Jerry Saltzer 505

M.I.T. LABORATORY FOR COMPUTER SCIENCE
Computer Systems Research Division

December 13, 1979
Request for Comments No. 179

DESIGN OF DISTRIBUTED SYSTEMS SUPPORTING LOCAL AUTONOMY

by David D. Clark and Liba Svobodova

This is an invited paper that will be presented at COMCON Spring '80 in San Francisco, February 1980.

DESIGN OF DISTRIBUTED SYSTEMS SUPPORTING LOCAL AUTONOMY

by David D. Clark and Liba Svobodova

The subject of this paper is distributed systems in which the individual nodes, while cooperating in a standardized manner, maintain a fair degree of autonomy with respect to their management and internal organization. Such systems closely model the structure and operations of most real world organizations; we believe that they will be the most widely used form of distributed systems. A research project that includes development of a distributed computing system with these properties is currently underway at the Laboratory for Computer Science at M.I.T. [1].

Many advantages that have been claimed for distributed systems, such as improved reliability and incremental growth, are not easily achieved in practice. The feasibility of such goals and the manner in which they can be achieved depend on the fundamental assumptions about the purpose and function of the system. Our intention is to show how the requirement of autonomy influences the architecture of a distributed computing system and the resulting capabilities of the system.

Motivation

The first important step towards building a distributed system is to understand the real motivation for and the nature of the distribution. A

---

1. This research was sponsored by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

common opinion is that the main motivation is cost: one can get more computing cycles per dollar if one buys lots of small machines rather than one large machine, if one is willing to ignore the cost of glueing the small machines together so that the intended application can run on them. While this observation may prove true in certain cases, we feel that it is not the fundamental motivation for a distributed system. Rather, the fundamental issue is that computer systems should reflect the structure and needs of the problem to which they are being applied. The justification for distributed systems is that many applications are naturally distributed; it is the centralized computer facility that is often only the artifact of economic forces.

Consider the application of information management as practiced in business or government, the application often called office automation. Clearly, the non-computer version of office work is distributed. Employers do not force all their employees to share one large desk, or use one huge room full of filing cabinets. Yet, more important than the physical equipment is the management of stored information. The manner in which individuals chose to organize and manage the information under their control is largely up to them. By storing this information in their own desk or filing cabinet, they are assured that they can have access to it whenever they need it, that the information is being organized in a way which suits their needs and is being protected with the right level of security, and so on. We believe that once the acquisition cost of small computers fell sufficiently low, their proliferation has been substantially motivated by this sort of reasoning [2]. If a computer is allocated solely to one person, or to a collection of people with an identical responsibility, then the users of that computer can be assured that the machine will not be taken down for preventive maintenance at

the precise moment when they must produce a high priority report, that the information stored on the computer has been backed up to the degree required for their needs, that they will not be preempted from the machine due to high load at the wrong moment, and so on. Perhaps most fundamentally, they can have substantial assurance that nobody can arbitrarily interrogate the information in that computer.

Returning to the example of information stored within one's personal desk, it is clear that one of the functions of that information is to enable employees to answer questions related to some aspect of the organization, as required by their jobs. However, it is equally clear that a supervisor does not obtain answers to those questions by arbitrarily opening desks and searching through the information stored there. Generally, one's desk is considered one's private domain. It is each individual's job to interpret the stored information and provide the appropriate answers, in whatever manner he sees fit. This is a critical need, well understood outside the computer world, but often ignored in the architecture of traditional data base management systems controlled by a central administrator.

A collection of dedicated computers could be interconnected and integrated to form a distributed computing system with such properties. Since individual computers will have to direct a variety of requests to other computers in the course of performing their duties, the format and content of these requests and responses must be recognized and acknowledged if the system as a whole is to fulfill its global function. However, the manner in which each computer is programmed to respond to the appropriate requests should be under the control of the person responsible for managing the information used to generate the response. In concrete terms, this means that there should not be centralized data base administrator that determines the format for the

stored information and access to that information for all of the computers composing a distributed system. This duty should be potentially delegated to the individual managers of the various computers.

We use the word autonomy to characterize this ability of each node to administer its own resources: the processing and release of the local information, and the allocation of the local hardware resources. Such autonomy is not an attribute inherent to all distributed systems. Most proposed distributed systems strive to give the application programmers an image of a monolithic central system. Such systems are perhaps best characterized as loosely coupled multiprocessor systems (i.e., multiprocessor systems in which processors do not share memory). We do not deny the value of such systems, but we will not consider them in this paper.

Distributed System Architecture

We will now describe a distributed system architecture that follows directly from the philosophy presented in the preceding section. This architecture is intended to support a variety of applications, not just the previous example of office automation. For this reason, we will attempt to keep a strong distinction between the application and the operating system, which is application independent.

We postulate that a system designed in accordance with our philosophy will have three classes of components: the computing nodes, special servers, and a communication substrate. Our goal is to produce an operating system for the computing nodes that works with the communication substrate and the servers to provide a coherent environment for the creation of application software. We will argue that there are certain functions, often described as

part of a distributed system, that can only be provided as part of the application programs; the requirement of autonomy makes it unreasonable to provide these functions in the operating system.

In the particular research project we are now conducting, we are assuming that the computing nodes are dedicated to individuals, that they are "personal computers". Of course, such computers could be allocated to groups of people, rather than individuals, provided that those people have identical responsibilities and requirements, for example a group of clerks that perform an identical function. However, in this paper we shall assume that these machines are personal computers, since that is the most extreme, and therefore from our point of view the most interesting, representation of the system. The servers are specialized machines that will perform particular services for the community as a whole. Possible services include archival storage, specialized printing, specialized computing, and access to foreign computer and communication facilities. The third component is the communication substrate, which will permit the exchange of messages between the various personal computers and servers. The communication substrate may also include some special machines for directory maintenance, message spooling, and monitoring and diagnosis of the physical communication lines, but these "servers" are invisible to the application programmer. The system does not include a central facility for management of information as one of its principal components. Based on the arguments made earlier, information management is the responsibility of the personal computers. However, individual applications could support a common shared data base as a special server.

The most basic constraint that the assumption of autonomy places on an application built for our system is that each piece of information has one and

only one home, the computer of the person responsible for that information. This approach is in marked contrast to other distributed systems in which several nodes can share control of any particular piece of application data [3,4,5,6]. The schemes needed to accomplish this are very complicated, but above all, they are incompatible with our philosophy: they completely compromise the control which the owner of the information has over its release or modification. The assumption that information has only one home has a pleasing simplicity, but there are a number of issues that must be considered if a system based on this assumption is to be successful. In the following sections we will consider efficiency, reliability, transaction integrity, and expandability.

Efficiency

An often claimed advantage of distributed systems is that information can be brought close to the person requiring it, thus making his queries and updates proceed very rapidly. Has our fundamental assumption compromised our ability to achieve this goal? We believe not, for two reasons. First, the information one manipulates most often is the information one owns, and that by definition is in one's personal machine. Second, most of the information requests involving data not locally stored are not update requests but retrieval requests, and needed information can be locally stored to enhance the speed of retrieval requests with no compromise to our philosophical assumptions. However, if we do locally store information in order to speed our ability to query it, we must not model it as an exact copy of the remotely stored master version of the information. In fact, we have no idea what the remote representation is. All the local machine can know is the result of

attempts to query the remote information. Thus, the locally stored information must be viewed as a distinct entity, a different version. The application programs must always understand when they are manipulating the locally stored version and when they are manipulating the information itself.

Consider a simple example. Most people obtain from their bank a monthly statement that contains their checking account balance. This balance is then locally stored, and locally manipulated by the owner of the account to reflect checks cashed or deposits made. Every owner of a checking account understands that when one adds or subtracts to this local copy of the bank balance, that these operations do not manipulate the version of the balance maintained by the bank. In fact, it is critical that the local version is distinct because operations are often performed on it that cause it to have a value which intentionally differs from the balance stored by the bank. Additions and subtractions to the local version are usually made immediately, while the bank only modifies its version when the transaction has cleared at the bank. Thus, the two balances will often differ: the owner of the account has only loose control over this difference.

Returning to the computer world, we believe that this idea of a single master copy of the information itself, combined with distinct entities that are locally stored versions of the information, is fundamentally the correct way to manage the information in a distributed system. It will be necessary only infrequently to coordinate with the master copy, and almost all of the transactions performed by a user will involve versions stored at the personal machine. Thus, we believe that a system built according to our philosophy can be very efficient.

Reliability

Another claim often cited for distributed systems is that because there are several machines in the configuration, physically separated and therefore unlikely to crash simultaneously, the overall reliability of the system should be higher than in a single, centralized site where any single crash disables the entire system. The reliability claim is based on two observations. First, both functions and data can be replicated on independent hardware. Second, propagation of low level errors is restricted by physical separation of processes and resources. However, physical distribution and decentralized control present new reliability problems, and the requirement of autonomy restricts replication of data.

The notion of reliability is often simplified to mean accessibility of physical resources and information needed to perform the tasks of the system users. Such a simplification is incorrect and misleading. The most crucial reliability problem in an information processing system is the prevention of information loss. A distributed system offers a new mechanism for dealing with protection against information loss due to damage of a storage device: it is possible to use other nodes on the distributed system to store backup copies of important information, so that this information can be retrieved after a catastrophic crash of an individual node. This kind of replication of information in a distributed system should not be confused with the replication of information for the purpose of enhancing the speed of queries or allowing update of the information to be performed at several sites. In many distributed system proposals, these two distinct functions of information replication, efficiency enhancement and reliability enhancement, have become so tangled that it is difficult to convince oneself that the enhancement goals were indeed achieved. In particular, in the schemes where multiple copies are

used to enhance both efficiency and reliability simultaneously, these copies must be kept mutually consistent. To do this reliably in face of node failures and especially communication failures that may leave the system partitioned into subsystems that cannot communicate with each other, requires very complex protocols. The overhead of these protocols may be very high, thus thwarting the efficiency enhancement promised by the multiple copy scheme. Partitioning is a particularly unpleasant problem: copies of the replicated information may end up in different partitions, and if each partition continues updating the information, the copies will diverge. Making all the copies consistent again when the system is recombined is a hard problem. Although for some update patterns there may exist an inexpensive solution, the only general solution is to undo some of the updates. Thus at most one partition should be allowed to make updates. The nodes in the other partition must either wait, or resort to a scheme similar to that proposed in the section on efficiency, that is, view the information as a "local" version for that partition, a version that may be obsolete.

In contrast, in our system the remotely stored backup copies of information are also under the complete control of the home node. In fact, the information replicated for reliability might be stored in an encrypted form, so that it be usable only by the home node. Thus every piece of information has only one home where it can be updated, the computer of the person responsible for the information. This means that a failure of the personal computer or its detachment from the network makes this information unavailable to the rest of the system. We believe, however, that the proper solution is to ensure that the individual nodes are available both to their direct users and to the rest of the system. Thus the requirement for the nodes is that they should fail only very rarely and their time to repair

should be very short. The communication network should support multiple links between nodes so that the system remains fully connected even if some of the links fail. If partitions and failures of the communication network and failures of individual nodes can be limited in duration, then individual nodes can survive a period of inaccessibility of some information or even a complete isolation by manipulating only the locally stored versions, and remembering only the transactions that must ultimately be enacted when the master copy again becomes accessible.

The experience with the ARPANET indicates that it is indeed possible to build robust communication networks such that partitioning is very rare. For the individual nodes, the following strategy is suggested. If the system is indeed composed of a large number of small machines, it is quite reasonable to imagine having on hand enough physical spare machines that a failed machine is simply and quickly replaced by a new machine that is initialized where the old one left off by transfering the storage devices to it. However, for some special nodes, in particular some shared servers, it may be necessary to provide better assurance of continuous service. If a server maintains information that changes unpredictably and rapidly, replicating the server leads to the multiple copy problem discussed earlier. Thus, the appropriate approach, in our opinion, is to design such servers to be highly reliable, by replicating critical components internally. Examples of such highly reliable configurations can be found in [7,8]. However, for other types of servers, replication may be the right solution [9].

It may seem that we have eliminated most of the difficult "reliability" problems relevant to distributed systems as discussed in the literature. Unfortunately, it is not the case. The most difficult reliability problems arise in the actual modification of information on a storage device and in the

communication protocols [10]. The most fundamental problem is how to store and maintain, within each individual node, the current state of updates, connections to other nodes, and suspended transactions in such a way that this information survives a crash of the node: in other words, the nodes must be recoverable in the context of their function in the system. Also, it should be understood that some information used by the communication substrate may need to be replicated in order to make the substrate reliable. However, it is expected that this will be a simpler problem than the fully general case needed to support arbitrary distributed applications.

Coherent Transactions

A fundamental requirement of a data management system is the ability to transform a data base from one consistent state to another consistent state in a reliable manner. This means that the data base must not end up permanently in some intermediate inconsistent form. For example, consider a transaction that transfers funds from one bank account to another. When such a transaction terminates, it should not be possible that one account has been credited but the other not debited. The term generally used to describe such a transaction is atomic update. An atomic update either succeeds or fails, but if it succeeds it succeeds completely. Fairly sophisticated mechanisms have been designed for data management systems to ensure that two atomic updates neither slow each other down unnecessarily nor prevent the successful completion of each other [11]. In a distributed system, an atomic update may involve information in several nodes. Thus, it is necessary to provide some mechanism for coordinating nodes so that an action performed by one node is completed if and only if the related actions in some other nodes can also be

completed.

The distributed atomic update is the focus of a great amount of research in distributed systems presently, and our approach to this problem is unresolved at the moment. In principle, distributed atomic update is in conflict with our fundamental assumption of autonomy. A node that is asked to participate in a distributed atomic update must, for the moment, abandon its autonomy to the coordinator of the update. In every algorithm designed for distributed update, there is a crucial moment at which a failure of the coordinator of the update leaves the participating nodes unable to either abort the update or complete it; they must wait for the coordinator to recover. During this interval, all access to the information in question is denied. The only alternative to possibly indefinite waiting is to destroy the atomicity of the update and leave the data base in a potentially inconsistent state. Thus it is necessary to ensure that the probability of a failure of the coordinator is very low, and if the coordinator does fail, it will recover in a reasonably short time. Several schemes described in the literature suggested replication of the coordinator [12,13] - unfortunately, this substantially complicates an already complex algorithm. Other schemes advocate marking the information as tentative and disclosing it before the update is committed [14,15]. This may necessitate a cascaded back out should the update be aborted by the coordinator.

Thus, even if we accept the necessity to support distributed atomic update, we advocate that it should be limited only to those cases where it is absolutely necessary. Distributed applications should be carefully analyzed to determine whether the consistency of the underlying database cannot be ensured by some other means. We believe that examination of the real world will confirm the conclusion that it is adequate to use distributed atomic

update only as special case. The operating system should provide mechanisms that simplify implementation of distributed atomic transactions, but should not enforce such atomicity automatically. However, this approach carries with it a price. If some transactions that update a distributed data base are not programmed to be atomic, it may not be possible or easy to introduce new transactions for that data base that require distributed atomic update on different collections of data. Basically, this puts the burden on the application system designers: they have to decide if it is necessary to make distributed updates atomic, not just because of the present but also the future needs.

Extensibility

An obvious virtue of distributed systems is that it should be possible to make the system larger by adding another machine without redesigning all of the existing application code. This assumption is true, within limits. If one is simply adding a new instance of a given function, for example increasing the work force in an office that performs a particular task, then a distributed system ought to incorporate this new component with no modification, providing that the initial structure of the application included fairly flexible assumptions about the physical location at which a particular function is performed. The system we are designing will include a name lookup service, which will provide a mapping between the name and the address of a function within the system. Machines invoking a function must be prepared to discover that the address used in a previous invocation is no longer valid, so that they must refer to the name lookup server to find the new location. The necessity for tolerating failures of this sort constrains the design of

network protocols, since the name from which the current address was derived must always be available.

The more complicated problem arises when a new function is added to a distributed system. This is not simple expansion, but is sometimes confused with it. Addition of a new function to an existing system may require some existing function with which the new one must interact to be restructured and redesigned. This problem is independent of whether the system is distributed or centralized.

Protection

The philosophical assumption made at the start of this paper impacts strongly on the protection requirements that our system must enforce. Since a given personal computer is used only by one person, or by a closely cooperating group of people, there is no need to implement mechanisms inside a given machine that protect the computation of one user from the hostile advances of any other program in the same machine. This makes the building of the operating system for the personal computer a much simpler task, since the only protection requirement imposed inside a given node is that the user is protected from his own folly. Of course, folly comes in several flavors. We have already alluded to the necessity of providing a mechanism for protecting information against physical destruction. It is also necessary to protect application information and the operating system from destruction due to the execution of faulty programs. Certain hardware checks may be required for this protection, or it may be possible to do a satisfactory job by a careful analysis of the program at compile time. This is an interesting research problem in its own right, not strictly related to the issue of distribution of

the system.

Between two machines, however, the protection problem is more substantial. We choose to assume that the communication subsystem itself is not secure, so that it is the responsibility of each node itself, rather than the communication subsystem, to ensure that a transaction request coming from outside is indeed coming from a legitimate source and that the information received from outside is consistent with the state of the node. As we will show below, this particular assumption leaves us much greater flexibility in the eventual architecture of our system. This assumption is also compatible with the basic idea of the autonomy of the individual node. It should be possible for individual nodes to determine the identity of the person making a request and to decide whether or not to honor that request based on the identity of the correspondent. Architecturally, this implies that it must be possible to determine the identity of the origin for any request. In the system which we are building, an integral part of the architecture will be a specialized server whose function is to authenticate the originator of a request. The technique that we intend to use for this is encryption, based on keys distributed by the authentication server [16]. The authentication server, when invoked by the originator of a request, will send a unique key to each end of the potential communication, along with information that specifies the identity of each end. When the request originator then uses this key to communicate with the other end, the fact that each can decode the messages of the other assures each that the other is what it claims to be. Thus, our assumptions about protection have a basic impact on the form of our communication protocols, since they imply the existence of encryption mechanisms integrated into these protocols.

Operating System for Distributed Processing

Given the foregoing discussion, what must the features be of an operating system for a personal computer, if that personal computer is to be a part of a cooperating collection of autonomous nodes? Clearly, the first requirement is that a compatible set of communication protocols must be integrated into each operating system. In particular, there can be no autonomy in regard to addressing and authentication. Machines deviating from the communication standard must simply be excluded from any future conversation. We also intend to specify high level standards for the format of requests and replies and the representation of information to be transmitted between nodes. However, the implementation of these protocols may vary from node to node. Each node is assumed to have its own operating system that has full control of the allocation of the local resources. The possibility of load sharing where a job of one user is directed by the operating system to another machine in the network that has a lighter load is incompatible with our model of distributed systems.

The next requirement for the operating system is a negative requirement. The operating system must not attempt to fool the application programs as to whether application information is local or remote. The application program will wish to know explicitly whether or not information it is using is remote, because a variety of failures may occur in the remote case that would not occur if the referenced information were local. The node managing the foreign information may be down, or it may simply refuse to honor the request. If we are to have a system that is indeed reliable and robust, then the failure of other nodes must not cause a failure of the local node. But this in turn means that it is necessary to provide an alternative action in those cases where the remote request fails. For example, the alternative may be to use a

potentially out of date local copy, or to derive the desired information in some other way. In general, we believe that the alternative actions cannot be designed as part of the operating system, but must be designed by the application builder as part of every remote request. To avoid burdening the application builder with building needless error recovery code when the action being performed is entirely local to the machine, it is necessary that he understand when an action being programmed potentially generates a remote request. This assumption is in marked contrast to many proposed distributed system architectures, whose avowed goal is to hide from the application the distributed nature of the underlying communications substrate. It is our belief that hiding the distributed nature from the application can only lead to a greatly reduced robustness of the application as a whole. We wish to point out, though, that the requirements laid out in the preceding discussion do not prevent the application builder from hiding the distributed nature of the system from the end user.

The operating system must also make it possible to build applications in which the end user can modify the manner in which information is stored and organized, and the way in which information is utilized to generate responses to requests from himself and from other nodes. In other words, the user must be able to exercise his autonomy. At the same time, we do not wish to require that every user be a programmer. Thus, the application development system must make it possible for the end user to express his desires about the way the machine is organized in a manner that is easy to understand and implement. This last problem, which is very difficult and substantial, is separable from the problem of system distribution. Our most significant step toward this goal is an ongoing project that involves development of a programming system that will interface closely with the operating system and provide powerful

abstractions for modeling distributed applications [1,17].

Conclusion

The purpose of this paper has been to state a particular philosophy about the architecture of distributed systems, and then to explore, at the first level, the implications of this philosophy for the structure of the system. The philosophy we have expounded is not the only one possible. As we suggested, it is also possible to build distributed systems that resemble loosely coupled multiprocessors. If one foregoes the possibility of autonomy, then certain problems in the system become easier to solve. For example, distributed atomic update no longer raises as substantial a problem, and one has a wider spectrum of solutions available to enhance reliability and efficiency. We are interested in the architecture outlined in this paper not because it makes problems easier or harder, but because we believe that systems built according to our philosophy have more general applicability. It is always possible to find a group of people who are willing to abandon autonomy with respect to each other, but to build a system that incorporates this mutual cooperation forever limits the scope of the distributed system to that group of people. Our architecture, in contrast, can expand to encompass any desired collection of individuals. It is possible to imagine a distributed system built according to our rules expanding so that it connects together not just the employees of one company, but cooperating enterprises. Coordination between purchasing agents in one company and buyers in another company could be regulated within the context of our distributed system. The only requirement is that these people be constrained to exchange between each other only certain requests; a natural constraint intrinsic to the

architecture we propose. It would even be possible to conceive of competing companies communicating with each other through such a network, because the constraints on what can be exchanged are sufficiently well defined that the electrical interconnection of the two systems does not automatically imply paths for the leakage of information except in a manner mutually agreeable to both companies.

Ultimately, the real justification of our architecture lies in its ability to meet the needs of the real world. One of the most severe problems of data base management systems is that the information they contain is invalid, incorrect, or otherwise corrupted. The best safeguard against corrupted data is to make the system and the data in it genuinely useful to all the participants in the system, so that all participants see a personal benefit in keeping the data accurate. Our focus on the issue of autonomy, therefore, is critical because we see it contributing to the general acceptability and utility of the system, without which no system can succeed.

Acknowledgement

We wish to acknowledge the contributions of all the participants in our distributed system project at M.I.T. Although some of them may not agree with all of the points made in this paper, the paper would not have been possible without the many stimulating discussions and arguments that we have had with them. We appreciate the suggestions of those who read the draft of this paper.

References

1. Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems: Structure and Semantics," Technical Report No. 215, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, March 1979.

2. D'Oliviera, C.R., "An Analysis of Computer Decentralization," Technical Memo No. 90, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, October 1977.

3. Alsberg, P.A., Day, J.D., "A Principle for Resilient Sharing of Distributed Resources," Proc. of the International Conference on Software Engineering, San Francisco, California, October 1976, pp. 562-570.

4. Bernstein, P.A., Rothnie, J.B., Goodman, N., Papadimitriou, "The Concurrency Control Mechanism in SDD-1: A System for Distributed Databases (The Fully Redundant Case)," IEEE Trans. on Software Engineering, Vol. SE4-No. 3 (May 1978), pp. 154-168.

5. Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database," Technical Report No. STAN-CS-79-744, Department of Computer Science, Stanford University, Stanford, California, June 1979.

6. Thomas, R.H., "A Majority Consensus Approach to Concurrency Control," ACM Transactions on Database Systems, Vol. 4, No. 2 (June 1979), pp. 180-209.

7. Bartlett, J.F., "A 'Non-Stop' Operating System," IEEE Hawaii International Conference of System Sciences, January 1978, pp. 103-117.

8. Foster, J.D., "The Development of a Concept for Distributive Processing," COMPCON Spring 1976, February 1976, pp.

9. Levin, R., Schroeder, M.D., "Transport of Electronic Messages Through a Network," Technical Report CSL-79-4, Xerox Palo Alto Research Center, Palo Alto, California, April 1979.

10. Lampson, B.W., Sturgis, H.E., "Crash Recovery on a Distributed Data Storage System," Xerox Palo Alto Research Center, Palo Alto, California, April 1979 (to appear in CACM).

11. Eswaren, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L., "The Notion of Consistency and Predicate Locks in a Database System," CACM, Vol. 19, No. 11, (November 1976), pp. 624-633.

12. Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," Technical Report No. 205, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, October 1978.

13. Hammer, M., Shipman, D., "Reliability Mechanisms for SDD-1: A System for Distributed Databases," Technical Report, Computer Corporation of America and M.I.T., Cambridge, Massachusetts, July 1979 (Draft).

14. Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," Technical Memo, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts 1979.

15. Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," Technical Report No. 207, Laboratory for Computer Science, M.I.T., Cambridge, Massachusetts, December 1978.

16. Needham, R.M., Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," Comm. of the ACM, Vol. 21, No. 12 (December 1978), pp. 993-999.

17. Liskov, B., "Primitives for Distributed Computing," 7th ACM Symposium on Operating Systems Principles, December 1979.