Operating Systems
Anita K. Jones
Editor

Performing Remote Operations Efficiently on a Local Computer Network

Alfred Z. Spector
Stanford University

A communication model is described that can serve as a basis for a highly efficient communication subsystem for local networks. The model contains a taxonomy of communication instructions that can be implemented efficiently and can be a good basis for interprocessor communication. These communication instructions, called remote references, cause an operation to be performed by a remote process and, optionally, cause a value to be returned. This paper also presents implementation considerations for a communication system based upon the model and describes an experimental communication subsystem that provides one class of remote references. These remote references take about 150 microseconds or 50 average instruction times to perform on Xerox Alto computers connected by a 2.94 megabit Ethernet.

CR Categories and Subject Descriptors: C.2.2 **[Computer Communication Networks]**: Networks Protocol—protocol architecture; C.2.4 **[Computer Communication Networks]**: Distributed Systems—*network operating systems*; C.2.5 **[Computer Communication Networks]**: Local Networks—buses, rings; D.4.4 **[Operating Systems]**: Communication Management—*message sending, network communication*; D.4.5 **[Operating Systems]**: Organization and Design—*distributed systems.*

General Terms: Performance, Reliability

Additional Key Words and Phrases: efficient communication, transactions, communication models

1. Introduction

This paper discusses efficient communication techniques for very high speed local networks. A major question that this paper attempts to answer is how to reduce the traditionally high processing overhead of network communication. This overhead, characterized by the time to send a null message and receive a null answer, was reported by Peterson to be about 20 milliseconds on a variety of systems [21]. If communication processing overhead can be reduced and processors are interconnected with high bandwidth, low latency networks, distributed programs having a relatively fine granularity of parallelism can be executed. Networks operating at up to 100 megabits/second, developed at TRW, Mitsubishi, and Xerox, exemplify the technology that will permit small data transmission times to be achieved [2], [13], [23].

Ultimately, studies of efficient communication must be concerned with the specification and implementation of language-level primitives. For example, Cook, Hoare, Liskov, and Nelson have written about remote procedure calls and message passing primitives [4], [10], [17], [19]. However, we concentrate on a communication subsystem on which to base these language-level primitives. To provide overall efficiency, this intermediate communication subsystem must be a good foundation on which to implement the desired high level primitives. Also, it must be specialized enough to be implemented efficiently using the semi-reliable packet transmission facility that most local networks provide. Efficient implementations may require the use of microcode or specialized hardware.

The particular communication subsystem that we propose differs from typical network communication subsystems in two ways. First, we advocate an integrated implementation approach rather than the layered approach discussed by Zimmerman [31]. Though layered approaches facilitate the use of similar communication subsystems on heterogeneous networks and permit simplified design and maintenance of network software, the crossing of layer boundaries results in decreased efficiency. Second, we place more emphasis on ensuring that the functions provided by the communication subsystem are suitable for implementing high level primitives. Rather than providing, for example, only a transmission function for asynchronous byte streams, we propose that the communication subsystem provide functions that can be used more easily in implementing high level communication primitives. These functions must not be too complex (e.g., guaranteeing reliability when it is not necessary) nor must they be so simple that implementations of high level primitives using them are difficult or inefficient. In some instances, we hope for almost a trivial mapping from language-level functions to those provided by the communication subsystem. In related work, Nelson and Popek have discussed the benefits of streamlined implementations of communication primitives in the context of programming language constructs and operating systems, respectively, [19], [22].

The initial sections of this paper discuss functions that should be provided by a communication subsystem

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.

Author's present address: Alfred Z. Spector, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

and techniques for implementing these functions. More specifically, Sec. 2 presents a communication model called the *remote reference/remote operation* model in which a taxonomy of communication primitives is defined. We argue that a communication subsystem suggested by this model can provide powerful primitives yet be implemented efficiently. Section 3 presents implementation considerations for such a subsystem, and Sec. 4 exemplifies a highly reliable communication primitive that it could provide.

Section 5 adds substance to the discussion of the remote reference/remote operation model by describing an experiment in which a simple type of communication instruction was microcoded on Xerox Alto computers using the 2.94 megabit experimental Ethernet. (Information on the Alto and the Ethernet can be found in [18] and [30].) We show that these instructions execute quickly: they typically take about 150 microseconds or about 50 macroinstruction times on this hardware. This time is about two orders of magnitude faster than could be expected if they were implemented in a conventional way. This improvement in performance is due to three factors: the specialization of the communication interface, the use of simplified protocols, and the direct implementation in microcode.

2. The Remote Reference/Remote Operation Model

In the remote reference/remote operation model, a process executes communication instructions called *remote references*; each remote reference causes a single *remote operation* to be performed by a remotely located process. The remote operation may return a value to the caller. A remote reference is analogous to a subroutine call instruction that specifies certain properties concerning the execution of the subroutine. During the execution of any remote reference, the process that issues the reference is called the *master* and the process that executes the operation is called the *slave*.

Remote operations can vary greatly in complexity, from the simplicity of a memory access to the complexity of an asynchronously executing 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 queue manager (at the remote site) is the slave.

There are four reasons why this model includes primitives that not only transmit data but also initiate operations and return their results.

1. *Utility of Primitives.* An implementation of high level primitives based upon remote references can be efficient because remote references are relatively powerful and reduce the need for costly software (and protocol) layering on top of them. Examples of communication primitives that can be naturally implemented using remote references are remote memory references, remote subroutine calls, and message passing operations. Specific examples include the SIGP operation on the IBM 370 [11], "requests" in the Tandem Guardian system [1], and certain Cm* operating system functions [14], [20].

2. *Support for Reliability.* In the case of a transmission medium with high reliability such as a local network, Saltzer [24] suggests that providing "end-to-end" reliability is often more important than providing reliable message transmission. Because references initiate remote operations, such "end-to-end" reliability can be supported by remote references; references can have an attribute that ensures that an operation is performed, not merely that a message is delivered. (As will be discussed in Sec. 3.3, the communication system can only assist in providing high reliability; the remote operation must also be specially implemented.)

3. *Potential for Protocol Simplification.* Many types of remote references can be implemented using simpler protocols than are required for more general and common communication mechanisms such as byte stream primitives.

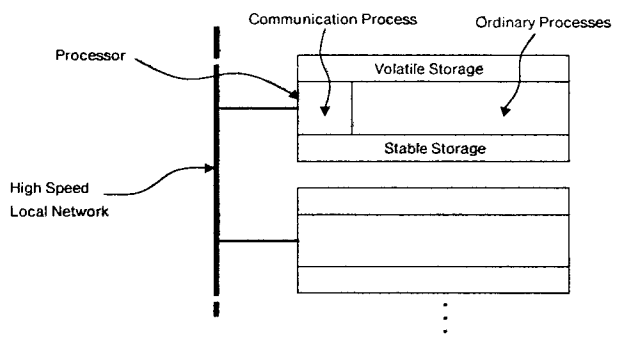
4. *Potential for Efficient Implementation.* Remote references are specific enough that they can be specially implemented, perhaps in microcode or hardware.

2.1 Definitions and Assumptions

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

Data is exchanged between 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. The reader should note that this paper does not discuss internetworks.

Fig. 1. Underlying Architecture.



As will become clear in Sec. 2.3, some remote references are intended to provide very fine grain communication. Such remote references are practical only if transmission latencies are very small for short packets. If many (e.g., 100) processors must also be supported, a high capacity (e.g., 100 megabits/second) network may be necessary.

In the model, processors occasionally fail in a detectable fashion and are then restarted. In addition, processors contain *processes* with names unique to that processor. There are two classes of processes: *regular processes* that disappear after a processor failure and *recoverable processes* that are automatically reincarnated after a processor crash and are reset to one of several predetermined states. Recoverable processes require the availability of *stable storage*—storage that survives failures [16]. Processors are sufficiently reliable to allow recoverable processes to make progress.

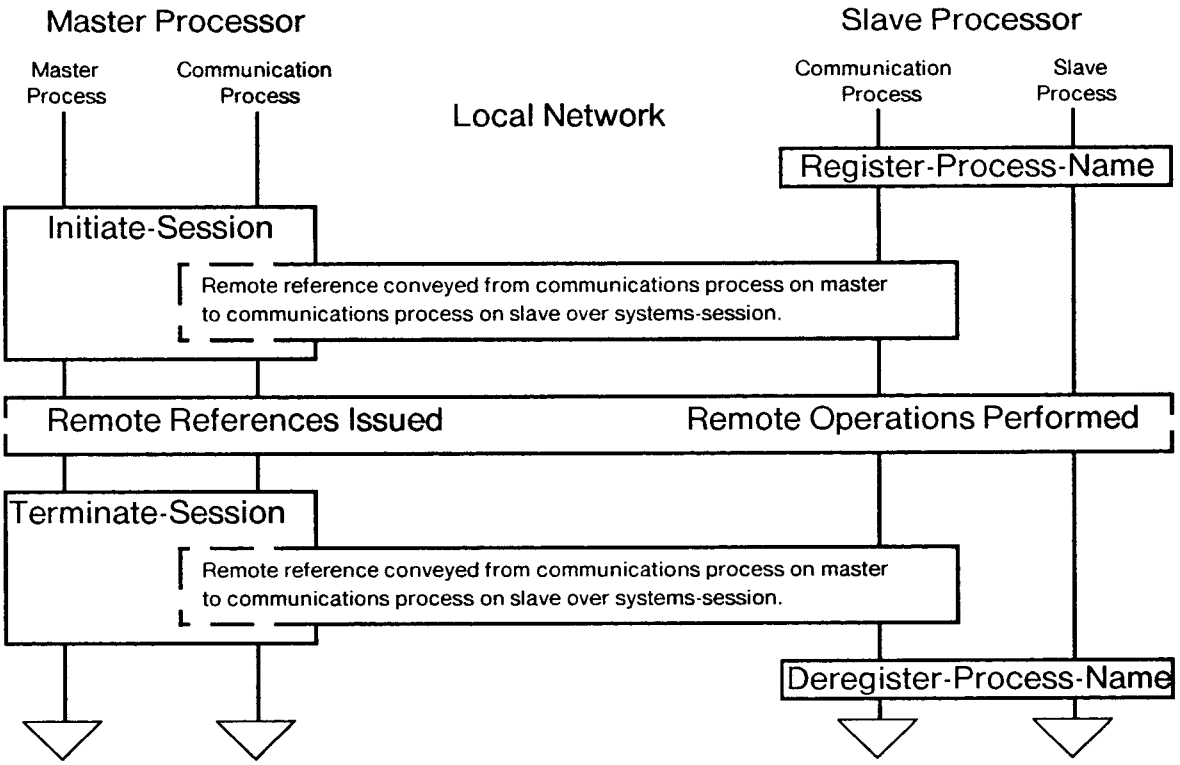
We assume that the regular and recoverable processes are used to provide one or more *ordinary* processes that are used by applications. We also assume the existence of a distinguished process called the *communication process.* This process is recoverable and is specially implemented—off-loaded or at interrupt level—so that it can be activated 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 Sec. 2.3.) The communication process does not have the full capabilities of other processes, but it can perform the following functions efficiently.

1. The transmission and reception of packets on the local network. Only the communication process accesses the network-specific hardware directly.
2. The manipulation of specialized communication state information. The communication process can quickly access the state tables described in Sec. 3.1. Some table entries need to be saved in stable storage.
3. The execution of communication primitives issued by other processes on the same processor. This process implements the communication interface seen by other processes.
4. The execution of simple operations initiated by a remote communication process; i.e., simple requests can be directed to a remote communication process for efficient processing.

2.2 Model Description

Figure 2 illustrates the remote reference/remote operation model in the absence of communication 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 communication process initiates the appropriate remote operation by issuing an action called **Op-Begin-Commit.** The remote operation is complete only when the action labeled **Op-End-Commit** has been executed. If the remote operation produces

Fig. 2. Normal Operation of Remote References.



a value, the communication process on the slave issues a *response* to the master, which causes the action labeled **Result-Commit** to be executed. For remote reference types having certain reliability semantics, some of these four actions must be *atomic* (i.e., execute indivisibly) and either correctly write a record on stable storage, or fail completely and do nothing. (See [9] for a discussion of atomic actions.)

There are many possible protocols for implementing remote references. In simple cases, 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. Section 3.4 contains a more detailed discussion of this.

2.3 Reference Taxonomy

Remote references have associated with them five attribute classes. These are based on the following: varying reliability semantics, whether a value is returned, how the remote operation occurs temporally with respect to the reference, the need for flow control, and the kind of process by which the operation will be performed. These attribute classes have been 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. Sections 2.3.1 through 2.3.5 describe these five different attributes in more detail.

2.3.1 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 communication and processor failures. They are summarized in Table I and described below.

The four reliability attributes, listed in increasing order of function complexity and implementation cost, are as follows: **maybe, at-least-once, only-once-type-1,** 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 effect of the four attributes under conditions of communication failure (i.e., lost packets) is summarized below.

The **maybe** attribute: 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.

The **at-least-once** attribute: An **Op-End-Commit** will be performed one or more times. The **Result-Commit** will also occur one or more times.

The **only-once-type-1** or **only-once-type-2** attribute: Exactly one **Op-End-Commit** and one **Result-Commit** will occur regardless of communication failures.

A slave processor failure may cause **maybe, at-least-once,** and **only-once-type-1** 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 I summarizes these points.

The **only-once-type-2** attribute applies to references issued to recoverable slaves from both recoverable and nonrecoverable 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.

Table I. Reliability Semantics Survey.

	Reference Semantics Under Different Failure Conditions				
Protocol Class	No Failures	Lost Packets	Lost Packets & Slave Failure	Lost Packets & Master Failure	Lost Packets, Master & Slave 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	op performed: 0,1 result-commit: 0
At-Least-Once	op performed: 1 result-commit: 1	op performed: $\geqq$ 1 result-commit: $\geqq$ 1	op performed: $\geqq$ 0 result-commit: $\geqq$ 0	op performed: $\geqq$ 0 result-commit: 0	op performed: $\geqq$ 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	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	regular master process op performed: 1 result commit: 0 recoverable master process op performed: 1 result-commit: 1	regular master process op performed: 1 result-commit: 0 recoverable master process op performed: 1 result-commit: 1

These attributes have a major effect on the protocol that is needed to implement remote references. Protocol considerations are discussed in Sec. 3.4.

2.3.2 Value and Novalue References

All references, except those with **maybe** semantics, explicitly return a value to the master process. These are called **value** references. In Fig. 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.

To permit greater efficiency, we allow **maybe** references to not return a value. These are called **novalue** 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.

2.3.3 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 of the order of the **asynchronous** references that initiate them. In summary, a master process can issue **processor-synchronous**, **process-synchronous**, or **asynchronous** references.

References have been divided into these three synchrony classes because of the different implementation efficiency that is possible for each. For example, **processor-synchronous** references can be implemented very efficiently and permit fine grained communication; **process-synchronous** references require task switching on the master; and **asynchronous** references usually require more complex protocols due to the existence of multiple outstanding requests. These considerations are presented in more detail in Sec. 3.2.

2.3.4 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 cannot issue a new reference until the last reference has been acted upon. However, **asynchronous** references can be either **flow-controlled** or **not-flow-controlled**. When required, flow control adds to the cost of executing remote references.

2.3.5 Operation Types: Primary and Secondary

Operations are **primary** if they are performed by a remote communication process and **secondary** if they are performed by an ordinary process. **Primary** operations can be executed rapidly on the remote processor, because the communication process can be activated without substantial overhead. Furthermore, requests do not cause scheduling or require additional queueing, because there is only one communication 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 communication 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 queueing, and more costly process switching on the remote side. Remote subroutine calls are typical examples of **secondary** references.

2.4 Discussion

This communication model provides a large number of well-specified communication primitives. Table II summarizes the feasible reference types. Because many types of references are available, and because 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 specialized thereby supporting highly efficient operation.

Though a communication system based upon the model need not provide all of these reference types,

Table II. Reference Type Summary.

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 Only-Once-Type-1	No-Value (See 1)
Asynchronous			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 should be Primary.
4) Only-Once-Type-2 references must be directed to recoverable slaves.

many have direct uses in distributed systems. **Primary, processor-synchronous** references are useful for sharing memory, enqueueing small blocks of data, signalling remote processors, etc. **Primary, process-synchronous** and **primary, asynchronous** references are useful for implementing message passing primitives. **Secondary, process-synchronous** references are useful for implementing remote subroutine calls and cross-network paging. Finally, **secondary, asynchronous** references 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.

3. Implementation Considerations

The preceding section described a communication model and introduced a taxonomy of primitives. Whereas Sec. 2 concentrated on defining the semantics of remote references, this section presents implementation considerations for a communication subsystem based upon the model. We envision that such a system would predefine some remote references and provide mechanisms for the definition of others. Predefined remote references would include **primary** remote references and remote references that are used in the definition of others. The predefined **primary** references would be highly optimized and provide low overhead fine granularity communication.

First, this section describes *sessions*; these are connections between processes over which references are conveyed. Then, general implementation issues regarding the reliability and synchrony attributes are presented. This section concludes with a discussion of protocols and the circumstances under which they can be used.

3.1 Sessions and Sockets

A session, as defined in this paper, is a logical connection over which a single master process can issue requests and a single slave process can issue responses. Associated with a session are the semantic attributes of the references which are conveyed over it. For some attributes, considerable state information (for such purposes as flow control or duplicate detection) must be maintained. In this work processes are assumed to be located on fixed processor nodes for the life of the session.

When a session is established, one *socket* is created on each of the master and slave processors. These sockets serve three main functions: First, packets associated with that session are addressed using references to these sockets. Second, sockets contain information that permits requests and responses to be mapped to individual processes. Third and most importantly, they contain state information that enables sessions to provide specific semantic attributes for references conveyed during that session. Sockets for active sessions are contained in a per-processor socket table that is accessed by the communication process and implemented using both volatile and stable storage.

There are two types of sockets: *regular* sockets, which do not survive processor crashes and *recoverable* sockets, which use stable storage and do survive crashes. Recoverable sockets can be used by recoverable processes to ensure that their communication capabilities are not lost after processor failures. After a failed processor has been restarted, a recoverable process associated with recoverable sockets can continue executing references if it is a master or receiving requests if it is a slave. No regular sockets survive a processor restart.

A session can be considered to be a distributed abstract data object that is manipulated by two cooperating communication processes via the two sockets. There are four 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**. In some instances, a master may have a **Receive-Response** outstanding and receive an interrupt when a response arrives. Sometimes, **Receive-Response** is issued in combination with **Issue-Reference.**

Receive-Request permits a slave to receive a request from its master. In some instances, a slave may have a **Receive-Request** outstanding and may receive an interrupt when a request arrives.

Return-Response allows a slave to return a result to its master and causes an **Op-End-Commit** to be executed locally.

Issue-Reference, **Receive-Response**, **Receive-Request**, and **Return-Response** are generic names for primitives whose implementations are application dependent; in fact, their call syntax will often be modified 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 communication processes to permit each communication process to act as both a master and a slave to each other's communication process. These sessions, called *systems-sessions*, allow other sessions between noncommunication processes to be created and destroyed. Communiction processes provide the following primitives.

Register-Process-Name is processed locally on the slave processor and registers a slave process name with the local communication process, and it specifies how requests for that process will be conveyed to it; for example, via interrupt or queueing. It also specifies the type of session in which the slave will participate. This permits the communication process to respond to requests asking for sessions with this slave.

Deregister-Process-Name is processed locally on the slave processor and expunges a slave process name from the socket table of the local communication process. It cannot be issued if a session is established. (The session must first be terminated with **Terminate-Session**.)

Initiate-Session is executed by a master and establishes a session with a previously registered slave. It requires as arguments the remote slave process name and address, the desired type of session (see Sec. 2.3), 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 communication process on the slave. This predefined remote reference serves the purpose of an initial connection protocol such as PUP's rendezvous protocol [3]. Additionally, it allows the presetting of defaults for that session.

Terminate-Session eliminates a session. This predefined remote reference requires the session number as an argument and initiates a remote reference to the remote communication process. It can be executed by either a master or slave.

Figure 3 illustrates the initiation and termination of a session.

In summary, the communication process performs four main functions: it maintains systems-sessions; it supervises the initiation and closing of the other sessions; it accepts references from a master process, initiates their remote execution and possibly performs certain actions to inform the master of the result; and it accepts remote requests from the network, awakens the slave process, if necessary, and possibly sends responses to the master.

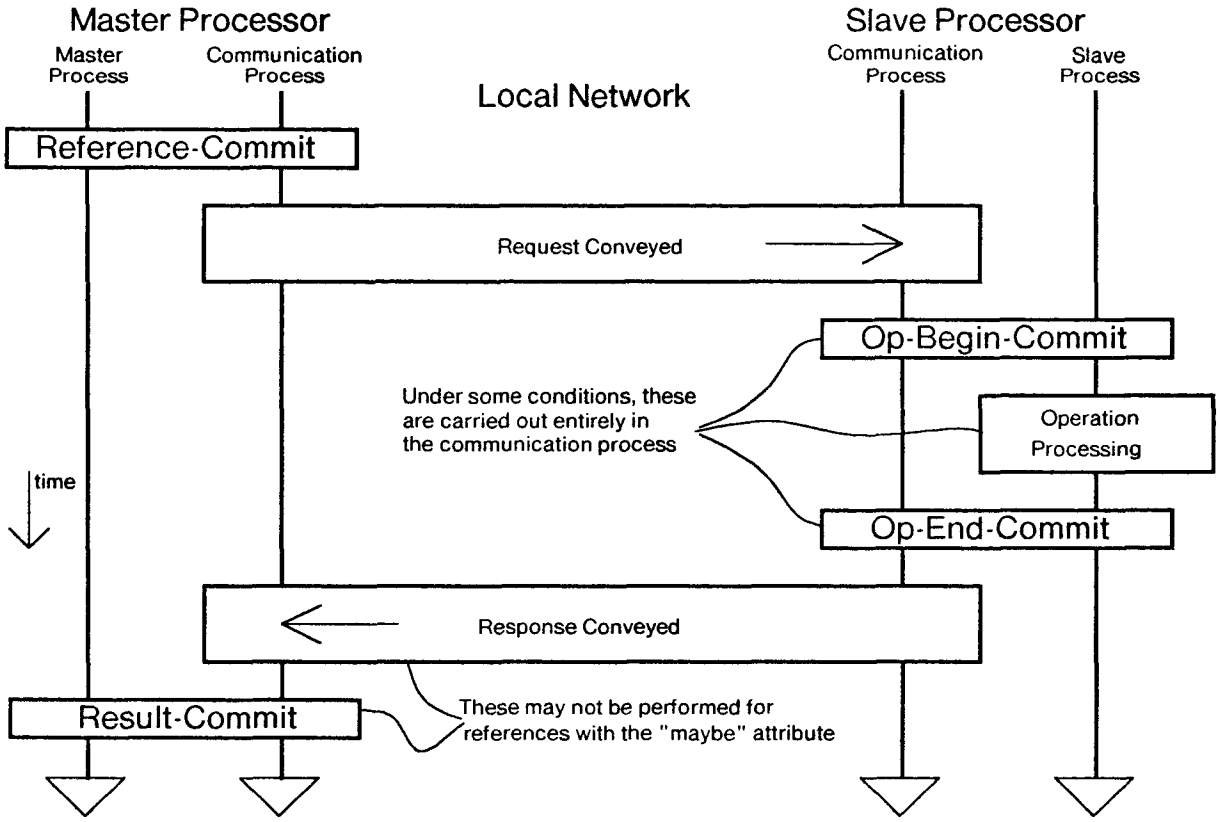
The concepts of sessions and sockets are not unique to this work. For example, sockets are called half-sessions in SNA [5] and TCB's in TCP [12], and sessions are implemented in Level 5 in the OSI protocol hierarchy [8]. However, in this work, there are many types of sessions, each fulfilling particular 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.

3.2 The Synchrony Attributes

Processor-synchronous references can be performed efficiently if requests and responses are short, and if the specified remote operation can be executed quickly. 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. When errors occur that would unduly slow the operation of **processor-synchronous** references, the reference can time-out and be re-executed in a **process-synchronous** fashion. In this way, the master processor will not be halted for too long.

Process-synchronous references are the next most efficient references, because they do not require a master process to account explicitly for multiple outstanding requests. **Asynchronous, only-once** references of both types require a more complex protocol than **processor-synchronous** or **process-synchronous** references, because a master can issue new references before previous references have completed. Because of this, slaves cannot

Fig. 3. Session Initiation and Termination.



automatically discard saved state information concerning a previous request when a new request arrives.

With **process-synchronous** and **asynchronous** references, each response must be demultiplexed and queued to an individual process, and request–response correlation is more complicated. Additionally, potentially costly process switches often occur.

Asynchronous references may require flow control. When needed, flow control requires that the master save information that reflects the amount of storage space reserved for buffering additional requests on the slave. Maintaining this information requires additional information to be transferred between the slave and the master.

3.3 The Reliability Attributes

Maybe references do not require any retransmission mechanism. However, to achieve **at-least-once** semantics, the master must transmit a request to the slave until either a valid response is returned, or it can be determined that a processor failure has occurred. **Only-once-type-1** semantics additionally require that information must be saved by the communication process on the slave to permit the suppression of duplicate requests and allow response retransmissions.

Implementations of **only-once-type-2** references are similar to those of **only-once-type-1**, except that the session state must be maintained in recoverable sockets on both master and slave. The slave process must be recoverable, and the remote operations that it executes must be transactions; for example, 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** cannot occur until a valid response is received from the slave. If the duration of **only-once-type-2** references is always to be small, a backup processor (that can reference stable storage) must be available for the slave to minimize the duration of failures. Also, **only-once-type-2** references require heavy use of stable storage. Traditional implementations such as mirrored disks are probably not suitable for reasons of efficiency.

3.4 Protocol

The reference attributes, the amount of time required for the remote operation to be performed, and the amount of data that must be conveyed between master and slave affect the communication protocols that can be used to implement remote references. The first protocol issue concerns requests and responses that do not fit in a single packet. This issue is important because local networks may enforce small maximum packet sizes to lower the transmission latency for small packets or to decrease the size of the packet buffers that run at the speed of the network.

Requests and responses that do not fit within a single packet can be transmitted as *multipackets*, or sequences of packets. Multipackets are an extension of the basic transmission facilities of the network to permit the efficient transmission of larger amounts of data. Multipackets take advantage of the underlying reliability of the network; hence, packets in a multipacket are neither acknowledged nor retransmitted. Many schemes are possible for multipackets; we describe a scheme based upon the use of a checksum over all packets in the multipacket in [26]. Aside from the checksum, this scheme requires only one byte of overhead per packet and permits single packet requests and responses to be interleaved with the receipt or transmission of multipackets.

Once arbitrarily long requests and responses can be issued, three protocols, the *request* (**R**) protocol, the *request/response* (**RR**) protocol, the *request/response/acknowledge–response* (**RRA**) protocol, are satisfactory for implementing the various types of remote references. (Occasionally, other protocols provide higher performance; these are discussed in [26].) Flow-control can be added to the **RR** and **RRA** protocol by providing *allocation* fields in responses that indicate the amount of space reserved on the slave for additional requests.

The **R** protocol is useful for **maybe, novalue, not-flow-controlled** references. Data is encapsulated into a request and transmitted to the slave.

The **RR** protocol is useful for many types of references. With it, the master initiates a remote operation by issuing a request, and the slave returns a result (either an explicit value or an acknowledgment) by issuing a response. In the **RR** protocol, requests and responses must contain a unique identifier that permits them to be matched to each other. The **RR** protocol is efficient to implement and can be used when the following two conditions are satisfied.

1. The master does not have to buffer too much data while it is awaiting a response from the slave. If the requests are very long and the remote operation requires a long time, buffer space may be wasted.
2. The slave does not have to buffer too much data. For example, consider using this protocol to implement **asynchronous, only-once** references of both types. The **only-once** property requires that the slave retain sufficient information so it can issue duplicates until the master has reliably received a response. Because this protocol does not inform the slave when a response has been received, the slave can never reclaim storage used for storing duplicate responses.

With **processor-synchronous** and **process-synchronous** references, the issuance of a new reference implies the receipt of the previous response. Because of this fact, the **RR** protocol is potentially useful for all **process-synchronous** and **processor-synchronous** references. It is also useful for **asynchronous, maybe, value** references,

asynchronous, **maybe**, **novalue**, **flow-controlled** references, and **asynchronous**, **at-least-once** references because these do not require any inter-reference state to be maintained on the slave. The **RR** protocol will always be the protocol of choice for **synchronous** references because requests and responses are necessarily short. Whether or not the **RR** protocol is used for **process-synchronous** references depends upon the length of requests and responses.

The **RRA** protocol can be used to lower the amount of storage that must be buffered on the slave. In this protocol, the slave's response is acknowledged, allowing the slave to reclaim space devoted to storing that response. It is similar to the **RR** protocol except that the master additionally issues an acknowledge–response to indicate that it has received certain responses, and the unique identifiers contained in requests and responses must be sequence numbers (i.e., ordered.) The acknowledge–response is interpreted as acknowledging the receipt of all responses that have a sequence number $\leq$ M, for some M. This interpretation ensures that the loss of an acknowledge–response is harmless. When a slave receives an acknowledge–response, M, it is free to delete all saved state asssociated with responses that have sequence numbers $\leq$ M.

With the **RRA** protocol, the master may not be able to acknowledge all responses immediately. This is because the master may receive responses for **asynchronous** references out of order. The master will be able to acknowledge a newly received response immediately only if all responses with lower sequence numbers have arrived. (If this causes an important delay in deleting state on the slave, a separate sequence number for responses can be added by the slave, and acknowledge–responses can acknowledge this sequence number.)

The **RRA** protocol may be useful for **only-once**, **process-synchronous** references, but more usually would be used for **only-once**, **asynchronous** references. The acknowledge–response packet is required for **asynchronous** references because a new request does not ensure that the last response has been received.

In all protocols using regular sockets, the slave must be able to eliminate sockets associated with crashed masters. Many techniques are available: for example, upon recovery, the master communication process could issue references—over the systems session—to communication processes on processors with which it may have communicated, requesting that regular sockets be eliminated. Alternatively, regular sockets could timeout.

4. An Example: Only-Once-Type-2, Asynchronous References

This section contains a brief discussion of one rather complex reference type: an **only-once-type-2**, **asynchronous**, **value** reference. (In this description, we consciously ignore flow-control and addressing considerations due to space limitations.) The purpose is not to fully specify an implementation but to demonstrate that this model includes rather complex primitives and to exemplify the **only-once-type-2** attribute. An example of such a reference type is a remote reference that reliably causes a slave process to write a page onto remote secondary storage, unlock that page, and return a version number.

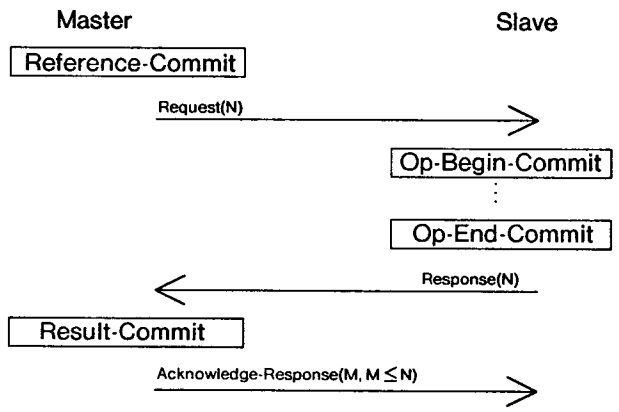
In an implementation using the **RRA** protocol, if **Reference-Commit** is successfully executed, a stable entry (containing the request, with sequence number N, that is to be sent to the slave) is made atomically in the master's socket. (Sequence numbers are ordered, and new remote references initiate transmission of requests with higher sequence numbers.) The request is then repeatedly transmitted by the master to the slave until **Result-Commit** occurs.

After the requested operation is performed and **Op-End-Commit** is executed, the slave issues a response, containing the result and the sequence number N, to the master. Normally, the master receives this response, and it executes **Result-Commit**. **Result-Commit** attempts to commit atomically to stable storage both the reference's result and an indication that the reference has completed.

After **Result-Commit** has occurred, the master sends an acknowledge–response to the slave. The acknowledge–response contains a sequence number M, M $\leq$ N, and indicates that the master has received all responses through sequence number M. The acknowledge–response does not need to be sent reliably to the slave because each future acknowledge–response will acknowledge at least as many responses as did the previous one. After the master has issued the acknowledge–response, the communication process can reclaim all storage except for the indication that the reference has completed. This protocol is illustrated in Fig. 4.

On the slave, **Op-End-Commit** must be an atomic action that first checks the operation that is about to be committed to see that it is not a duplicate. If the operation is not a duplicate, it then 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.

Fig. 4. Illustration of RRA Protocol for **only-once-type-2, Asynchronous** References. Illustrated case is where there are no lost packets.



In fact, the code sequence on the slave from **Op-Begin-Commit** to **Op-End-Commit** corresponds to a transaction where the *begin-transaction* takes a unique identifier as an argument and the *commit-transaction* commits the transaction only if the invocation associated with that unique identifier has not already committed.

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

In the previous sections, many types of remote references were described. The references vary substantially, both in their intended use, and in their implementation. In this section, however, we turn away from the generality of the model in order to study in detail one class of primitives. Our three goals are as follows: to show how the remote reference model terminology applies to a specific example; to illustrate the direct implementation approach that we advocate and show that specialized implementations are feasible; and to show some performance statistics that are indicative of the communication efficiency possible on local networks.

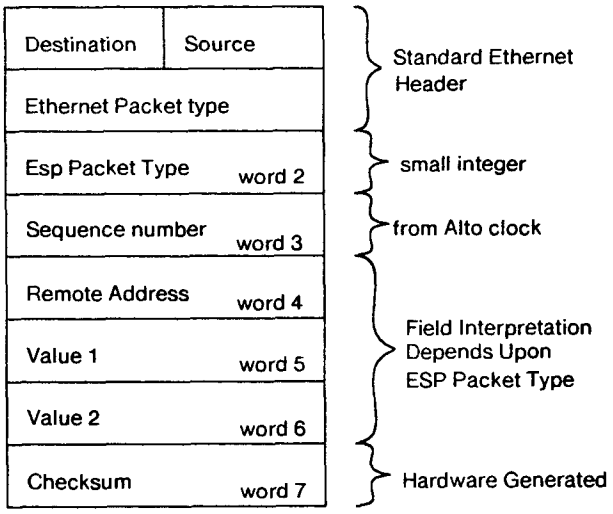
Below, we describe two implementations (one in software, the other in microcode) of **only-once-type-1, value, not-flow-controlled, primary, processor-synchronous** references using a request/response protocol. The software version is a layered implementation with which the performance of the microcoded version can be compared.

The implemented references are called RLDA, RSTA, RENQUEUE, RDEQUEUE, and RCS, and initiate remote load, remote store, remote enqueue, remote dequeue, and remote compare and swap operations, respectively, on a remote machine. The latter operation is similar to the IBM 370 *CS* instruction [11]. The exact semantics of these references are summarized in the appendix. Both implementations use an **RR** protocol which is called "ESP" for *E*fficient *S*ynchronous *P*rotocol. (See Fig. 5 for the packet format.)

The work was done on Xerox Alto computers, a microcoded 16-bit machine with an internal cycle time of 180 nanoseconds, a writeable control store, and a memory bandwidth of 29 megabits/second. The Altos [28], [31] are interconnected with a 2.94 megabit Ethernet. Though the Ethernet is somewhat slower than the networks with which we are primarily concerned and cannot be extended to work at high speeds with the short packets that are used, it is a satisfactory experimental vehicle.

The Alto's macroinstruction set as well as its peripheral device controllers are implemented on the micromachine through the operation of up to 16 microcoded tasks, each executing 32-bit microinstructions. Mechanisms exist to switch between tasks in one microcycle. Because I/O device controllers—including the task that controls the Ethernet—are implemented as microcoded tasks, they can use the full processing capability and

Fig. 5. ESP Packet Format.



temporary storage of the micromachine, and can access main memory easily.

The emulator task microcode that was used provides a macroinstruction set similar to a Data General Nova [7] and executes macroinstructions at about 330 KIPS. No protection or virtual memory facilities are implemented on the Alto.

5.1. Implementation—Software Version

We first implemented a software package that provides five subroutines that implement the RLDA, RSTA, RCS, RENQUEUE, and RDEQUEUE references. These subroutines cause a request packet to be transmitted to a remote Alto and return control when a proper response packet is received or when an error condition is detected. On the remote site, a slave process executes the desired operations and returns an appropriate response.

The software is written entirely in BCPL [6] and uses the raw datagram facilities of PUP Level 0 for packet transport [3]. Sessions are maintained between each pair of communicating processors. Duplicate elimination is handled by the sequence number field of the ESP packet.

5.2. 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 slave that executes and responds to ESP request packets; the other allows an Alto to act as a master and issue 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-1, 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 queueing 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, microcoded instruction decoding is performed to make the references more efficient. Finally, substantial performance benefits are realized by overlapping memory accesses with processing.

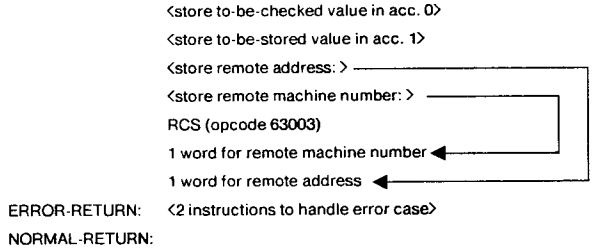
The microcode is simple due to more convenient handling of errors, multitasking, and timeouts in the micromachine. It comprises about 280 instructions though this number could be reduced by more clever microcoding. A total of 7 hardware registers are used in processing. The microcode executing on the slave uses an additional 728 (256 × 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 Sec. 3.1.

Use of the new instructions is illustrated by the description of RCS in Fig. 6. 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.

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 on the Ethernet 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.

Fig. 6. Instruction Call Sequence for RCS.



When a response packet is received, its source and sequence number are checked to ensure that it is a response for 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.

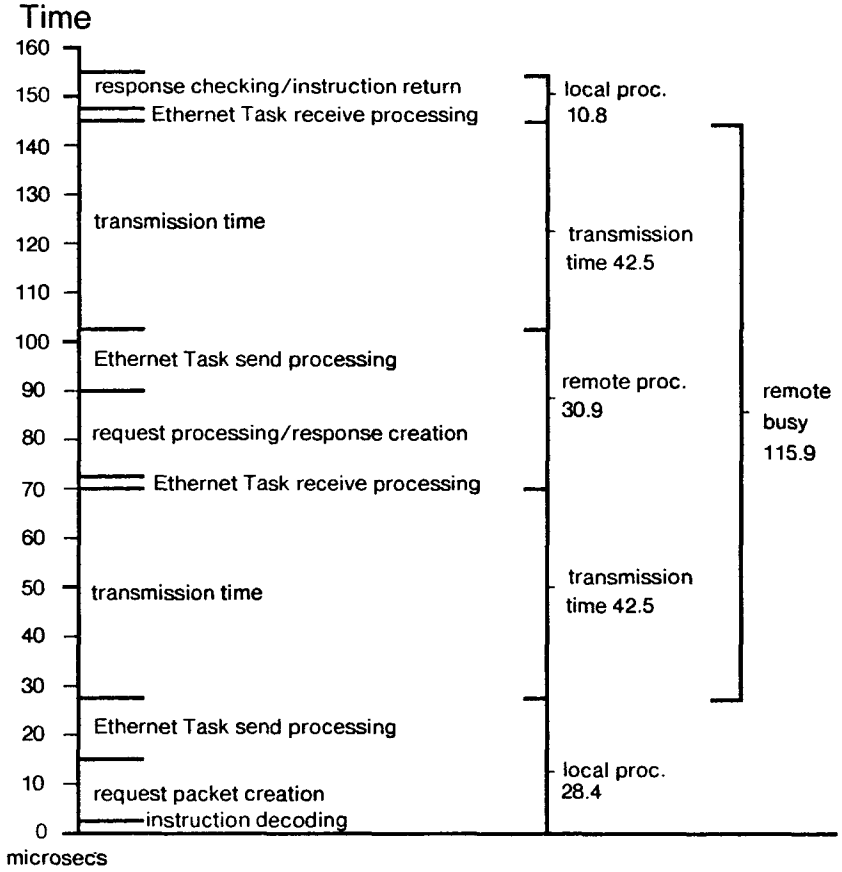
At the remote site, the microcode continually checks the Ethernet to see if a new packet has arrived. If an ESP packet arrives, the source byte is used to index into the socket table.If the sequence number of the received packet is the same as that in the corresponding table entry, the request is a duplicate 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. Up to 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.

5.3. Performance

In the software version, approximately 210 remote references can be executed per second on an unloaded Ethernet; this corresponds to 4.8 milliseconds/reference or about 1500 macro instruction times. Running at maximum speed, two machines communicating using this software package can impose a 1.8 percent load on the Ethernet. Using RSTA instructions, this corresponds to a 3.4 kilobit effective transfer rate. This software implementation is likely to be at least three times faster than any implementation using the PUP byte system protocol—a protocol that provides a full duplex, reliable byte stream protocol from one machine to another. This difference is due to the more complex protocol used by the reliable byte stream protocol and the more general interfaces that it provides.

The microcode version is capable of supporting 6450 references per second corresponding to a time of 155 microseconds/reference. As another characterization, each remote reference takes about 50 macro instruction times. Figure 7 shows a breakdown of the time spent when an RLDA instruction is executed, assuming no contention on the Ethernet. This time is representative of the times of the other instructions as well. Of the 155 microseconds required, transmission time accounts for more than half: 85 microseconds. Local processing leading up to the request requires 28 microseconds, processing at the remote site requires 31 microseconds, and local processing after the response is received requires 11 microseconds.

Fig. 7. Breakdown of Time Spent in RLDA Instruction, Microcoded Version.



The slave's Ethernet transceiver or processor is busy for 116 microseconds per request. Thus, a shared memory could support a maximum of 8600 references per second. A processor could initiate 6450 remote instructions per second, placing a load on the Ethernet of about 55 percent. 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 percent 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 provide high data throughput, the effective transmission data rate in this test was 80 kilobits. Table III summarizes these yardsticks.

If a processor executed an instruction stream that contained 1 percent remote references, a processor would slow from executing about 330,000 instructions/second to about 220,000 instructions/second, a 33 percent speed degradation. This slowing has implications with respect to the types of distributed programs that could be supported.

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 high load on the memory coupled with the fact that a slave does not listen to the Ethernet while it is processing a request results in many request retransmissions—a type of thrashing. The occurrence of this problem demonstrates that overuse of a resource in this environment is quite harmful.

Finally, in a test to determine if the Ethernet is a limiting factor with two outstanding sessions, two machines made requests to two separate shared memories and collectively put a load on the network of about 64 percent of 1.92 megabits. There were very few collisions or retransmissions but transmission times were longer due to the possibility of having to wait for a packet to pass. The longer transmission times reduced the number of references each machine could generate to about 3750 per second or about 1250 per second less than when only a single session was in use.

Table III. Performance Summary. The parenthesized measurements were achieved in a BCPL program and include iteration overhead.

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

5.4 Case Study Results

The case study demonstrated that relatively simple **only-once-type-1, primary, processor-synchronous** references can be implemented efficiently. The performance results show that the microcoded implementation executes about 30 times faster than the software implementation, and probably an additional 3 times faster than a software version built upon a general purpose byte stream protocol. The performance improvement is not surprising given the task switching, queueing, and subroutine calls in the software implementation. What is more surprising and important is the ease with which the direct microcoded implementation could be done.

The timings of the microcoded version show that **processor-synchronous** references are sometimes useful. Some remote references (particularly those that require the transmission of only a few data words) can be performed faster than the time to do a few task switches. Except in rare cases where errors or transmission delays are encountered, the simplification resulting from the **processor-synchronous** attribute can be beneficial.

The good performance of the microcoded version depends heavily upon the rapid decoding of the remote reference and the low process switch time to the communication process. The microtask organization of the Alto hardware facilitated both of these. The performance of the microcoded version would be much improved by the substitution of a 10 megabit Ethernet and slightly different hardware. Certainly, remote reference times could be well under 100 microseconds with these changes.

One point that the case study was *not* trying to make concerns the utility of unprotected shared memory in distributed systems. Where reliability is desired, direct memory reads and writes are potentially dangerous. The enqueue and dequeue instructions are more likely to foster reliability. Additional work on references like those described above must include consideration of protection and virtual memory.

6. Architectural Considerations

Two questions arise from the earlier sections of this work. The first concerns the proper hardware configuration for supporting very efficient implementations of remote references. Particularly, the network controller and underlying type of network are affected. The second involves the class of distributed programs that might be supported on such systems.

With respect to the local network controller, it must be closely integrated with the processor if fine granularity communication is to be supported. The communication process, as defined in the model, can be multiplexed on the processor or implemented on the network controller. However, in either case, process switches to the communication process must be inexpensive and **primary** operations must be executed rapidly. These restrictions require that the communication process must have fast access to main processor memory. In addition, low latency, high bandwidth stable storage is necessary for efficient implementations of **only-once-type-2** references. To lower contention problems on a slave, the network controller should allow reception of back-to-back packets.

The Cm* Kmap [27] is similar in function to a local network controller that can support the communication process. In fact, the Kmap has all the necessary properties (e.g., fast access to processor memory and small process switch times) except the ability to use stable storage. Both Cm* operating systems, StarOS [14] and Medusa [20], use the horizontal microcode executed by the Kmap in much the way that we would have the communication subsystem use the facilities of the local network controller.

With respect to the local network, the reduction in communication processing overhead may lead to increased bandwidth requirements. In addition, if fine granularity communication is important, the network must support the use of small packet lengths to ensure that fine granularity communication does not incur long delays. This combination of small packet sizes and high bandwidth requirements (say, 100 megabits) would probably not allow contention networks like the Ethernet to be used. Ring networks like those of Cambridge [29], TRW [2], or MIT [25] are better suited to high speed, small packet size requirements. We discuss these network-related issues in more detail in [26].

Turning to the implications of more efficient communication, typical applications for local network-based systems such as mail servers and replicated file systems could be more efficiently implemented. It is also possible to consider using such a local network-based architecture for supporting distributed programs of the type that might be executed on shared memory multiprocessors. However, assuming a matched technology (e.g., fifty cached, 1 MIPS, 32-bit processing nodes, two 100 megabit/second ring networks) and about 50 percent network utilization, shared memory access would probably be about two orders of magnitude more costly than local memory accesses. While this is much better than the four orders of magnitude that might be found in traditional communication systems, the few hundred nanoseconds required for a cached reference is difficult to approach with a bit-serial communication link crossing a few hundred meters. Hence, unless multiprocessor algorithms make less than 1 percent of their memory accesses globally, this architecture could not support them.

On the other hand, this architecture would be suitable for multiprocessor algorithms that use somewhat less finely granular communication. If the communication mix were to include some larger block transfers, high data communication rates between processors could be sustained. Thus, an architecture based upon efficient communication on a high speed local network seems to fit somewhere between a shared memory multiprocessor

and a conventional network-based system. In some ways, the resulting architecture is similar to the Tandem NonStop System [15].

One other issue affecting distributed programs concerns the utility of **only-once-type-2** references. Though the availability of low latency stable storage could make implementations of **only-once-type-2** references and their corresponding remote operations quite efficient, the fact that the references contain the transaction commit might result in decreased flexibility. A transaction commit that covers the work performed by multiple references is an alternative approach. More work on transaction-based distributed systems will be necessary to resolve this point.

7. Summary

We presented a communication model that includes a taxonomy of communication instructions, called remote references. The specialization of the references permits efficient implementations, and the semantic attributes of the references make them a good basis for the construction of distributed programs. Though relatively complete, the taxonomy could be extended to include more complex types of sessions, such as those with multiple slaves, and to include other semantic attributes such as intra-reference flow control for references having very variable size requests and responses.

Following the discussion of the model, we presented issues that arise when implementing a communication subsystem based upon the model. The experimental communication subsystem for the Altos shows that a streamlined system based upon the model can be implemented with great efficiency.

To demonstrate further the utility of the remote reference/remote operation model, a reasonable subset of communication primitives must be selected and a complete communication subsystem designed. Not all references can be implemented as efficiently as those in the example. But without doubt, they could be implemented much more efficiently than the normal communication mechanisms that are currently used on local networks.

Acknowledgments. 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 Gifford, John Hennessy, Cynthia Hibbard, Bruce Lindsay, Patricia Selinger, and the referees for valuable suggestions and comments.

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 are done atomically. The expression referred to as MACHINE-NUMBER[ADDRESS] refers to absolute memory address ADDRESS on the processor referred to by MACHINE-NUMBER.

Received 9/81; revised 11/81; accepted 12/81

References

1. Barlett, Joel F. A nonStop™ kernel. *Proc. 8th Symp. on Operating System Principles,* Dec. 1981, 22–29.
2. Blauman, Sheldon. Labeled slot multiplexing: a technique for a high speed fiber optic based loop network. *Proc. 4th Berkeley Conference on Distributed Data Manipulation and Computer Networks,* (Aug 1979), 309–321.
3. Boggs, David R., Shoch, John F., Taft, Edward A. and Metcalfe, Robert M. *Pup: an internetwork architecture.* Report CSL-79-10, Xerox Palo Alto Research Center, 1979.
4. Cook, R.P. * MOD—a language for distributed programming. *IEEE Trans. on Software Engineering SE6,* 6 (Nov. 1980), 563–571.
5. Cypser, R.J. *Communications Architectures for Distributed Systems.* Addison-Wesley, Reading, MA, 1978.
6. Curry, James E. et al. *BCPL Reference Manual.* Xerox Palo Alto Research Center, 1979.
7. *Introduction to Programming The Nova Computers.* 093-000067, Data General Corp., Southboro, MA, 1972.
8. Folts, Harold C. Coming of age: a long-awaited standard for heterogeneous networks. *Data Communications.* (Jan. 1981).
9. Gray, J., McJones, P., Blasgen M., Lindsay, B., Lorie, R., Price T., Potzulo, F., and Traiger, I. The recovery manager of a database management system. *Computing Surveys 13,* 2 (June 1981), 223–242.
10. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 12,* 8 (Aug. 1978), 666–677.
11. *IBM System 370 Principles of Operation.* GA22-7000-5, IBM Corporation, Poughkeepsie, 1976.
12. *DOD Standard Transmission Control Protocol.* Report RFC-761, Information Sciences Institute, Marina del Ray, 1980.
13. Ikeda, K., Ebihara, Y., Ishizaka, M., Fujima, T., Nakamura, T., and Kazuhiko, N. Computer network coupled by 100 MBPS optical fiber ring bus—system planning and ring bus subsystem description. *Proc. Compcon,* Nov. 1980, 159–165.
14. Jones, A.K., Chansler, R.J., Durham, I., Schwans, K., and Vegdahl, S.R. Staros, a multiprocessor operating system for the support of task forces. *Proc. 7th Symp. on Operating System Principles.* Dec. 1979, 117–127.
15. Katzman, J.A. A fault tolerant computing system. *11th Hawaii Int. Conf. on System Sciences.* (Jan. 1978); Also appears in Siewiorek, O., Bell, G., and Newell, A.: *Computer Structures: Principles and Examples.* McGraw-Hill, New York, 1981.

16. Lampson, B. and Sturgis, H.K. *Crash Recovery in a Distributed System.* (unpublished), Xerox Palo Alto Research Center, 1979.
17. Liskov, Barbara. *Linguistic support for distributed programs: a status report.* Laboratory for Computer Science Computation Structures Group Memo 201, MIT, Cambridge, 1980.
18. Metcalfe, R.M., and Boggs, D.R. Ethernet: distributed packet switching for local computer networks. *Comm. ACM 19*, 7 (July 1976) 395–404.
19. Nelson, Bruce Jay. *Remote Procedure Call.* Ph.D. Dissertation, Report CMU-CS-81-119, Carnegie-Mellon University, Pittsburgh, PA, 1981.
20. Ousterhout, John K., Scelza, Donald, A., and Sindhu, Pradeep. Medusa: an experiment in distributed operating system structure. *Comm. ACM 23*, 2 (Feb. 1980), 92–105.
21. Peterson, James L. Notes on a workshop on distributed computing. *Operating Systems Review 13*, 3 (July 1979), 18–27.
22. Popek, G., et al. Locus: A network transparent, high reliability distributed system. *Proc. 8th Symp. on Operating System Principles*, Dec. 1981, 169–177.
23. Rawson, E.G., and Metcalfe R.M. Fibernet: multimode optical fibers for local computer networks. *IEEE Trans. on Computer Communication COM-26*, 7 (July 1978), 983–990.
24. Saltzer, J.H. End-to-end arguments in system design. *Proc. 2nd Int. Conf. on Operating Systems.* Paris (April 1981).
25. Saltzer, J.H., Clark, D., and Reed, D. *Version Two Ring Network.* Laboratory for Computer Science Report, MIT, Cambridge, 1981.
26. Spector, Alfred Z. *Multiprocessing Architectures for Local Computer Networks.* Ph.D. Dissertation, Report STAN-CS-81-874, Stanford University, 1981.
27. Swan, R.J., Fuller, S.H., and Siewiorek, D.P. Cm* A modular multi-microprocessor. *Proc. of the National Computer Conference.* June 1977, 636–644.
28. Thacker, C.P., McCreight, E.M., Lampson B.W., Sproull, R.F., and Boggs, D.R. Alto: A personal computer. In Siewiorek, O., Bell, G., and Newell, A. *Computer Structures: Readings and Examples.* Second ed. McGraw Hill, New York, 1981.
29. Wilkes, M.V., and Wheeler, D.J. The Cambridge digital communication ring. *Proc. Local Area Communication Network Symposium.* Boston, May 1979.
30. *ALTO: A Personal Computer System Hardware Manual.* Xerox Palo Alto Research Center, 1979.
31. Zimmerman, H. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Trans. on Communication COM-28*, 4 (Apr. 1980), 425–432.

Operating Systems
Anita K. Jones
Editor

Grapevine: An Exercise in Distributed Computing

Andrew D. Birrell, Roy Levin,
Roger M. Needham, and Michael D. Schroeder

Xerox Palo Alto Research Center

Grapevine is a multicomputer system on the Xerox research internet. It provides facilities for the delivery of digital messages such as computer mail; for naming people, machines, and services; for authenticating people and machines; and for locating services on the internet. This paper has two goals: to describe the system itself and to serve as a case study of a real application of distributed computing. Part I describes the set of services provided by Grapevine and how its data and function are divided among computers on the internet. Part II presents in more detail selected aspects of Grapevine that illustrate novel facilities or implementation techniques, or that provide insight into the structure of a distributed system. Part III summarizes the current state of the system and the lessons learned from it so far.

CR Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications, distributed databases*; C.4 [**Performance of Systems**]—*reliability, availability and serviceability*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; H.2.4 [**Database Management**]: Systems—*distributed systems*; H.2.7 [**Database Management**]: Database Administration; H.4.3 [**Information Systems Applications**]: Communications Applications—*electronic mail*

General Terms: Design, Experimentation, Reliability

Part I. Description of Grapevine

1. Introduction

Grapevine is a system that provides message delivery, resource location, authentication, and access control ser-

Authors' Present Addresses: Andrew D. Birrell, Roy Levin, and Michael D. Schroeder, Xerox Palo Alto Research Center, Computer Science Laboratory, 3333 Coyote Hill Road, Palo Alto, CA 94304; Roger M. Needham, University of Cambridge Computer Laboratory, Corn Exchange Street, Cambridge, CB2 3QG, United Kingdom.