NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-86-165

Communication Support in Operating Systems for Distributed Transactions

Alfred Z. Spector

November 1986

Abstract

This paper describes the communication functions required for distributed transaction processing. The paper begins with a discussion of models that illustrate how a communication subsystem fits into a proposed system architecture. Then, it describes the system and user activities that depend on the communication subsystem. Finally, it uses these activities to motivate the facilities that should be provided by a communication subsystem that supports transaction processing.

Technical Report CMU-CS-86-165

Copyright © 1986

This work was supported by IBM and the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

1. Introduction

Communication subsystems permit their client programs to invoke operations on remote sites and to perform auxiliary control tasks. For example, an application-level client may send a SQL Select operation and associated data to a remote database, which then returns the data that matches the selection criteria. A client system facility may piggyback time information on application-level messages to maintain a consistent global notion of time. The exact nature of the applications and system control facilities using a communication subsystem substantially influence the functions it must implement.

The functions may be complex if they must support *distributed transaction processing*, that is, the execution of transactions on data stored in multiple partitioned and replicated databases on various network nodes. Distributed transaction processing provides applications with access to shared data that are stored with high data integrity and availability. Such applications require the usual communication functions such as datagram, file transfer, data streaming, network virtual terminal, and RPC. They may also require more unusual facilities to support support data replication, atomic commitment, a coherent system-wide notion of time, and authenticated protected data access. In addition, the volume and frequency of communication performed by distributed applications may require high bandwidth, low-latency communication if users' response time and throughput requirements are to be met.

Because of such complexity, this paper takes the view that a communication subsystem for supporting distributed transactions must be designed to fit within the complete framework of processing that will occur. With this viewpoint as a basis, this paper first describes generic system, computation and architecture models. Only then does it describe the communication facilities that are required. While this approach has the disadvantage of limiting the discussion of communication to that required for one particular set of models, it shows more clearly how a communication subsystem is *integrated* into the greater system — the major point of the document. Furthermore, the system, computation, and architectural models are general enough to apply to many real systems.

Following the discussion of models, Section 3 discusses the system activities that require communication support. Section 4 uses these activities as a basis to describe important communication primitives and implementation strategies. These primitives are motivated by the Camelot distributed transaction processing system, developed at the Carnegie Mellon Computer Science Department [Spector et al. 86]. The paper concludes with Section 5, which is a brief summary.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

2. Three Models

There is substantial agreement on the underlying *system model* for distributed processing. The model has processing nodes and communication networks, as illustrated in Figure 2-1. Processing nodes are fail-fast and may be either uniprocessors or shared memory multiprocessors. In general, there are many types of processing nodes on the same distributed system. Processing nodes are assumed to have independent failure modes.

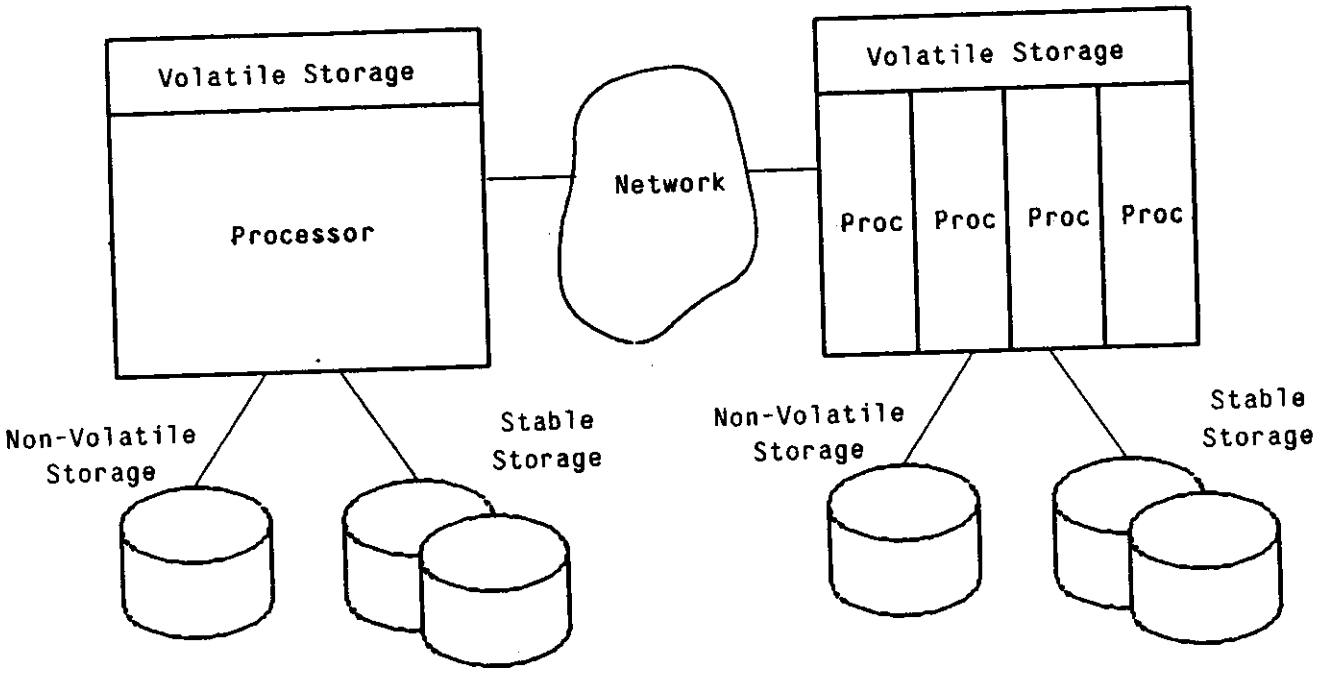

Figure 2-1: Hardware Model

This figure shows the components of the hardware model. Stable storage and non-volatile storage do not necessarily have to be implemented on disks.

Storage on processing nodes comprises volatile storage — where portions of objects reside when they are being accessed, non-volatile storage — where objects reside when they have not been accessed recently, and stable storage — memory that is assumed to retain information despite failures. The contents of volatile storage are lost after a system crash, and the contents of non-volatile storage are lost with lower frequency, but always in a detectable way. Stable storage can be

implemented using two non-volatile storage units on a node, or using a network service [Daniels et al. 86].

The system model's communication network provides datagram-oriented, internetworked OSI Level 3 functions [Zimmermann 82] such as the Arpanet IP protocol [Postel 82]. In other words, the network comprises both long-haul and local components and permits processes to send datagrams having a fixed maximum size. Some local area networks may specially support multicast or broadcast, and the network protocols are assumed to support these features for reasons of efficiency. Because applications using the system may need high availability, communication networks should have sufficient redundancy to render network partitions unlikely. However, network partitions can nonetheless occur, so higher levels of the system must take measures to protect themselves against the erroneous computations or inconsistencies that could result.

The *computation model* comprises applications that perform processing by executing transactions performing operations on data objects. Data objects are distributed across the network and are encapsulated within protection domains that (1) export only operations that make-up the defined interface and (2) guarantee that the invoker has sufficient access rights. Data objects may be nested. This model applies to many systems, including R*, Argus, TABS, and Camelot [Lindsay et al. 84, Liskov and Scheifler 83, Spector et al. 85, Spector et al 86, Spector et al. 86].

The model further defines transactions as encapsulation units that provide three properties [Gray 80]: *Synchronization* properties, such as serializability, guarantee that concurrent readers and writers of data do not interfere with each other. *Failure atomicity* simplifies the maintenance of invariants on data by ensuring that updates are not partially done. For example, failure atomicity guarantees that a transaction that updates two distributed copies of a replicated file will either succeed and modify both, or fail and modify neither. *Permanence* ensures that only catastrophic failures in stable storage will corrupt or erase previously made updates. Transactions can be nested to reduce the likelihood that they need to abort completely and to provide protection from concurrently executing processes within a transaction.

The *architectural model* describes how processing on a node is organized; that is, it describes how to realize the computation model on the system model. It is structured in five logical levels, as shown in Figure 2-2. As one might hope, few calls proceed from lower levels to upper levels. (The levels referred to in this model are distinct from the OSI levels, and subsume functions in OSI levels 4 to 7.)

At the base in Level 1 is the operating system kernel that implements processes, local synchronization, and local communication. Examples are the V, Accent, and Mach kernels [Cheriton 84, Rashid and Robertson 81, Accetta et al. 86], though V and Mach also include inter-node

communication facilities which this model includes in Level 2, the communication level. This level is the subject of this paper and the following sections analyze it in more detail.

Above the communication level is the distributed transaction facility (DTF), Level 3. Although there is room for diversity in its functions, the DTF must make it easy to initiate and commit transactions, to call operations on objects from within transactions, and to implement abstract types that have correct synchronization and recovery properties. For example, it is this level that implements commit protocols, stable storage, recovery, and deadlock detection. The DTF uses the process, synchronization and local and inter-node communication facilities of the kernel and communication levels for its own needs and exports them to higher levels as well.

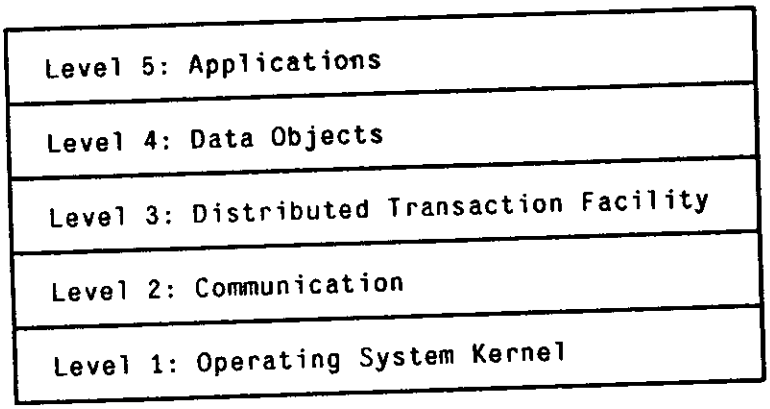

Figure 2-2: Five Level Architecture Model

This figure illustrates the five system levels. The kernel level provides processes and inter-process communication. The communication level provides inter-node communication. The distributed transaction facility provides complete support for transaction processing on distributed objects. Data objects are maintained in Level 4. The applications that use them are in Level 5.

Implementors of abstract data objects, such as database managers, use the DTF to construct objects that can be used by clients within transactions. Objects may be grouped into a subsystem, and there may be multiple subsystems in the Data Object Level (Level 4). These subsystems are called Resource Managers in R*, Guardians in Argus, and Data Servers in TABS and Camelot. On a

distributed system, subsystems are frequently called *servers* and invoked via a request; the user of a subsystem is frequently called a *client*. Frequently, servers send a response to clients to return a result. In the common case that a client calls a server on the same node, protected procedure calls may be substituted for messages to reduce invocation overhead.

In Level 5, applications use the DTF to begin, commit, and abort transactions and to execute operations on objects. Example applications include a banking terminal system and an interactive interface to a database manager.

This architecture provides two benefits over traditional architectures that blur the distinction between Levels 3, 4, and 5: First, because many of the components that support transactions are standardized and moved lower into the system hierarchy, there is the potential to implement them more efficiently. Second, the architecture provides a common notion of transaction and data object for all objects and applications in the system, and permits more uniform access to data. This permits an application, for example, to update transactionally a relational database containing indexing information, a file containing image data, and a hierarchical database containing performance records. All the system components also use standardized facilities for performing remote accesses, for transaction commitment, etc.

Having characterized the computational activities required for distributed transaction processing, it is now possible to examine the activities that use the communication subsystem. We can then turn our attention to requirements of the communication subsystem and how to meet them.

3. Activities Requiring Communication Support

Distributed transaction facilities require communication for data objects and applications in Levels 4 and 5 and system activities in Levels 1 through 3. This section lists those activities and then describes a set of communication subsystem functions that will support them.

3.1. Communication-related Activities of Data Objects and Applications

Before data objects and applications can begin to access other data objects, they must first establish a communication path to them. A *name service* locates servers that encapsulate objects and returns lower-level names that can be used to establish communication. To support distributed replication algorithms, a single object name may be associated with a several copies of objects, each stored on a different node. Typically, replication techniques specify the number of copies of an object to which they require access.

The name service must manage the name space to prevent unintended name duplication and to

ensure appropriate authorization for name insertion and deletion operations. This is clearly a distributed data management problem, which is best solved by a collection of trusted Level 4 objects that can use the DTF. Thus, while the name service is logically related to communication services, it need not be implemented within the communication subsystem. However, the communication subsystem must provide the name service with well-known connections through which name servers can communicate with each other.

Questions arise concerning the permanence of name mappings, the granularity of objects that are named, and the management of the name space. There are many feasible answers to these questions, but here are some reasonable ones:

- Name mappings are relatively useless for objects that are inaccessible; when an object is unusable, it usually does not help to know its location. Hence, the motivation for replicating name mappings on another node is to reduce communication, not to provide availability.

- Objects registered in the system-wide name service should be coarsely grained; e.g., a database name rather than the names of all its relational tables. More detailed name resolution can be performed in an object-specific fashion. This decision is almost a necessity, both to reduce the number of communication paths to a server and to obviate the need for a uniform name space for each data object in an entire distributed system.

- Name mappings should survive node crashes to reduce the amount of work required to restart a node.

Once the name service has located an object, the lower-level object name can be passed to the communication subsystem and a session created. Clients and servers require sessions betwen them for many reasons:

- **Authentication and protection.** If clients are to be certain they are accessing a particular server and servers are to check the access rights of a caller, then the communication session must be authenticated in some way, possibly using encryption techniques [Needham and Schroeder 78]. Prevention of active and passive attacks on the communication channel is also a desirable service for many applications [Sansom et al. 86, Birrell and Nelson 84].

- **Flow-control and pipelining.** Large amounts of data may be passed to and from objects and require flow-control. For example, a request to a remote object could result in a response containing megabytes of data. Pipelining may be useful on networks having long latencies. Even on local area networks, the increasing use of networks interconnected by bridges or gateways tends to increase delays and the consequent need for pipelining.

- **Crash detection.** There must be a mechanism for determining if a server has crashed after its first use and prior to commit. Timing out while awaiting a response from an object is one crash detection technique, but a session failure provides more uniform and timely information for most errors. For example, sessions can detect most crashes even when a client is not calling its server.

Communication on a session usually takes the form of a (synchronous) remote procedure call having *at-most-once* semantics. While there is room for diversity in the definition of these semantics, all definitions guarantee that an operation on a server will be performed at most one time, despite network failures and retransmissions. Providing higher service levels in the communication subsystem (atomicity, or exactly-once semantics) is unneeded because the DTF can completely abort arbitrary units of work, which may then be retried. As mentioned above, requests and responses may have unlimited lengths so intra-message flow control may be needed.

Sometimes, more general forms of remote procedure call may be useful [Spector 82]. Asynchronous RPC's permit a client to continue processing and to receive a signal when a response is returned. Multicast RPC's issue a request to multiple servers. A multicast RPC primitive may await all responses before returning or it may signal the client as each response arrives. The latter organization is useful when a client invokes an operation on multiple servers but does not need all responses before continuing work. Multicast RPC's may be implemented on multiple sessions, or may use a single session having multiple destinations. The latter is required if low-level network multicast primitives are to be used.

3.2. Communication-related Activities of System Levels

The data objects and applications of Levels 4 and 5 require communication primitives that provide very general functions: support for arbitrarily long messages, authentication, and the like. In Levels 1, 2, and 3, communication is more constrained and there is more a-priori knowledge of message contents. For example, knowledge that a message usually fits within a network packet permits a simpler transmission protocol to be used. Similarly, some data can be piggy-backed on messages sent by Levels 4 and 5.

Communication services required by Levels 1, 2, and 3 must support at least five different functions: time services, commit processing, deadlock detection, certain higher-level communication services themselves, and failure inducement for testing purposes.

Logical time services, as provided by a Lamport distributed clock [Lamport 78], order events in a distributed system. All observable dependencies between events are reflected in time values provided by the clock; that is, if Server 1 observes the time as A and it sends a message to Server 2, and then Server 2 receives the message, Server 2 will then observe the time as B, with B > A. Such a mechanism is useful for various types of synchronization, for example, for supporting hybrid atomicity [Herlihy 85]. The underlying algorithm makes use of a counter on each node and a field included in each inter-node message that may update the counter.

A distributed real time service that is synchronized across nodes supports synchronization algorithms and performance measurement techniques. Many implementations of such mechanisms require periodic exchange of time information, which is done by appending information to existing message traffic and sending short messages during idle periods.

To perform atomic commit processing, the DTF must send control messages such as **Prepare-to-Commit**, **Prepare-Ack**, **Commit** and **Commit-Ack** [Lindsay et al. 79]. Some of these messages are typically sent to one or more of the nodes involved in a transaction. Regardless of protocol, the communication subsystem should maintain appropriate information on the nodes involved in the transaction, and control messages should be sent with low-overhead. Usually, they can be sent as network datagrams because messages are short and reliable transmission is not needed; the transaction manager must deal with node crashes anyway. Even though data encryption and authentication may be needed for Level 4 and 5 communication, control messages are difficult to forge and they contain so little data that is valuable to outsiders that there may be no reason to encrypt them. However, certain commit protocols can benefit from transmission to a multicast address that is incrementally developed as the transaction executes.

A DTF that supports nested transactions also requires a lock-resolution protocol in addition to the commit protocols. This protocol is invoked to determine if a nested transaction can inherit a lock from a relative in the tree of transactions. Depending upon the frequency of lock inheritance, this protocol may be invoked often and require high performance.

Distributed deadlock detection algorithms typically require piecing together enough of the distributed "wait-for" graph to break cycles. This requires the periodic transmission or the piggy-backing of information on other messages.

The communication facilities themselves require communication in addition to the usual demands for session establishment and the transmission and acknowledgment of user-supplied data. For example, control messages are sent by authentication servers as part of session creation. "Are you there messages" may be periodically sent on sessions to rapidly detect server or node crashes.

Finally, to aid in *reliability* testing, the communication subsystem should enable users to test the system under conditions of communication failures: lost, duplicate, and corrupted packets; partitions; and delays. Being able to simulate these conditions is an important feature. Also, facilities for monitoring the performance of the communication subsystem are useful. Methodical, empirical testing is needed to develop robust systems.

4. Communication Subsystem Functions and Implementation

This section lists a plausible set of functions that a communication subsystem should provide, given the requirements described in the previous section. It also describes the broad outlines of an implementation strategy for them. This ideas are loosely based on our design of TABS and Camelot with additions from other systems where needed.

4.1. Name Service

The name service should provide primitives to associate a name with one or more servers that implement the named object. It may also associate a lower-level name used by a server to distinguish between the multiple objects it implements. The name service also provides primitives to lookup and delete names. The lookup primitive should permit the caller to specify how many servers should be returned and to set a timeout interval after which control will be returned. While the name service does not need to be part of the communication subsystem, it is closely related and worthwhile to include in this section.

One implementation strategy is to have multiple name servers on the network that communicate with each other. Because of the desirability of storing name bindings permanently (so as to not have to register objects after a crash), the name service should be implemented as transactional (Level 4) servers, which can utilize stable storage. The DTF's services also simplify the consistency management of the name space. For example, new names can be added within a transaction. Locally storing recently used name bindings (hints) reduces the amount of inter-node communication, provided that applications are willing to detect and handle potentially out-of-date information.

4.2. Session-based Communication

Sessions should be the basis for communication between Level 4 and 5 entities. They are also appropriate for some of the communication between system-level entities. At minimum, sessions should support an efficient implementation of RPC with "are you there" messages to detect crashes. However, a session's required functions and implementation (including the amount of state that must be maintained) varies with the requirements of the DTF and the structure of the underlying system. For example, sessions supporting multicast RPC should have multiple recipients to take advantage of low-level multicast facilities. Differing needs for asynchronous RPC's, protection, authentication, conversion of heterogeneous data, and arbitrary internetworking also influence the functions and implementation of sessions.

There are at least two facilities for supporting a DTF that a communication subsystem can perform: It can record the participants in a transaction by watching the messages and the transaction

identifiers contained in them. This information is needed at commit time. Additionally, the communication subsystem can incrementally distribute a network multicast address to all the sites within a transaction so that network multicast can be used during the two-phase commit protocol. This multicast address can be related to the global transaction identifier and be piggy-backed on request messages. Cheriton describes a design for this in the V System [Cheriton 86].

4.3. Datagram-based Communication

The raison d'etre of datagram-based communication is to reduce transmission latency and CPU overhead. In order to keep datagram-based communication sufficiently lightweight, it is inevitably restricted in function: limited datagram sizes, lack of protection or authentication, etc. New functions that slow datagram transmission should be avoided.

Certainly, datagram communication should support unreliable point-to-point transmissions; also, it should support multicast, because many networks provide necessary hardware support. Both of these two services require little protocol layering. Possibly, there should be some datagram support that is tailored to operation on a single local area network recognizing that there are services that would not be used over a long-haul network. For example, a stable storage server (log) would almost certainly be on the same local area network as its client nodes [Daniels et al. 86].

4.4. Miscellaneous Features

There are a collection of miscellaneous features that a communication subsystem should support: a distributed (logical and/or real) time service, the parameterized insertion of errors or creation of network partitions, and a communication performance monitor. Other features may be needed for real-time applications or some high-availability architectures.

5. Summary

After examining the uses of a communication subsystem in a distributed transaction processing environment, it is not surprising to find that sessions and datagrams are the two most important facilities. However, in this environment where there is closely-coupled distributed processing, atomic commitment, replication, and a strong emphasis on reliable, highly available operation, there are some additional features that a communication subsystem should support. These include multicast, logical time, real time, performance evaluation, and fault insertion services. Higher level protocols not part of the communication subsystem but closely related to it are needed for commitment, nested transaction lock resolution, deadlock detection, and name resolution.

All these additional facilities necessarily require standardized interfaces and protocols to support open systems. In some instances, these facilities are being considered by standardization

committees. In others, they represent new protocols not yet under consideration. Further work on prototypes (e.g., Argus [Liskov 84], Camelot, ANSA [ANSA 86]) is needed to develop the necessary experience with them.

Acknowledgment

Thanks to Jeffrey Eppinger and who read and commented on this paper.

References

[Accetta et al. 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*. July, 1986.

[ANSA 86] *Functional Specification Manual (Release 1)* 1986.

[Birrell and Nelson 84] Andrew D. Birrell, Bruce J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems* 2(1):39-59, February, 1984.

[Cheriton 84] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software* 1(2):186-213, April, 1984.

[Cheriton 86] David R. Cheriton. Fault-tolerant Transaction Management in a Workstation Cluster. 1986.Stanford University.

[Daniels et al. 86] Dean S. Daniels, Alfred Z. Spector, Dean Thompson. *Distributed Logging for Transaction Processing*. Technical Report CMU-CS-86-106, Carnegie-Mellon University, June, 1986.

[Gray 80] James N. Gray. *A Transaction Model*. Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[Herlihy 85] Maurice P. Herlihy. *Availability vs. atomicity: concurrency control for replicated data*. Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7):558-565, July, 1978.

[Lindsay et al. 79] Bruce G. Lindsay, et al. *Notes on Distributed Databases*. Technical Report RJ2571, IBM Research Laboratory, San Jose, California, July, 1979. Also appears in Droffen and Poole (editors), *Distributed Databases*, Cambridge University Press, 1980.

[Lindsay et al. 84] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, Robert A. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems* 2(1):24-38, February, 1984.

[Liskov 84] Barbara Liskov. *Overview of the Argus Language and System*. Programming Methodology Group Memo 40, Massachusetts Institute of Technology Laboratory for Computer Science, February, 1984.

[Liskov and Scheifler 83] Barbara H. Liskov, Robert W. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* 5(3):381-404, July, 1983.

[Needham and Schroeder 78] Roger M. Needham, Michael D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM* 21(12):993-999, December, 1978. Also Xerox Research Report, CSL-78-4, Xerox Research Center, Palo Alto, CA.

[Postel 82] Jonathan B. Postel. Internetwork Protocol Approaches. In Paul E. Green, Jr. (editor), *Computer Network Architectures and Protocols*, chapter 18, pages 511-526.Plenum Press, 1982.

[Rashid and Robertson 81] Richard Rashid, George Robertson. Accent: A Communication Oriented Network Operating System Kernel. In *Proceedings of the Eighth Symposium on Operating System Principles*, pages 64-75. ACM, December, 1981.

[Sansom et al. 86] Robert D. Sansom, Daniel P. Julin and Richard F. Rashid. *Extending a Capability Based System into a Network Environment*. Technical Report CMU-CS-86-115, Carnegie Mellon, April, 1986. To appear in *SIGCOMM '86: Futures in Communications*, August 1986.

[Spector 82] Alfred Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM* 25(4):246-260, April, 1982.

[Spector et al 86] Alfred Z. Spector, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson. The Camelot Interface Specification. September, 1986. Camelot Working Memo 2.

[Spector et al. 85] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985. Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

[Spector et al. 86] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson. The Camelot Project. *Database Engineering* 9(4), December, 1986. Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.

[Zimmermann 82] Hubert Zimmermann. A Standard Network Model. In Paul E. Green, Jr. (editor), *Computer Network Architectures and Protocols*, chapter 2, pages 33-54.Plenum Press, 1982.