M.I.T. Laboratory for Computer Science January 19, 1978

Computer Systems Research Division Request for Comments No. 154

NAMING OF OBJECTS IN A DISTRIBUTED AUTONOMOUS COMPUTER SYSTEM

Ph.D. Thesis Proposal

by David P. Reed

The problem of naming objects in a distributed computer system where each node of the distributed system is under autonomous control is considered. I argue that in order to deal with the problems of autonomy, unreliability, and delay, the basic naming mechanism used in each node must be able to deal with multiple copies of objects and reversible bindings. In addition, the naming mechanism must be able to allow the evolution of objects (especially program objects) over time. I propose to investigate a naming mechanism based on a concept I call object-families that takes into account these issues.

Introduction

A current trend in the construction of computer systems is the construction of systems made up of multiple separate computers connected to each other through a data communication network. Although this construction may have advantages in terms of cost, reliability, extensibility, etc., the main reason for this construction seems to be the relative autonomy of each individual system from each other that can be achieved while at the same time allowing a high degree of resource sharing.

A key mechanism for achieving effective sharing of resources among a set of computers is the naming mechanism used to access the resources. In a single, central computer system the sharing of resources is aided by the presence of a universal naming mechanism usually provided at the operating system level as part of the file system. In a distributed system, where each individual computer node has a high degree of autonomy, whatever naming mechanism is used will need to reflect the autonomous nature of the individual systems.

I have identified three basic issues that impact on the design of a naming mechanism to be used in a distributed system consisting of a set of autonomous computers. They are the use of names to achieve sharing and communications among users via the system, the use of multiple copies of data and programs in the system, and the need for reversibility in the bindings between objects in the system. Before discussing these issues in detail, I will present a model distributed system that incorporates the ideas of distribution I am trying to capture. Then I will discuss each of the issues in turn. Finally, I outline the basic ideas of an approach to the problems of naming in a distributed system.

Distributed System Model

For the purposes of the research proposed here, I will consider a fairly general model of a distributed system. The kind of distributed system I will consider might be called a "distributed programming system," to distinguish it from other special purpose kinds of systems, such as distributed database management systems or distributed command and control systems. The basic distinctions are that a distributed programming system provides a diverse user community with diverse needs with a common means to share and communicate programs and data, and that it is not possible to restrict in advance the modes of sharing among users.

I will assume that the distributed system consists of a number of computers, called nodes, and an underlying communications system that allows each node to communicate data with each other node. All of the nodes are computers of identical architecture, although the scale, performance, and I/O complement of the nodes may vary widely. The main reason for this uniformity of architecture is to facilitate sharing of all kinds of data and programs by copying through the computer system. Of course, the naming architecture of every node must be the one that I will develop. (1)

Each node of the distributed system is under autonomous control. That is, each system is under the control of a distinct person, the owner, who has the right to completely control the uses to which the resources of his machine are put. Thus the owner may or may not let others use the processor or memory of his system for execution of their programs, he may or may not allow programs entered into his system to be shared with users on other systems, and he may in fact arbitrarily choose to revoke the use of resources that he has previously granted for the use of others in the distributed system. He may also set his own schedule for his machine's availability, and choose his own desired level of cost/reliability/speed tradeoff.

(1) At this point, I have eliminated from consideration the problems of retrofitting the research I propose onto existing heterogeneous computer networks -- an important problem, but one that is much too complex to tackle now.

Autonomy stops where maliciousness begins, however. The owner of a system, as part of being a member of the distributed system, must agree to a certain set of protocols that his system must follow in its relations with other systems. These protocols, of course, must be carefully designed in order not to infringe upon his autonomy. The inter-system naming mechanism is such a protocol.

All components will also be subject to reasonably likely failures. In particular, the communications system may fail in whole or in part, making communications between particular pairs of nodes occasionally fail. Each node may crash (not too different from the owner shutting the system down), and individual programs, data, or other services provided by a node may vanish or fail. Another goal to be satisfied by the protocols used between nodes in the distributed system are that they should be robust in the face of these kinds of failures.

The distributed system I will consider is object oriented. By this I mean that the system has facilities for constructing new abstract data types in the manner suggested by SIMULA classes, CLU clusters, or ALPHARD forms. An abstract data type consists of a set of objects together with associated operations that are allowed on the objects. In my model system, the objects of the defined type are implemented (stored) as objects of another type (the representing type), and the operations on the object are implemented by a set of programs called the type manager that have the special privilege of being able to access the representation of objects of the managed type directly. Other programs can deal with the managed type only through the operations provided by the type manager.

The restriction of the interface provided by objects managed by type managers increases the opportunity for the protocols between the nodes of the distributed system to be aware of the "equivalence" of distinct things by masking non-essential detail of the implementation. An important example of this "equivalence" is the idea of an upward-compatible object manager. The supplier of the object manager for a type may choose to improve his implementation of objects of a particular type, without changing the representation form (by using better algorithms, or providing additional operations not present in earlier implementations). Changes to object managers of this form can be made transparent to users of the objects -- a new manager can replace the old with no disruption of function. In a distributed programming system, this mode of communication of programs will be important.

In any system where sharing of information is provided, security and protection should be considered. The basic problem in security and protection in a distributed system is to decide what security policy best suits the nature of the interactions within the system. I consider this a separable problem from the problem of naming, and will consider it only secondarily. Several mechanisms for implementing protection mechanisms in a distributed system are well-known, such as encryption, physical isolation, etc. There are two kinds of protection that should eventually be considered in a distributed programming system. The first is protection from attacks at the intersystem interface. I include in this class all attacks involving interception and interference with communications, and also all attacks that involve modifying the architecture of a foreign node so that it violates the basic agreement to use common protocols. The second kind of security involves the protection of data that has been shared with a foreign node by copying into that node. Some of the mechanisms for providing autonomy provide also solutions that can help with this problem, such as prevention of copying a program or data base from one machine to another, requiring it to be used on its home machine.

I am assuming an underlying universal naming scheme that allows each node in the distributed system to uniformly name each other node, and allows each node independently to create objects without name conflicts arising. An underlying naming scheme where each object on a node is named by a concatenation of the node's unique identifier, and an identifier unique within that node will suffice to satisfy these requirements.

Naming and Sharing

In central computer systems, a major function of naming is to allow the sharing of data objects, such as programs and data bases. Sharing is a kind of communications function among different human users and their program agents. The view of communication through a shared mutable data object, such as a file, is quite different from the view of communication as a telephone line between a pair of communicators. The sender of information through a shared object does not have to know who will eventually use the information, and the receiver of information does not have to know who supplies the information. The decoupling of senders from receivers is an important idea, and one that we would like to retain as much as possible in a distributed system.

In a distributed system, the general case of sharing of mutable objects so easily obtained through shared memory in a central system is quite difficult and expensive to obtain. The basic problem is that a change to an object cannot be propagated immediately to all users of that object because of communications delays. It is possible to turn this disadvantage to an advantage, however, because on closer inspection of the uses of shared objects in a central system, we find that immediate propagation of changes to an object to all users causes trouble in most cases.

The sharing mechanisms provided by a distributed system must be tailored to the distributed environment. An example of a sharing technique that is important in distributed systems is the use of multiple, duplicate copies of an object. Multiple copies are discussed in the next section.

A powerful tool in managing the sharing of objects in a distributed system is an ability to name each element of the sequence of values that a mutable object assumes through time. The term I prefer to use for the elements in the value-history of an object is <u>version</u>. With the ability to name individual versions of an object, the problem of synchronization of accesses to data bases is easier to formulate. Synchronization is simply ensuring that the correct version of each object is accessed in achieving the desired result.

Having given the individual versions of an object names, one can allow multiple versions to co-exist within the system.

<u>Multiple Copies</u>

A natural result of distribution is the need for multiple copies of an object. There are four basic reasons for this.

1) Autonomy. If a user on machine M1 wishes to offer to a user on machine M2 a program or data base, the user of M2 may want to copy

the program or data base because he knows that M1 does not want to provide the resources for storing or executing the program state or data base on behalf of M2, or M1 may not always be up when M2 wants the resource.

2) Robustness. A user offering a service in a robust way may want to prevent the bad effects of an isolated local communications failure or machine failure, by providing, throughout the network on cooperating machines, multiple copies of the programs and data that make up his service.

3) Quick response. If a user wishes to apply a program to some data and each is on a separate machine, it may be much more efficient in terms of communications time to ship the program to the data or the data to the program (or even have both rendezvous in the middle), rather than having the program access the data bit by bit through the network. If the program is run on the data several times, it is advantageous to do the shipping of program or data only once, resulting in a semi-permanent second copy of program or data.

4) Orderly evolution. One example of orderly evolution is a shared program module, maintained by some individual, who releases new versions as he improves his module. As a user of the module, one would like to have the current version (if possible), but updates to the module should not interfere with executions of the module in progress at the time the update is performed. Thus there may be the copy that is in use, and also the copy that is current. If the modules have "equivalent" behavior (i.e. the maintainer improved its speed but not its interface), such an update may not be a "side-effect" in the traditional sense, yet it has multiple copies.

I should note here that my view of updates, noted above, also results in multiple copies.

For these four reasons, there may be several "equivalent" copies of an object at any particular moment in the system. A new process that wants to transact with the object may be willing to deal with any one of several copies -- it is part of the job of the naming system to provide access to some particular copy. An especially important case of the multiple copy problem is allowing a system to access a copy of an object already on that system rather than going out to the object's owner system to get a new copy on each access. This case is analogous to encachement of data in primary memory or registers in a central computer system.

Reversible Bindings

The use of object references within objects in a computer system raises a number of binding-time problems. Typically, when an object such as a program module is constructed by a user, the objects named are specified by human-level names. There are two immediately obvious times to resolve the human-level name to a specific version of the named object. One can resolve the name when the module is created, or when it is used. Resolving the name when it is created has the drawback that the object can never refer to more recent versions of an object than the version referred to when the object is created. On the other hand, resolving the name when it is used has the drawback that the version chosen may be not comparable to the version intended by the programmer. Even worse, if a program uses a name several times, resolving each use independently at the time it is used can cause the behavior to be wrong if distinct versions are used.

Thus, even in a central system, the problem of how late to bind a human name to a particular version is difficult. In a distributed system, differences of binding time may have a strong performance effect and a strong effect on autonomy and robustness as well. Consider a shared program module, for example. If it is possible to copy program modules from system to system, the most efficient version to access will always be the local copy on the local computer system. If we always try to resolve the shared module name at each use, the cost of always obtaining the most recent copy may be high in terms of network bandwidth and delay. Further, getting the latest version may be handicapped by finding the home system of that copy to be down for maintanence or a crash at the moment, thus making the local system dependent on a large portion of the net for its operation.

On the other hand, if there is no local copy of the shared module at the moment, having a name that was resolved too early can be a handicap for the same reason. If the system that houses the version to which the name is bound is down, it is not possible to perform the service, even if there are equivalent versions on other systems that have been provided against just such a possibility.

An approach to the problem of when to bind names in a distributed system is what might be called "reversible binding". For each resolved name, it may be discovered at some later time that the object referred to has disappeared -- either because the object is at a host that is down, or because some autonomous holder of the object has deleted it. The disappearance of an object may eventually cause some kind of program exception when the object is used. At the time of the exception it may well be possible to reevaluate the original name to obtain an equivalent version of the object. The mechanism of reversing a binding and then re-evaluating it requires that the original naming information be available at the time the binding is re-evaluated. Thus a name for an object will contain a "hard reference" -- the evaluated binding, and a "soft reference" -- the unevaluated binding information.

An important problem in doing binding reversal is limiting its impact on the correctness of the running programs. This problem is tied to the notion of an "equivalent" object. Equivalence is highly context dependent. In an object based system, however, there are several important cases where transparent reversal of bindings is possible. One case is the substitution of a different version of a type manager that supports the same representation and at least the same set of operations as the original. Another is the substitution of a different instance of a multiple-copy object that supports simultaneous updates, such as that of Thomas [19].

Object Families and Reversible Pointers

The two basic concepts I introduce to help in the management of objects in the distributed system are object families and reversible pointers. An object family is a group of related objects that a user thinks are "equivalent". Basically, an object family consists of a number of versions of an object that are related by evolution over time (the successive values of a cell, for a simple example), and within each version, a number of instances of that version, perhaps differing in representation or location within the system but not in behavior.

In order to model updates, within an object family there is one distinct version (which may consist of several different objects) that is designated to be the current version. Performing a update on an object involves creating a version of the object and designating it to be the current version. This mechanism is a variant of the approach taken by Stearns, et al. [18].

At least one of the objects in the current version of an object family is known to a part of the distributed system called the family controller for the object family. The family controller is part of the type manager for the abstract type containing the object. The family controller may be distributed, allowing updates to the object family to be handled robustly, perhaps by a mechanism such as that of Thomas [19]. It is possible to create copies of the current version, but not have them known to the family controller. A version is identified by a unique version number, and it is possible to ask the family controller at any time whether an object is a copy of a "current" object.

All objects behave like the tokens of Henderson's Binding Model [10]. They are created without a contained value, and then the value is stored. Once the value has been set, they may be read, but not modified further. Any read attempted before the value is stored will wait until the value has been stored. An improvement on this notion associates with an object before the store has been finished a timeout and a "previous value" that will be assigned to the object if the timeout is exceeded before the object is completely filled in. This mechanism provides for robustness in the case that the program that is to fill in the object fails before doing so. The timeout and previous value are managed by the family controller.

Two notions of autonomy are incorporated into the structure of object families. First, there is a notion of copiability restrictions that are enforced by a family controller. Basically, some object families may be restricted to not have copies in certain nodes in the system. The second notion is that any object in a family may be deleted at any time by the node that holds it. The reason might be lack of resources, or the fact that the object has become old enough so that it is obsolete (no other systems are likely to reference the object).

To aid in the orderly deletion of objects, a sort of timeout garbage collection mechanism will be provided on each system. Each object will have a "time of last use" associated with it, and a timeout period. If the time of last use is more than the timeout period in the past, the object should be deleted. The timeout period may be set to an appropriate value to indicate the expected frequency of use of the object. Protecting an object from garbage collection may be done by periodically sending a signal that updates its time of last use. A mechanism will be provided whereby a whole structure connected by pointers can be protected by propagating the signal.

Object families in the system are named at the human level by human level pathnames. Since object families may be distributed, it is important that the human level naming scheme be distributed also, in order to insure robustness. A scheme I intend to use for the human level naming function has been described in a separate paper discussing a mechanism for naming generic services in a network [16]. That scheme can be fairly simply adapted to fit in the scheme I am describing here.

Interobject naming in the scheme will be accomplished by using reversible pointers. At any particular time, a reversible pointer contains two basic components -- a hard reference that names a specific object, and a soft reference that contains enough information to create a new hard reference if the current hard reference information becomes unusable because the object referred to is either deleted or unreachable. The actual form of the information in the soft reference is a subject for further research -- it might be a logical pathname of an object family, for example. The hard reference in a reversible pointer may vary in its degree of specificity -- it may refer to a particular object family, or a particular version within that family, or a particular object within that version. If a name refers to an object family, without further qualification, conceptually it refers to the "current" version among all the versions contained in the family. Thus such a name will track updates in the family. There will be information in the hard reference for such a pointer that indicates where the current version number is likely to be (derived from the last use), and where an object that is current is likely to be found, in order to refer to the object. Such information is advisory only, and is checked on the actual reference.

If a hard reference refers to a specific version, then it does not see updates to the object (although the whole version may be deleted by fault or autonomous action). If copiability restrictions allow, such a name will probably refer to a copy of the object on a local system when it is used. This allows the system to preserve a version of an object that otherwise would be under autonomous control at a foreign site. In the hard reference will be advisory information suggesting where an object of the appropriate version may be found.

Another mechanism I am considering is the ability to control the process of pointer reversal, by having the programs responsible for translating from soft reference to hard reference trap to a program to be executed when the pointer is reevaluated. This mechanism would allow a more general view of "equivalence" to be supported by the system, since the program doing the re-evaluation can be specific to the application and context of the reference.

Related Work

The three major themes of this work are naming of objects, synchronization of concurrent operations on objects, and robustness of networks under partial failures of communications and nodes. While there is a reasonably large literature on each of these topic, I have not found any attempts to tie the three areas together. Here I will try to describe those pieces of work that come closest to mine.

The concept of naming objects with identifiers unique for all time is a key concept in my work. A good example of the use of unique identifiers is found in the implementation of the Multics storage system [2] segments. Also related to the idea of unique identifiers is the concept of capability [4, 9]. Extending the unique naming of objects into a network context has been suggested by me for naming the terminations of virtual connections in an end-to-end network communications protocol [15].

The semantic model developed by Henderson for objects and structures of objects is one of my basic jumping-off points for defining the types of objects I want to implement [10]. However, this model and other models of objects developed for object naming in programming languages do not take into account very well the sharing of permanently stored objects among a community of "autonomous" (not under common authority) users. Some of the issues of sharing permanent objects among a community of users have been discussed by Saltzer [17]. Saltzer discusses one aspect of autonomy -- that of allowing sharing between users of programs and data in a way that was not planned for at the time the programs and data were created.

The idea of naming distinct versions of objects -- i.e. assigning names to values rather than value-containers -- is also very important to my work. This idea is basic to systems based on pure applicative languages such as LISP [13]. It is also basic to the data-flow architecture of Dennis [5]. The naming of succeeding versions of objects with related but distinct unique names is closely related to the idea of naming versions of files with increasing version numbers in a file system, coupled with the convention of never rewriting a file in place. This practice in using files is encouraged by operating systems such as ITS [6] and TENEX [3]. Another use of this idea is the proposal of Easton [7] eliminating the use of long-term locking on objects by referring to successive versions by distinct version numbers.

The description of time in a distributed system due to Lamport [11] is very important. One of the major results of the thesis will be showing that synchronization of accesses to values should be carried out with respect to such a distributed notion of time, rather than an omniscient notion of time as provided by locking protocols.

An important problem in handling updates and accesses to multiple objects is that of mutual consistency. Eswaran, et al. [8] have given a careful definition of this notion. Stearns, et al. [18] have given locking protocols that guarantee mutual consistency in a distributed system, and have also examined the problem of preventing deadlock in such a system. An interesting feature of Stearns' formalism is that it is defined in terms of the creation of distinct versions of objects rather than in terms of objects being updatable cells.

The last theme of this thesis is maintaining correctness in the face of failures. I have previously described a user level naming scheme for objects that is robust in the face of certain kinds of failures [16]. Other robust protocols that seem relevant here are in the area of achieving consistent database transactions in the face of failures of various distributed systems. Lampson and Sturgis [12] have described a mechanism that achieves update of multiple copies of data in a way that is robust. I hope that the ideas I present in the thesis will lead to a mechanism that is simpler and more easily understood. Thomas's [19] algorithm for distributing updates to multiple copies has a weaker form of robustness. Alsberg [1] presents a much simpler approach to robustness in the case of multiple copies, which sacrifices autonomy by using a more centralized form of control. Menasce, Popek, and Muntz [14] present a protocol for maintaining a system-wide locking database that is robust in the face of communications and processor failures.

Proposal

I propose to continue investigating the use of an object naming mechanism like the object family model above as a solution to the problems of effective sharing, multiple copies, and reversible bindings in the context of an autonomously managed distributed system. Among the important questions left unanswered are the suitability of the naming mechanism for use in a purely intra-host environment, so that inter-object naming is done in a uniform manner no matter where in the distributed system, and whether the proposed mechanism has any hidden restrictions that would tend to reduce drastically the autonomy of any host connected to the distributed system.

I propose to investigate the model more thoroughly, and describe it in careful detail. My basic schedule is to spend the summer and fall, 1977 terms predominantly in research developing the model in detail, and the late fall, 1977 and spring, 1978 terms describing the results of the work in a thesis.

The basic result of the research will be a paper study of the proposed naming mechanism in the context of a distributed programming system. Some small programming experiment may be made to clarify details of the model, but no extensive implementation is intended as a result of the project.

References

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

[2] Bensoussan, A., Clingen, C.T., and Daley, R.C. "The Multics virtual memory: concepts and design", CACM 15 5, pp 308-318, May 1972.

[3] Bobrow, D., et al., "TENEX - a paged time sharing system for the PDP-10," Communications of the ACM 15, 3 (March 1972), pp. 135-143.

[4] Dennis, J.B. and Van Horn, E.G., "Programming semantics for multiprogrammed computations," Communications of the ACM 9, 3 (March 1966), pp. 143-155.

[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] Eastlake, D., et al., ITS 1.5 Reference Manual, M.I.T. Artificial Intelligence Laboratory Memo AIM-161A, July 1969.

[7] Easton, W.B., "Process Synchronization without Long Term Interlock," Proceedings of the Third ACM Symposium on Operating Systems Principles, (Operating Systems Review 6, 1 and 2) (June 1972), pp. 95-50.

[8] Eswaran, Gray, Lorie, Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM 19, 11, November, 1976.

[9] Fabry, R.S., "Capability-based addressing," Communications of the ACM 17, 7 (July 1974), pp. 403-412.

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

[11] Lamport, L., "Time, Clocks, and the Ordering of Events in a distributed System," Mass. Computer Associates Technical Report CA-7603-2911, March 1976.

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

[13] McCarthy, J., et al., LISP 1.5 Programmer's Manual, 2nd edition, M.I.T. Press, Cambridge, Mass. 1965.

[14] Menasce, D.A., Popek, G.J., and Muntz, R.R., "A locking protocol for resource coordination in distributed systems," UCLA Computer Science Dept. SDPS-77-001, October 2, 1977.

[15] Reed, D.P., "Protocols for the LCS Network," LCS-LNN #3, November, 1976.

[16] Reed, D.P., "A Service Addressing Protocol for the Local Network," M.I.T. Laboratory for Computer Science Local Network Note #5, December 1976.

[17] Saltzer, J.H., "Naming in Information Systems," chapter 5 of 6.033 notes, fall, 1976.

[18] Stearns, R., et al., "Concurrency control for database systems," IEEE Symposium of Foundations of Computer Science CH1133-8 C, October, 1976, pp. 19-32.

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