M.I.T. Laboratory for Computer Science

Computer Systems Research Division

December 20, 1977

Request for Comments No. 153

# Providing for Sharing in a Distributed System

By Warren Montgomery

One feature of a distributed computing system that distinguishes it from more conventional computing systems is the lack of low-level sharing. In most distributed systems, primitive objects (memory cells, segments, etc.) cannot be shared by processes executing on different machines. The lack of low level sharing in distributed systems has both advantages and disadvantages.

The lack of sharing across machine boundaries establishes those boundaries as natural "firewalls" both for protection and reliability purposes. The use of an object requires the cooperation of the machine that manages that object, and thus security flaws on other machines do not directly threaten the integrity of that object. Machine boundaries also contain the damage done by failures. The software in each machine can be constructed so as to allow computation to proceed in spite of hardware or software errors in other machines.

The lack of low-level sharing also makes a distributed system more difficult to implement. Because processes can only reference directly local objects, references to remote objects must be "simulated" by bringing the remote object to the local machine, moving the referencing process to the remote machine, sending a message to a process in the remote machine that actually performs the desired operation on the remote object, or by maintaining a local copy of the referenced object. Sharing of an object by two processes on different machines requires coordination between the processes in order to move the object between the machines or to manage two copies of the object.

Many applications of computers involve some form of sharing. Thus a distributed system must provide for sharing in some way. The level at which sharing is implemented and the method of implementation have a great influence on the kinds of applications that can be run efficiently on a distributed system and the way in which they benefit from distribution. A distributed system that implements sharing at a very low level, such as Cm* [1], can run tightly-coupled applications requiring

a high degree of sharing very efficiently by providing an environment that looks like a conventional multiprocessor system. Unfortunately, the firewall effect is lost in such a system, as each processor is capable of referencing objects belonging to other processors without the cooperation of these processors.[1]

Implementing sharing at a high level, as is done by RSEXEC [2], or NSW [3], maintains the firewall effect, but makes the implementation of applications that share objects intensely more difficult, as the implementor of the application must manage copies of shared objects. An application requiring intense sharing will probably not execute efficiently in a distributed system in which sharing is implemented at a high level. If objects are to be shared between non-homogeneous machines, conventions or conversions must be established for their representation and the operations that can be performed on them.

Before implementing a distributed system, it is useful to know the nature and extent of sharing in the applications for which that system is intended. Sharing of objects in a computer system serves two purposes: It reduces the amount of storage required by eliminating the need to maintain duplicate copies of objects, and it provides a means for communication between processes because each process using a shared object can observe modifications made by other processes. The first of these purposes is of dubious value in a distributed system, because duplicate copies of shared objects may need to be maintained on several machines in order to provide all processes with efficient access to shared objects.[2] Because of the firewall effect, it also seems desirable to deliberately maintain duplicate copies of important information in order to reduce its vulnerability to failures or security violations.

Communication through shared objects, however, serves an important purpose in many current computer systems, as it is the only way in which processes can interact. Such communication significantly complicates the verification of programs, however, as one cannot depend on the contents of a shared, mutable object to remain the same while a process is using it unless synchronization is

1. For example, if one of the switching controllers in the Cm* system should fail or be corrupted, all of the objects in that system are vulnerable.

2. In fact, in many centralized computer systems sharing of memory pages between processes is not supported, thus each process has its own private copy of all of the objects that it references, including pure procedures.

imposed to prevent other processes from modifying it. Synchronization mechanisms frequently restrict the parallelism in a system unnecessarily, as constructing a synchronization mechanism that imposes the minimum constraints necessary to ensure correct operation of the parallel processes can be very difficult. Recovery from failures in a system that uses a complicated synchronization mechanism can be difficult, as one cannot determine whether or not a failed process has left shared objects in a consistent state.

Because of the awkwardness of managing shared objects in a distributed system, it is interesting to investigate alternative mechanisms for providing interprocess communication, such as the message based systems proposed by Hewitt [4], Dennis [5], or Lampson [6],[1]

The Actor model [4], describes a computation as a partially ordered sequence of events, each of which consists of the reception of a message by an Actor. An actor which receives a message follows a script which may instruct it to transmit additional messages to its acquaintances (actors previously known to it or known to the message). The following of a script is not necessarily an atomic action, so that it is possible for one actor to be processing many messages concurrently. Actors can in this way be shared, and if an actor has side effects, synchronization must be imposed to prevent anomolous behavior due to such sharing [7].

Lampson's message based model defines the only means of communication among processes to be by message passing. The contents of the messages exchanged are not constrained, nor are the patterns of message passing by individual processes. The lack of constraints may make systems based on this model hard to verify.

In the data flow model, a program is viewed as a group of interconnected data flow operations, which perform primitive operations. Each such operation performs a well defined transformation, which receives a number of typed objects and produces a number of typed outputs. Because of the primitive level of the operations, the construction of large data flow programs or complex data

1. The use of message based semantics does not avoid the need for synchronization, as constraints may need to be enforced on the order of arrival of messages at processes. A message based model does, however, make communication more explicit, and thus may make it easier to express the desired synchronization constraints accurately. Verification techniques developed for sequential programs can be used directly in systems employing only message-based communication, as the variables which are manipulated by a particular process are local to that process and are not effected by other processes that are executing concurrently.

structures can be awkward. Extensions to the basic data flow model [8], allow the construction of complex data structures and hierarchically structured programs. Data structures are constrained to be immutable, so that they can be shared, just as primitive objects can be shared, without introducing communication through the sharing. The constraint of immutability, however, makes it difficult to construct large, dynamic, and extensively linked databases. Such a data base may need to be completely reconstructed each time that a modification is made, as all parts of the data base with pointers to the modified element need to be rebuilt. Alternatively, the mutable pointers can be "simulated" by values in transit between data flow operations, however adding new elements to a data base implemented this way may require modifying the data flow program in addition to the objects that it manipulates.

The following paragraphs describe my thoughts on a model for a distributed system that provides communication only through message passing, yet is efficient and robust. The comments made on the Actors and data flow models above suggest that a good model for distributed computing must include the following concepts:

1) Independent "processes", whose only means of communication is through message passing. These processes can be of all sizes, ranging from processes that implement simple functions, such as a memory cell, to a process that responds to queries to a data base. The two important characteristics of a process are that it communicate with other processes only through message passing, not through shared objects, and that it be deterministic in the sense that the actions performed by a process should depend only on the initial state of the process and the sequence of messages that it receives. (One way to express these constraints is to require that a process be sequence-repeatable [9], so that the sequence of messages that a process sends depends only on the sequence of messages that that process has received and the initial "state" of the process.)

2) A flexible and efficient means for structuring dynamic data bases. One of the serious drawbacks of the data flow model noted above is the lack of mutable objects. The actors model, which allows mutability, seems more appropriate. The semantics of the operations that can be performed on a data structure in the Actors model are defined by the actors that

implement the data itself. It would be useful to add new ways to manipulate an object without having to re-create the object, as could be done with a model that explicitly distinguishes between the representation of an object and the manager of that object (which implements the operations that can be performed on the object).

3) A means to construct units that represent appropriate abstractions from units implementing lower-level abstractions. This can be done either strictly hierarchically, as in CLU [10], or by the acquaintances relationship used in the Actors model. The decision of which organization to use interacts with the goal of sequence-repeatability, as different rules must be followed to insure that all modules that can be constructed are sequence repeatable.

The requirements above suggest an organization in which there are modules that represent various abstract concepts, such as functional or data abstractions, which communicate via message passing. What remains to be specified are the rules for building new modules, the conventions for interconnecting modules, and the means to insure that such a system is robust in the face of failures of the modules or the communication links between them.

The proposed organization can be viewed as a hierarchy with levels of modules. At the top level, an application is viewed as a collection of communicating modules, which exchange messages among themselves in either a fixed or variable pattern. Each of these modules can in turn be viewed as an application which is constructed as a collection of modules. There is no sharing in this hierarchy, so that the sets of modules used to implement two different modules at the same level in the hierarchy are always disjoint.

Programming in this system consists of describing an application as a collection of communicating modules. When such a program is executed, instances of the modules used are created and installed in the application. Each instance of a module is completely independent of other instances of that module, and shares only the specification of what it does. Modules are building blocks like integrated circuits, in that complex structures are built up from interconnections of relatively few types of primitive components.

The modules implement primitive processing steps known as transactions, in which a set of input values is absorbed by a module and a set of output values is produced. The transactions performed by a process are atomic in that they do not overlap in time.[1] The operation of a process is required to be sequence repeatable in that the transactions that it performs and the values produced by those transactions must depend only on the sequence of input messages and the initial state of the process. For example, a process that implements a true random number generator, with each transaction producing a random output value, would not be allowed. A pseudo-random number generator, in which each output depends on the last value produced, is a sequence-repeatable process.

In addition, many modules may be memoryless (repeatable in Henderson's terminology [9]) meaning that the outputs of each transaction depend only on the inputs to that transaction and not on the previous history of the module. The implementation rules for the structure of modules that can be used to implement a particular module must be carefully chosen so as to maintain sequence repeatability. It is possible to construct a module that is not sequence repeatable from a collection of modules that are.

Modules can be classified by whether or not they are memoryless, and by the pattern of message passing among them, or equivalently by the set of modules to which the outputs of a module are sent. This set can be fixed, and established when a module is bound into the implementation of an application, or dependent on the inputs to the current transaction, or dependent on the sequence of inputs to the module. By restricting the way in which modules can be interconnected, we can guarantee that an application constructed of interconnected modules is sequence repeatable.

Modules can be used to represent the familiar concepts of current programming languages. Decisions can be made by directing the outputs of a module to one of two possible destinations, based on the inputs to the transaction being processed. Loops can be represented by memoryless modules that direct their outputs back to themselves. Procedures can be represented by modules that direct their outputs back to the the process that supplied the inputs to the current transaction. These constructions will be explored more thoroughly in later memos.

---

1. This is one important aspect in which my model differs from the Actors model in that the processing step performed by an Actor when it receives a message may overlap with the processing of other messages, as noted earlier.

An error in a computer system has frequently been treated as the failure of a function to conform to its specifications. While errors of this nature do occur in distributed systems as well as in centralized systems, they are generally less frequent than the temporary or permanent loss of information or processing ability due to failures of the nodes of the system or the communication links which connect them.[1] Such errors are not well modeled by the conventional treatment above because they tend to occur asynchronously with respect to other processing activity, and are not caused by the failure of any specific function, but are detected when a requested operation fails to complete or produces unexpected results. These failures also explicitly involve the notion of time.

We therefore define an error in a distributed information system to be the loss of information or processing capability for some period of time (which can be infinite). Several goals may be desired in dealing with this kind of error.

1) Minimize the probability that a particular unit of information is unavailable at any time.

2) Maximize the probability that a particular operation can be performed, even if processing capabilities are lost.

3) Maintain a group of information holding objects consistently, such that some predicate describing consistent states of the group is always satisfied.

4) Perform transactions operations atomically, such that either all or none of the effects of performing such a transaction are seen at all times.

Several algorithms have been developed to reduce the probability of information loss by maintaining duplicate copies of vital information [11,12,13,14].[2] These seem sufficient to guarantee the availability of data with high probability at the cost of maintaining redundant copies and increasing the difficulty of performing updates. One can increase the probability that a particular operation can be performed by performing it on several machines and using some voting scheme to determine which results to accept. Thus reliable operations can be obtained at the expense of wasted processing

1. In this section, we are primarily concerned with failures of the machinery required to execute a program for an application (hardware, operating system, communication links, etc.) rather than errors in the program itself. Programming errors can be eliminated by careful debugging or program verification, however, failures of the underlying machinery remain a significant problem, especially in a distributed system.

2. The Lampson and Sturgis algorithm [13] is somewhat different from the others in that it is not intended to provide reliable storage of data, but provides atomic transactions by maintaining duplicate copies of the information necessary to complete a transaction that has been started.

capability and some complexity in determining their results.

Performing atomic operations and maintaining consistency are closely related. Operations performed on consistent sets of objects must appear to be atomic, as a partially completed operation could leave such a set in an inconsistent state. Atomic operations alone do not insure that the objects on which they operate remain consistent if objects can be individually modified or destroyed by failures. While consistency cannot be maintained in the face of all possible failures, algorithms can and have been developed to perform atomic transactions and maintain consistency in spite of some restricted classes of failures.

In a message based system, reliability can be achieved by implementing operations as atomic transactions, implementing their state as a consistent set of objects, and using redundant copies and error-detecting and correcting communication protocols to minimize the probability that messages sent between modules are lost. Critical operations can be replicated on several machines in order to increase the probability that they can be performed.

The lack of write-sharing in a message based system reduces the difficulty of maintaining writeable objects consistently and reliably. Changes made to objects are observable only to the process making those changes and need not propagate to a large set of observers. Communication between processes must, however, be reliable so that the sequence of messages received by a process accurately reflects the sequence of messages that that process was sent. If this is not the case, then there is no way to distribute updated versions of objects and guarantee that the updates are received by all observers.

## References


[1] Swan, R.J., Fuller, S.H., and Siewiorek, D.P., "Cm* -- A Modular, multi-microprocessor," AFIPS Conference Proceedings Vol. 46 pp. 637-644, 1977.

[2] Thomas, R.H., "A Resource Sharing Executive for the ARPANET," AFIPS Conference Proceedings, Vol. 42, June 1973, pp. 155-163.

[3] Crocker, S.D., "The National Software Works: A new method for providing software development tools using the ARPANET," Proc. Meeting on 20 years of Computer Science, Pisa, Italy, July 1975.

[4] Hewitt, C., "Viewing Control Structures as Patterns of Passing Messages," M.I.T. Artificial Intelligence Laboratory, A.I. Memo #410, December, 1976.

[5] Dennis J.B., "First Version of a Data Flow Procedure Language," M.I.T. Laboratory for Computer Science Technical Memo, TM-61, May 1975.

[6] Lampson, B.W., "Protection," Proc. Fifth Princeton Symposium on Information Systems, pp 437-443, March 1971. (reprinted in Operating Systems Review 8, 1, pp 18-24, January 1974.)

[7] Yonezawa A, and Hewitt C., "Modeling Distributed Systems," Proc. Fifth International Joint Conf. on Artificial Intelligence, August 1977.

[8] Dennis J.B., "A Language Design for Structured Concurrency," M.I.T. Laboratory for Computer Science, Computation Structures Note 28-1, February 1977.

[9] Henderson, D.A., "The Binding Model: A Semantic Base for Modular Programming Systems," MIT-LCS TR-145, February, 1975.

[10] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," M.I.T. Laboratory for Computer Science, Computattion Structures Group Memo 144-1, January 1977.

[11] Alsberg, P.A., Belford, G.G., Day, J.D., Grapa, E., "Multi-Copy Resiliency techniques," CAC Document # 202, May, 1976.

[12] Johnson, P.R. and R.H. Thomas, "The Maintainence of Duplicate Databases," ARPANET NWG/RFC #677, January 1975.

[13] Lampson, B. and Sturgis, H., "Crash Recovery in A Distributed Data Storage System," Xerox Palo Alto Research Center, ca. November, 1976.

[14] Thomas, R. H., "A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control," BBN Report #3340, July, 1976.