ethernet hardware is busy. Third, instruction decoding is performed to make the references more efficient. Finally, substantial performance benefits are realized by overlapping memory accesses with processing.

The microcode is (surprisingly) simple due to more convenient handling of errors, multi-tasking, and time-outs in the micromachine. It comprises about 280 instructions though this number could be reduced by clever microcoding. A total of 7 hardware registers are used in processing. The microcode executing requests on behalf of a remote machine uses an additional

728 (256 x 3) words of main memory to store the last sequence number and response values for all possible machines connected on the Ethernet. This corresponds to the socket table in Section 2.3.

Usage of the new instructions is illustrated by Figure 6's description of RCS. Instructions take arguments in two general registers as well as in. the two words following the operation code. They skip return on success and return results in one or two registers.



```
                  <store to-be-checked value in AC0>
                  <store to-be-stored value in AC1>
                  <store remote address:
                  <store remote machine number:
                  RCS (opcode 63003)
                  1 word
                  1 word
ERROR-RETURN:     <2 instructions to handle error case>
NORMAL-RETURN:
```

instruction Call Sequence for RCS

Figure 6

On error returns, the sequence number of the request is returned, allowing for additional software retransmission of the request. In our model, this would be done by re-executing the reference as a process-synchronous operation and instructing the communication system to use the previous sequence number. To provide the proper error semantics in light of remote processor failure, a flag can be maintained on the slave that is set to 0 when a machine has been restarted after a failure. If a request arrives and finds this flag set to 0, a response can indicate that a machine failure has occurred prior to this request. Requests always set this flag to 1.

Upon receipt of control following a remote instruction, the microcode first collects information from various places and assembles it in a 7 word block of memory. This includes the machine number of both source and destination, the system time (which is used as a sequence number), various data values from the general registers and the words following the instruction, etc. Before the packet is transmitted, one internal register is set with the number of retransmissions and another with a counter that is continuously counted down to allow for timeouts. A transmission count of 2 and a timeout interval of 3 milliseconds is currently used, a time long enough to permit a long packet to pass. The small transmission count ensures that the processor does not suspend its operation for too long. With these parameters, the *maximum* time a remote reference can halt processing is 6 milliseconds.

When a response packet is received, its source and sequence number are checked to ensure that it is a response from the last request. If these numbers match, values are placed in one or two general registers and the instruction returns. If they do not match, either the machine waits for another packet, retransmission is attempted, or the instruction returns and indicates an error.

At the remote site, the microcode continually checks the Ethernet to see if a new packet has arrived. If an ESP packet arrives, the source byte is used to index into the socket table. If the sequence number of the received packet is the same as that in the corresponding table entry, the request is a duplicate and a response is generated using the state information saved after the first request. If the sequence number differs, the operation specified by the ESP packet type is performed using the remote address and value fields as arguments. Two values resulting from this operation as well as the new sequence number are placed in the table. Finally, a response packet is generated using these values.

3.4. Performance

In the software version, approximately 210 rcmote references can bc cxccutcd per second; this corrcsponds to 4.8 milliscconds/rcfcrcncc on an unloadcd Ethernet. Running at maximum speed, two machines communicating using this software package can impose a 1.8% load on thc Ethcrnet. Using RSTA instructions, this corresponds to a 3.4 Kbit effective transfer rate. This softwarc implementation is likely to be at least 3 times faster than any implementation using the PUP byte strcam protocol, a protocol that providcs a full duplex, rcliablc byte strcam protocol from one machine to another. This difference is duc to the more complex protocol used by the rcliablc byte strcam protocol and the morc general interfaces that it provides.

The microcode version is capable of supporting 6450 references pcr second corresponding to a timc of 155 microseconds/reference. As anothcr characterization, each rcmote rcfcrcnccs takes about 50 macro-instruction timcs. Figure 7 shows a breakdown of thc time spent when an RLDA instruction is executed, assuming no contention on the Ethernet. This timc is rcprescntativc of the times of the othcr instructions as well. Of the 155 microsccondsrcquircd, transmission time accounts for morc than half: 85 microseconds. Local processing lcading up to the rcqucst requires 28 microseconds, processing at the rcmotc site rcquircs 31 microseconds, and local proccssing aftcr thc rcsponsc is rcccived-rcquircs 11 microseconds. Roughly, remote rcfcrcnccs are two orders of magnitude faster than they would be if implemented using thc PUP byte stream protocol.

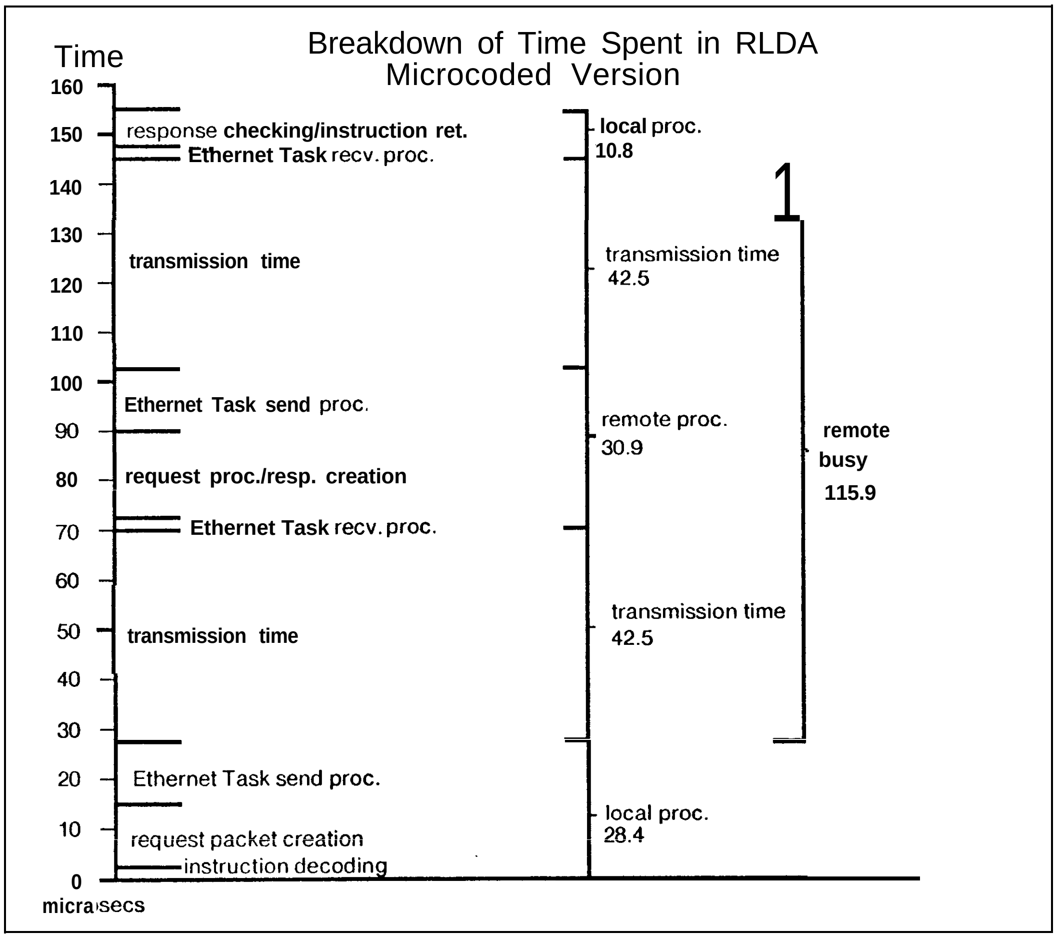


Figure 7

The slave's Ethernet transceiver or processor is busy for 116 microseconds per request. Thus, a shared memory could' maximally support 8600 references per 'second. A processor could initiate 6450 remote instructions per second, placing a load on the Ethernet of about 55%. In practice, neither the shared memory nor a processor issuing references would operate at their maximum rates.

We have measured a single processor issuing RLDA's to a remote memory at the rate of 5000 per second. The difference between 5000 and the theoretical maximum of 6450 can be accounted for by the time necessary to execute the instruction loop iterating over the RLDA's. The Ethernet load generated by this test was 42% or 1.28 megabits. As one would expect, practically no retransmissions or timeouts occurred during this test. Though one would not use RLDA instructions to maximize throughput, the effective transmission data rate in this test was 80 kilobits.

In a more complex test, where two machines attempted to generate 5000 requests per second to a slave, severe contention problems on the shared memory occurred. The conten tion caused less than 3000 references/master/second to be completed. Many retransmissions were required to reliably issue a remote request. Part of this was due to the high load on the memory. The slave does not listen to the Ethernet while it is processing a request and, consequently, misses many requests. The occurance of this problem demonstrates the necessity of not overloading a slave.

Finally, in a test to check the capacity of the Ethernet, two machines made requests to two separate shared memories and collectively put a load on the network about 64% or 1.92 megabits with very few collisions or retransmissions. Each machine was generating about 3750 packets per second. Table 3 summarizes the most important yardsticks.

Description	Software	Microcode
RSTA's / second (achieved)	210	6450 (5000)
microseconds / RSTA	4800	155
Ethernet load (achieved) 1 master = > 1 slave	1.8%	55% (43%)
Real Data Rate (achieved)	3.4 Kbits	103 (80) Kbits

The parenthesized measurements are from BCPL test programs and include the overhead of a test loop.

Performance Summary

Table 3

4. Conclusions

In this paper we first presented a communication model that suggests a taxonomy of reference types. Many of these types have natural uses in distributed systems, and each has an implementation most suited to it. Though relatively complete, the taxonomy could be extended to include more complex type of sessions, such as those with multiple slaves, and to include other semantic attributes. For example, intra-reference flow control has been suggested for references having very variable size requests and responses.

Next, we looked at a highly efficient implementation of one specific reference type. This experiment demonstrates the power of "special-casing" the implementation-both protocols and interface- of a particular reference type. We met our goal [Spector1980] of allowing a processor to issue 1% of its non-instruction memory references to a remote processor and suffer no more than a 50% speed degradation. (An Alto executing a remote reference 1% of the time would slow from 330,000 instructions/second to 208,000 instructions/second – a 37% degradation.) With better hardware and a 10 megabit Ethernet, the remote reference times would be well under 100 microseconds. Additional work on these references must include consideration of protection, virtual memory, and contention on the server.

To further demonstrate the utility of the remote reference/remote operation model, a reasonable subset of communications primitives must be selected and a complete communications system designed. Not all references can be implemented as efficiently as only-once-type-l, primary, processor-synchronous references. But without doubt, they could be implemented much more efficiently than the normal high level communication mechanisms that are currently used on local networks.

We would like to design this communication system with relatively few constraints. For example, to permit high speed operation of processor-synchronous operations, we must use relatively small packet lengths to ensure that packets used for processor-synchronous references do not suffer long latencies. If remote reference times are to be small and higher speed processors are to be supported, a higher capacity network is needed. A ring network like that of Cambridge [Wilkes-Wheeler] or TRW [Blauman] would probably meet these needs.

The network interface must be sufficiently clever to allow reception of back-to-back packets. The communications process should be implemented so that process switches to and from it can be made quickly. To provide only-once-type-2 semantics, some easily accessible stable storage is needed. It would be most beneficial to off-load some portions of the communications process to a fast microprocessor so as not to burden the main processor with unnecessary overhead.

We have ignored the issue of the programming language support for this model. Before this model can be easily applied to distributed systems, specific instantiations of various reference types need to be defined for use within the context of a distributed programming language.

Finally, we need to implement a variety of systems using remote references. With a clever enough communications system implementation and a high speed network (greater than 50 megabits) some of the applications done on CM* [Jones-Gehringer] would work on a local network based multiprocessor. However, many other types of distributed systems should be tested to see if their communication requirements can be naturally and efficiently met by the remote reference/remote operation model presented above.

Acknowledgements

I am indebted to Jim Gray and Richard Pattis for their many readings of this document. I would also like to thank Forest Baskett, David Cifford, John Hennessy, Bruce Lindsay, Jeff Mogul, and Patricia Selinger for valuable suggestions and comments.

Bibliography

[Andler *et al*]
Andler, Sten, Dean Daniels, Alfred Spector: "On Enhancing Local Network Communication Devices," *Proceedings IFIP WG 6.4 International Workshop on Local-Area Networks,* Zurich, Switzerland, August 27-29, 1980.

[Bartlett]
Bartlett, Joel F: *A NonStop™ Operating System*, Proc. of the 11th Hawaii International Conference on System Sciences, Western Periodical Company, Los Angeles, 1978.

[Blauman]
Blauman, Sheldon: "Labeled Slot Multiplexing: A Technique for a High Speed Fiber Optic Based Loop Network," *Proceedings 4th Berkeley Conference on Distributed Data Manipulation and Computer Networks,"* San Francisco, pp. 309-321, August 1979.

[Boggs et al]
Boggs, David R., John F. Shoch, Edward A Taft, Robert M. Metcalfe: *Pup: An Internetwork Architecture,* Research Report CSL-79-10, Xerox Palo Alto Research Center, May 1979.

[Cypser]
Cypser, R .J., *Communications Architectures for Distributed Systems,* Addison-Wesley Publishing Company, Reading, Massachusetts 1978.

[Curry *et al*]
Curry, James E. *et al: BCPL* Reference *Manual*, Xerox Palo Alto Research Center, September 1979.

[Data General]
Introduction to Programming The Nova Computers, 093-000067, Southboro, MA, 1972.

[Folts]
Folts, Harold C.: "Coming of Age: A Long-Awaited Standard for Heterogeneous Networks," *Data Communications,* McGraw Hill, January 1981.

[IBM]
IBM System 370 Principles of Operation, GA22-7000-5, Poughkeepsie, August 1976.

[ISI]
DOD Standard Transmission Control Protocol, RFC-761, Information Sciences Institute, Marina del Ray, California, January 1980.

[Ikeda *et al]*
Ikeda, Katsuo, Ebihara, Yoshihiko, Ishizaka, Michihiro, Fujima, Takao, Nakamura, Tomoo, Kazuhiko, Nakayama: "Computer Network Coupled by 100 MBPS Optical Fiber Ring Bus– System Planning and Ring Bus Subsystem Description," *Proceedings Conzpcon,* November 1980, pp. 159-165.

[Jones-Gehringer]
Jones, Anita K., Edward F. Gehringer, ed.: *The CM* Multiprocessor Project: A Research Review,* Research Report CM U-CS-80-13 1, Carnegie Mellon Universi ty, July 1980.

[Jones-Schwarz]
Jones, Anita K., Peter Schwarz: "Experience Using Multiprocessor Systems - A Status Report, "*ACM Computing Surveys,* 12(2):121-165, June 1980.

[Lampson-Sturgis]
Lampson, Butler, Howard K. Sturgis: *Crash Recovery in a Distributed System,* unpublished, Xerox Palo Alto Research Center, April 1979.

[Liskov]
Liskov, Barbara: *Linguistic Support for Distributed Programs: A Status Report,* Laboratory for Computer Science Computation Structures Group Memo 201, Massachusetts Institue of Technology, October 1980.

[Metcalfe-Boggs]
Metcalfe, R. M. and D. R. Boggs: "Ethernet: distributed packet switching for local computer networks," Comm. *ACM,* 19(7):395-404, July 1976.

[McQuillan]
McQuillan, John M.: *Adaptive Routing Algorithms for Distributed Computer Networks,* Research Report, Harvard University, 1974.

[Nelson]
Nelson, Bruce: *Remote Procedure Call: A Thesis Proposal,* unpublished, Computer Science Department, Carnegie-Mellon University, June 1980.

[Ousterhout *et al*]
Ousterhout, John K., Donald A. Scelza and Pradeep Sindhu: "Medusa: an experiment in distributed operating system structure," *Proceedings 7th Symposium on Operating System Principles,* December 1979.

[Peterson]
Peterson, James L.: "Notes on a Workshop on Distributed Computing," *Operating Systems Review, 143):* 18-27, July 1979.

[Rawson-Metcalfe]
Rawson, E. G., R M. Metcalfe: "Fibernet: Multimode Optical Fibers for Local Computer Networks," *IEEE Transactions on Computer Communication,* COM-26(7), July 1978.

[Shoch-Hupp]
Shoch, J. F. and J. Hupp: "Performance of an Ethernet Local Network--a preliminary report," *Proceedings Local Area Communication Network Symposisum,* NBS, Boston, May 1979.

[Spector1980]
Spector, Alfred Z.: "Extending Local Network Interfaces to Provide More Efficient Interprocess Communication Facilities," *Proceedings ACM Pacific* 80, San Francisco, November 1980.

[Spector1981]
Spector, Alfred Z.: *Multiprocessing Architectures for Local Computer Networks,* forthcoming Ph.D. Dissertation, Stanford University, August 1981.

[Swan et *al*]
Swan, R. J., S. H. Fuller, D. P. Siewiorek: "Cm* A Modular Multi-microprocessor," *Proceedings of the National Computer Conference,* pp. 636-644, AFIPS Press, 1977.

[Thacker e t *al*]
Thacker, C. P., E. M. McCreight, B. W. Lampson, R. F. Sproull, and Dr. R. Boggs, "Alto: A personal computer," in Siewiorek, Bell, and Newell, *Computer Structures: Readings and Examples,* second edition, 1979.

[Wilkes-Wheeler]
Wilkes, M. V., and D. J. Wheeler: "The Cambridge Digital Communication Ring," *Proceedings Local Area Communication Network Symposium,* NBS, Boston, May 1979.

[Xerox]
Xerox Corporation, *ALTO: A Personal Computer System Hardware Manual,* Xerox Palo Alto Research Center, May 1979.

Appendix – Reference Semantics

```
DEFINE RLDA(MACHINE-NUMBER, ADDRESS) =
    IF NO RESPONSE
        RETURN [ERROR-CONDITION, INTERNAL-SEQ-NUMBER]
    RETURN MACHINE-NUMBER[ADDRESS]

DEFINE RSTA(MACHINE-NUMBER, ADDRESS, VALUE) =
    IF NO RESPONSE
        RETURN [ERROR-CONDITION, INTERNAL-SEQ-NUMBER]
    MACHINE-NUMBER[ADDRESS] : = VALUE

DEFINE RCS(MACHINE-NUMBER, ADDRESS, VALUE-1, VALUE-2) =
    IF NO RESPONSE
        RETURN [ERROR-CONDITION, INTERNAL-SEQ-NUMBER]
    IF MACHINE-NUMBER[ADDRESS] EQ VALUE-1
    THEN BEGIN
        MACHINE-NUMBER[ADDRESS] : = VALUE-2
        RETURN IS-EQUAL
    END ELSE BEGIN
        VALUE-1 : = MACHINE-NUMBER[ADDRESS]
        RETURN IS-NOT-EQUAL
    END

DEFINE RENQUEUE(MACHINE-NUMBER, ADDRESS, VALUE) =
    IF NO RESPONSE
        RETURN [ERROR-CONDITION, INTERNAL-SEQ-NUMBER]
    IF FULL-QUEUE(MACHINE-NUMBER[ADDRESS])
        THEN RETURN IS-FULL
    ELSE ENQUEUE( MACHINE-NUMBER[ADDRESS], VALUE)

DEFINE RDEQUEUE(MACHINE-NUMBER, ADDRESS) =
    IF NO RESPONSE
        RETURN [ERROR-CONDITION, INTERNAL-SEQ-NUMBER]
    IF EMPTY-QUEUE(MACHINE-NUMBER[ADDRESS])
        THEN RETURN IS-EMPTY
        ELSE RETURN DEQUEUE(MACHINE-NUMBER[ADDRESS])
```

Notes: All remote references arc done atomically. The expression referred to as MACHINE-NUMBER[ADDRESS] refers to absolute memory address ADDRESS on the processor referred to by MACHINE-NUMBER.