M.I.T. Laboratory for Computer Science

Computer Systems Research Division

July 20, 1978

Request for Comments No. 164

Copying in a Distributed System

Karen R. Sollins

Attached is my recently accepted Master's thesis proposal.

Massachusetts Institute of Technology

Laboratory for Computer Science

Proposal for Thesis Research in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

Title: Copying in a Distributed System

Submitted by: Karen R. Sollins
9 Southwick Circle
Wellesley, Mass. 02181

Date of Submission: July 20, 1978

Expected Date of Completion: October 30, 1978

Brief Statement of the Problem:

This research explores some of the issues surrounding sharing of information in computer systems. The particular type of a computer system considered here is that of a distributed system composed of autonomous nodes naming objects independently. The assumption is made that the system supports extended type objects. The particular issues that are addressed are related to naming. The mechanisms developed are aimed at achieving modularity and sharing of information while providing uniformity of mechanism. Two types of copy operations are to be supported. The first is that of copying only the top level of the structure representing the abstract object to be copied. The second is a copy operation on the complete structure. The specific problems attacked here are (1) what information and in what form should be passed between machines to copy information across machine boundaries, and (2) how to generalize the copy requests across machine boundaries.

Supervision Agreement:

The program outlined in this proposal is adequate for a Master's thesis. The supplies and facilities required are available, and I am willing to supervise the research and evaluate the thesis report.

L. Svobodova
Assistant Professor of
Computer Science and Engineering

Background

In a distributed system composed of a collection of interconnected computers, there must be a layer in which the system as a whole is able to communicate. Above that layer, the distributed system appears to be a single entity, although its distributed nature may still be evident at higher layers when machine boundaries are crossed, if nothing else, then, by the delays experienced. Below the communication layer the hosts appear to be a group of individual machines receiving and sending out information on the communications medium, which at these lower layers is only considered to be a collection of sources and sinks of information. The level at which the communications layer is placed varies from one system to another. Halstead[1] presents a system[1] in which communication takes place deep inside the operating system and is transparent to the user. A system such as RSEXEC[7], on the other hand, does not hide as completely from the user the machine boundaries. I will be considering a situation in which the fact that there are separate machines will be obvious to the user, although the mechanisms provided will simplify crossing machine boundaries. In particular, I will explore copy instructions for crossing machine boundaries to be made available to the user and the naming mechanisms necessary to support them.

1. One of the goals of the system is load sharing at a low level and support of parallelism within individual computations. As a result, the machines in this system are so intimately involved with each other that the system might better be labelled a multiprocessor system than a distributed system.

Naming

Saltzer [6] has analyzed, in detail, naming issues and solutions in a centralized system. He presents two goals to be achieved by his low level naming mechanism, sharing and modularization in naming. By sharing is meant the ability of two or more people or programs to access the same piece of information. This does not mean that they have different copies of the information that appear to be identical, but rather that they may use some of the same pieces of information. This implies that there must be a way of naming the shared object so that several people or programs can access it, although the different parties need not necessarily use the same name. This in turn implies that at some level in the naming mechanism, there must be a way of uniquely identifying an object. The assumption here is that at some level, a single name will exist for a specific piece of information. Modularization means allowing two or more people, procedures, or objects to use the same names for different objects. The need for modularization arises from the desire to create and use programs and subprograms without regard for the names that are local to them. A program should be able to call a subroutine, without first checking that the names that are internal to the subroutine do not conflict with any of the names that are used by the calling program.

In order to describe the mechanism developed by Saltzer, several definitions are needed. A context is an entity that translates names of objects into other names for the objects. The set of names from which it translates must contain no duplicate names; the translation by a context must contain no ambiguities. An object can contain among other things names of other objects. In order to allow for modularity the names that an object uses will be interpreted relative to a context. An object and the context in which the names it uses are resolved is called a closure. A closure table can be created to hold closures, each of which consists of an object name and a context name. Thus contexts and closures both represent functions on object names: the context maps one version of a name into another, and the closure table maps the name of an object into the name of a context in which the names the object uses will be resolved.

The complete mechanism developed by Saltzer makes use of two registers as well as a closure table and context to specify completely a reference that consists of an object name and an offset into the object. Figure 1 is an example of Saltzer's mechanism.

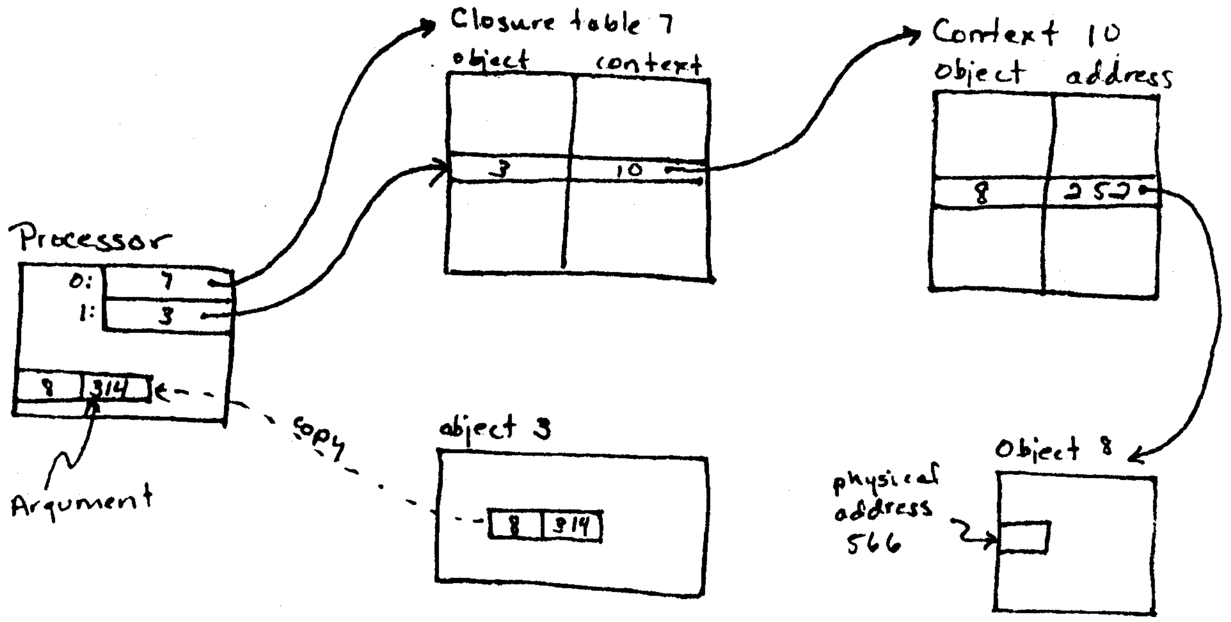

Figure 1

The reference to be resolved is object 8, offset 314, in object 3. Register 0 is the closure table pointer register, which contains a pointer to the closure table, 7, to be used to find the closure of the object providing the current reference. Register 1 is the pointer source register which contains the name of the source object, 3, which contains the name to be resolved. These two registers together indicate the context, 10, to be used to resolve the name 8 specified by the source object. The context contains an entry for name 8 with the corresponding address 252. Now when 252 and 314, the offset, are summed, the word to be accessed is found to be 566. By assuming that the closure table pointer register, the slot for the context name in the closure table, and the slot for the address of object 8 in context 10 all contain absolute addresses, the simplest form of this mechanism has been exemplified. There is no reason that this always need to be the case, as long as the name resolution process is guaranteed to stop somewhere, i.e., at some point a name which needs no further resolution is produced by a context. Multiple levels of name resolution may be useful in allowing flexibility in binding names and objects together. Since at some point, a name that will not be resolved further (possibly an address) must be reached, the second name of each of the pairs of names in a context can be either a name to be resolved further, or the end of the name resolution procedure. There is no reason that a context cannot contain both types of names as long as they can be distinguished. This is easily achieved by tagging. An alternative method for handling the two kinds of names is to have two types of contexts, those that produce names needing further resolution, and those that produce names that do not. If one creates one of the latter type of context with all the objects in it that are known in the system, one has what Luniewski[4] calls a system map in his discussion of contexts.

Objects

There are two current trends in computer systems that point in the same direction. The first is the approach of providing a programming methodology conducive to specifying from the top down a more and more specific set of solutions to a problem. This is the approach taken by the CLU project[2,3]. It has led to a statement of a solution at the most abstract level. At each level the abstractions needed to specify the abstractions at the previous level must be found (created if they do not exist). This proceeds on down until the basic types provided by the system are reached. Another trend is the bottom up approach taken by the HYDRA[9] project. In this case, resources are abstracted, to the point where they provide the virtual resources that are to be made available to users of the system. In light of both of these points of view, I will consider systems composed of objects or abstractions.

Objects will be considered to be fairly simple entities. They are nameable; one has to be able to refer to an object. There may be more than one name for an object, but there must be at least one. There is for every object, a manager or guard that administers the object, keeps track of whether the object exists, and only allows references to those objects that do exist. This name manager also is a repository for the names of the objects that exist. The result of this is that an object may have a lifetime extending beyond the lifetime of the object or process that created the object. Another consequence of this is that saving a name of an object somewhere besides in the appropriate mapping maintained by the manager does not guarantee the existence of the object. A second basic quality of an object is that it is capable of having a value associated with it. There is no reason to require that an object be assigned a value when it is created (alternatively, an object could be created with a special value "undefined"); it may turn out that the value of a particular object that is important must be computed at some time after the creation of the object, and therefore it would only be a waste of resources to require that the object have a value at all times. Object values, when they exist, come in two degrees of permanence, making the corresponding objects mutable or immutable. An immutable object can be assigned a value at most once, whereas a mutable object can be assigned a value more than once. This is not meant to imply that either of these necessarily happens, only that the possibility exists. Thirdly, every object is of exactly one type for its whole life. Type is a description of those characteristics that a group of objects have in common, a set of rules by which the objects and the users of the objects must abide. There exists a type manager or some other mechanism for each type (there may be one instantiation of this per object that may be considered an integral part of the object, or there may be an overseer for all objects of a particular type) that insures that only certain operations can be performed on the object(s) being maintained by it. The name and type managers need not be one and the same. Except for the most primitive types, every type is defined in terms of other types, and the representation of every type is in terms of the representations of other types. The types that are not basic types are known as extended types.

Copy operations

As described above, an object has three attributes, name, value, and type. To copy an object the type and value are passed to the receiver of the copy. The receiver must have a type manager for that type, although its implementation and the representation of the objects it manages may both be different from the sending machine. The object receiving the value sent must be of the same type, but will have a different name. (In comparison, when an object is moved, the name must go with it also.) There are two types of copy operations as defined in CLU[3]. As Saltzer discusses, an object contains an object when it names that object. One way to copy an object is to copy only the top level of this structure, along with the names it contains, and devise a mechanism to resolve the contained names if necessary. If the complete structure of an object is large, it may be reasonable to copy only part of it, at least until the user of the information can decide whether the complete object, or possibly just some other parts of it, are needed. For example, it may be that the objects that are named by the copied object, are only for use in exception handling, and the exceptions are very rare, so there is no point in passing those pieces of the structure until they are needed. A second approach to copying an object is to copy the object along with all the objects that it contains by name, plus all the objects they contain by name, etc., until the complete structure has been copied.

Distribution

Now consider a different situation. There are a number of computers connected by a communications network. They are to appear to comprise a single coherent system. This does not necessarily mean that the user perceives the accessing of information on his own machine to be the same as accessing information on other machines. Assume that the machines are autonomous in their object naming, although they all make use of the mechanism previously described.

We still have the goals of modularity and sharing; however, these take on different meanings, when users of objects are on different machines. Sharing in the centralized facility meant actually using the same object, while modularity implied the ability to overlap name spaces, because the contexts used to resolve names were themselves uniquely named. In the distributed situation, the machines are allowed a great deal of autonomy. The implication for sharing is that if the user on machine A wants to operate on data on machine B, there are three alternative scenarios. First, a request can be sent to machine B to perform the operation. Second, the information can be sent or copied to machine A. Or, finally, the request and the information can be sent to a third site. All three require the ability to copy across machine boundaries. Since the assumption has been made that everything can be considered an object, whether the procedure is sent to the data or the data is sent to the procedure, or they rendezvous somewhere else, some copying must take place. By copying a value from one site to another and not moving an object, the problem that the lowest level name spaces (the names not needing further resolution) conflict is solved.

There is a third goal that should be added to modularity and sharing; this is uniformity of mechanism. We will be examining the two types of copy operations discussed in the preceding section within a distributed system. Uniformity in the mechanism supporting these will allow for a simpler generalization of the two into a general copy operation. It also will allow for an initial indication of a copy at only the highest level, to be expanded later into a full copy, without having to repeat work already performed. In order to achieve one level copying of the sort described previously, there must be several changes to the naming mechanism. First of all, the name of the context used for name resolution by the object being copied must be sent along with the object itself. Because the machines are autonomously naming objects, there now exists the problem that the name spaces for contexts can conflict between machines. When an object is created on the receiving machine to contain the arriving information, the closure for this new object must indicate that the context is foreign. There are several ways to achieve this. A new context with a local name could be used. This new context would do nothing except point to the foreign context. Another alternative would be to allow entries in the closure table that specify not only the context associated with an object, but also the home machine of that context. A third alternative is to create a context table. This indicates for each context how to find the context. Thus for the foreign context, a local context name would be used in the closure. When any context is invoked, the invocation passes through the context table, and thus the context for the object that is a copy of a foreign object is discovered to be foreign. The first choice, that of creating a new context local to the new machine and pointing to the foreign context on its machine, is the one that will be used here. The reason for this is that it will allow for more uniformity in the mechanisms used for the two copy operations, as will be shown later.

In addition to the problems of naming foreign objects and contexts, there is yet another level of the naming problem and solution. Now each machine must be able to name all the other machines that are of interest to it. This may be in the form of absolute addresses or unique names that are known to and used by the community of machines as a whole. On the other hand it is possible that all the naming of machines is relative to where the naming is occurring. This alternative has not been explored in depth. Thus, the assumption will be made that within the distributed system each machine will have a unique name that is known and used by all.

To conclude, for the single level copy the mechanism will be enhanced as follows: when an object is copied from one machine to another it must be accompanied by its context name on the original machine. In the new machine a new context will be created that will do nothing but point to the old, foreign context. A new closure table entry must be created for the new object-context pair. Figure 2 presents an example of this.

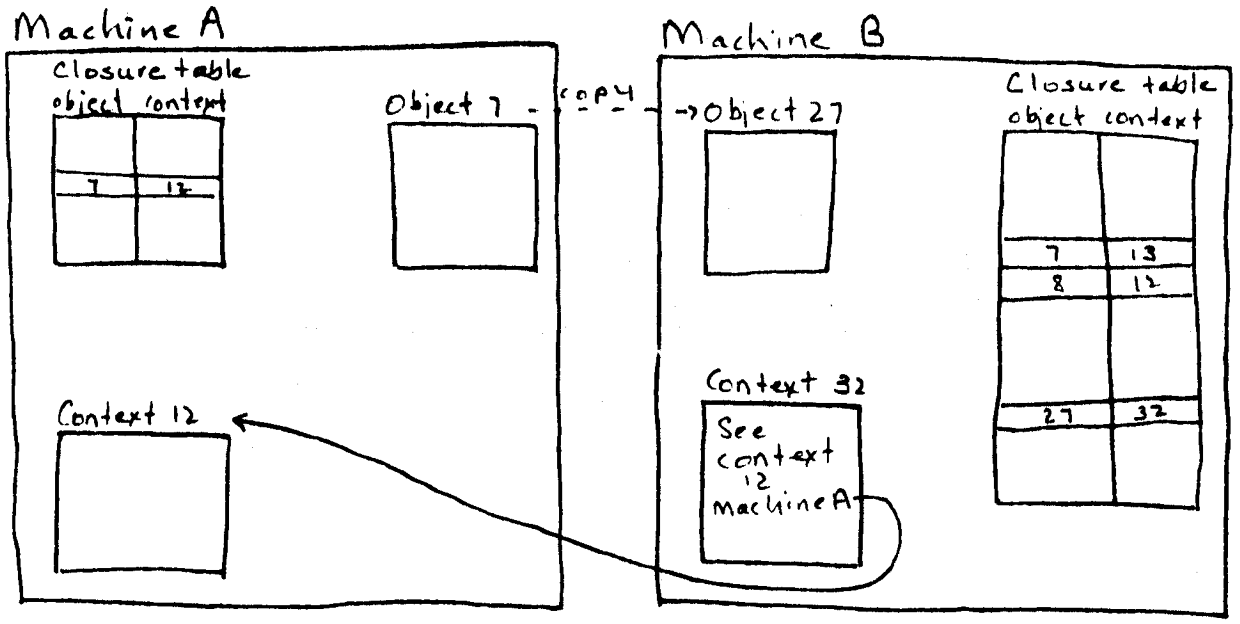

Figure 2

Object 7 on machine A is being copied to object 27 on machine B. Object 7 on machine A names other objects relative to context 12 on machine A. This is reflected in the closure tables on the two machines. Machine B also has an object 7 which performs its name resolution relative to context 13 and a context 12 which is used by object 8, but no conflict can arise.

We now must consider the other type of copy operation, which involves copying the complete structure of an object from one machine to another. In this case, since all relevant information is being copied to the new site, all the names used must also reflect this move. The final names that need no further resolution must refer to the new objects on the new machine.

A mechanism for this type of copying problem is as follows: the complete structure is prepared for sending to the new site by the source machine of the object. All objects that are not closure tables or contexts will contain only names that are to be resolved relative to a context or will contain values. Preparation of an object involves creating, if they do not exist already, a context for each object to be sent that resolves nothing but names that are relevant to that object. Actually, there can be a little saving here; if two objects being copied use the same context on the original machine, there is no reason that they cannot do the same on the new one. Thus in the preparation stage, one context can be created for both. Once these contexts exist, the actual object structure accompanied by these contexts can be sent to the new machine. Now the receiving machine has the problem that the name spaces for both objects and contexts may overlap with those of the original machine. Besides, as the information is passed, it is placed into new objects at the receiving end. What must happen is that at the receiving end, the information sent must be pulled apart, and all the pieces put into appropriate objects. Contexts must be recreated to fit the new situation. The local names from which translation occurs must remain the same (otherwise, the non-context objects will need to be modified).[1] If the name into which a name is being translated is to be further resolved relative to yet another context, that name will also not change. If the name requires no further resolution, it will change to reflect the new objects containing the information being copied. Because the name spaces for contexts on the two machines can overlap, and the machines are copying information not the objects themselves, new contexts will be created for closures for the objects. Thus, it will be these new context names and new object names that will appear in the relevant closure tables. Consider copying the structure depicted in Figure 3 from one machine to another.

1. Modification by renaming may cause problems if, for instance, at a higher level some reliability is being achieved by redundancy. It may be important to be able to compare the actual contents or at least recreate from the new object, the state of the original object.

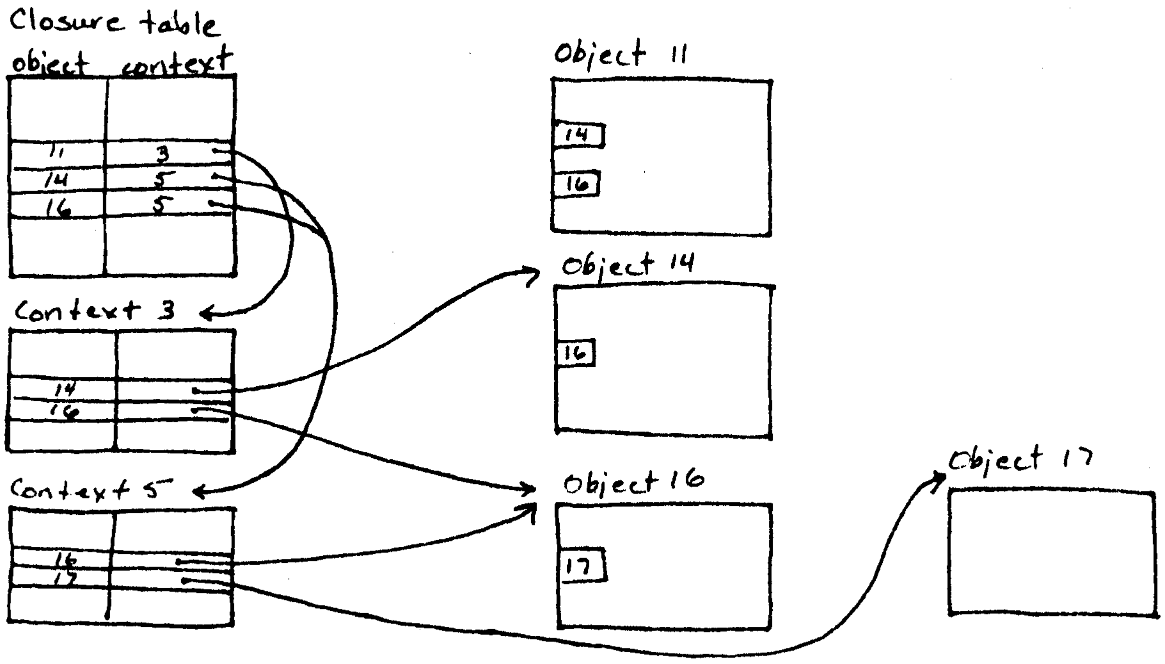

Figure 3

For simplicity it will be assumed that there is one closure table for each machine. Preparation will be done, by creating the temporary structure of Figure 4 to be copied to the receiving machine. Figure 5 shows the new structure on machine B.

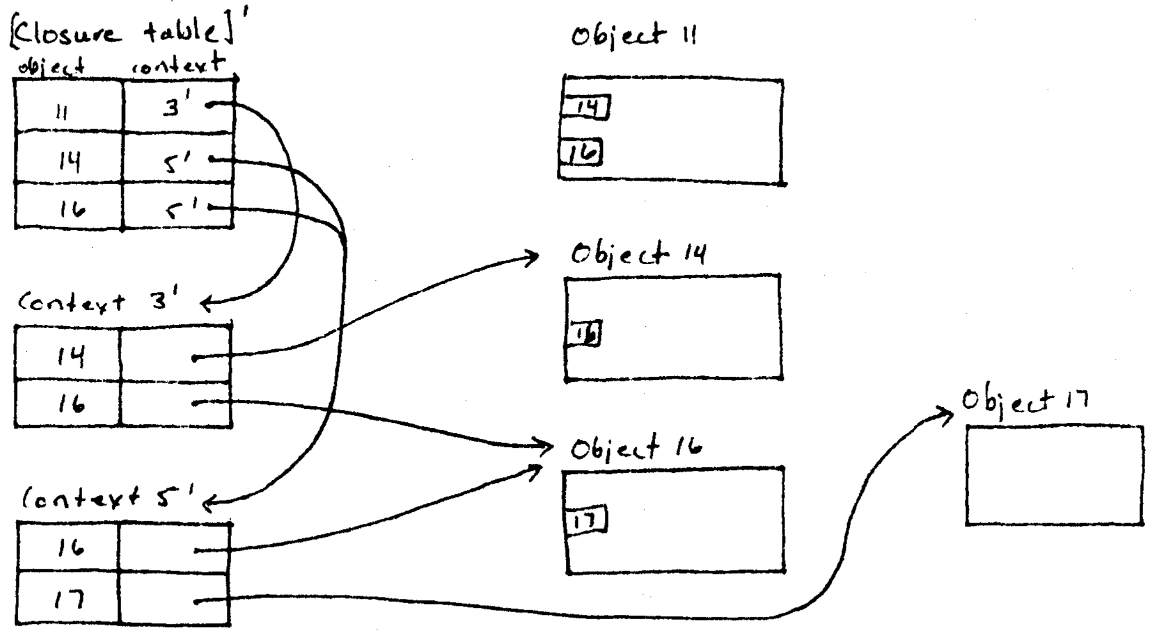

Figure 4

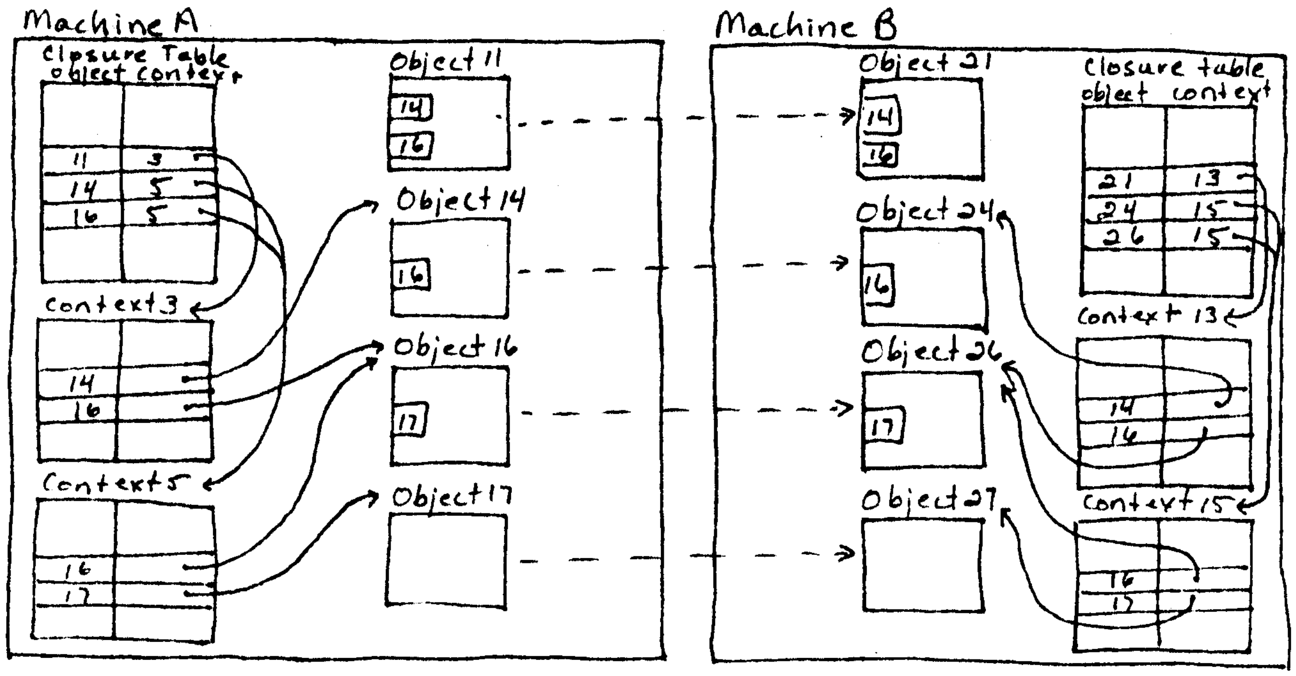

Figure 5

The dashed lines indicate from which objects on machine A, the objects on machine B were derived. Object 17 on machine A and object 27 on machine B are terminal in that they contain no names to be resolved. Nor do the contexts contain names to be further resolved. Neither of these have to be the case; in order that the name resolution procedure halt, there must be names that need no further resolution. On the other hand, in order to allow for recursive objects at the lowest level, the lowest level objects may actually contain names that will need resolution as well as values or states. This concludes the description of the basic extension to contexts.

Proposed Work

There are two specific problems that this thesis will consider. The first is an in depth study of the protocols and contents of the messages passed to achieve the two copy operations, the single level copy and the full copy. An initial approach to these has been indicated above. A partial list of the problems related to this and still to be solved follows.

(1) There is the problem of garbage collection of objects that are contained in objects that have been copied to other machines when only a single level copy has been done.

(2) It must be decided exactly what should be sent between two machines when a copy is made. There is a question of what should be optimized. Should the minimal amount of information necessary be sent, or would it be wiser to send more and thereby avoid some work at either or both ends? There is also a problem related to what information is sent when a copy is made of a copy. A decision must be made about whether the copy of a copy refers back to the original of the object, or whether it refers to the first copy.

(3) How explicit the differences should be between local copy operations and distant copy operation must be determined.

(4) For the full copy operation there is a problem caused by recursively defined objects. There must be a mechanism to keep the system from sending the same object more than once.

(5) Finally, there must be a specification of exactly what procedure is to be used to find the set of names relevant to an object. Part of the protocol mentioned earlier must find this set of names. So far, the protocol has not indicated how this is to be done has not been indicated here.

As other issues of detail arise they also must be addressed.

The other problem to be addressed in this thesis is the generalization of the copy operations into one copy operation for which the user of the operation can specify how much of the structure being copied should be copied. For this approach to be useful, it is necessary to have the means to specify in some convenient way which parts should be copied. Given a layered structure for the definition of an object, it must be decided whether or not it is feasible to specify a copy operation that will be a full copy operation except that at some level it will copy only part of that layer, and then will continue with the full copy.

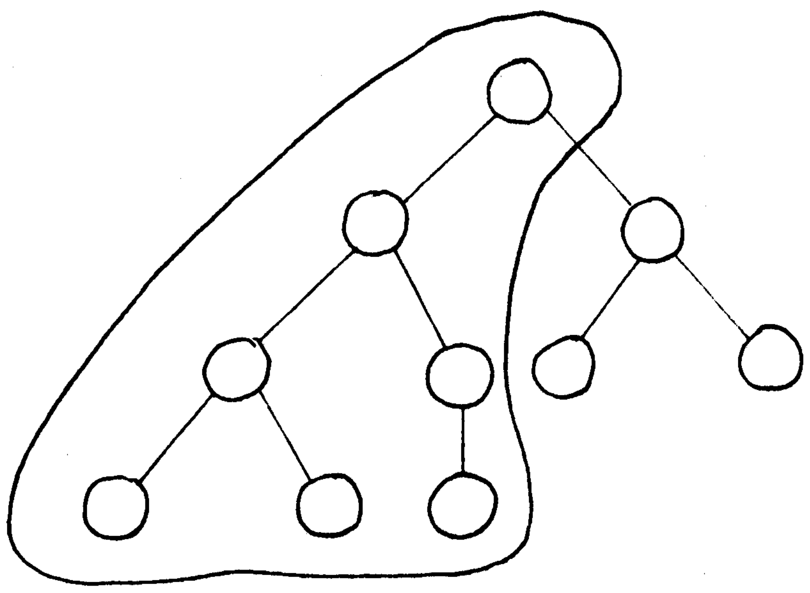

Figure 6

Figure 6 depicts an example using a hierarchical structure for the object. The circled part of the structure is to be copied. Judgment of the form and utility of such an operation is much more subjective than support for the two copy operations discussed above, but may be helpful is deciding in the future whether or not such a general copy operation ought to be provided by systems.

Related Problems

There are a number of related problems, some of which depend on the ability to copy. For instance, providing reliability by redundancy of information at different nodes in the distributed system may be desirable. Given a copy operation and synchronization mechanism, redundancy can be provided by the subsystem or applications programmer for higher level users. Another interesting problem is how to move an object. Again, the copy operation is crucial. In addition there also must be some mechanism forwarding requests that arrive for the moved object at its old home to its new one. A third interesting issue is protection of information; this is a problem that grows with the ability to copy. As long as information remains on the machine used exclusively by the owner of the information, the machine boundary can be used to enhance protection. As soon as information is copied to another site out of the control of the owner a much more complex protection mechanism may be desirable or necessary. These are some of the interesting related problems; I will not present solutions to them since they are not included in the main research of this thesis, although they have arisen either in discussion or in the literature.[1]

Schedule

The following schedule is proposed for this thesis research:

July 1978	complete literature search
	begin developing mechanisms for single and full copy operations
August 1978	complete development of mechanisms for single and full copy operations
	develop specifications for general copy operation

1. Moving information comes up frequently in discussions within the Computer Systems Research Group and the new Distributed Systems Group, Laboratory for Computer Science, M.I.T. Reliability through redundancy has been discussed by Thomas[8]. Protection has been discussed by, among others, Saltzer[5].

begin writing first draft of thesis

September 1978 complete first draft of thesis

revise thesis as required

October 1978 - complete thesis

Resource requirements

Computer time on the MIT Information Processing Center's Multics system may be required for small experiments with some of the mechanisms and protocols developed. Computer time in the form of text preparation facilities will be needed for this thesis.

References

[1] Halstead, R.H., Multiple Processor Implementations of Message-Passing Systems. S.M. Thesis, M.I.T. Department of Electrical Engineering and Computer Science. January, 1978.

[2] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," *CACM 20*, 8 (August 1977), pp. 564-576.

[3] Liskov, B.H., et al., "The CLU Reference Manual," CSG Memo # 161, M.I.T. Laboratory for Computer Science, July, 1978.

[4] Luniewski, A.W., private communication, May, 1977.

[5] Saltzer, J.H., and Schroeder, M.D. "The Protection of Information in Computer Systems," *Proceedings of the IEEE, 63*, 9 (September 1975), pp.1278-1308.

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

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

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

[9] Wulf, W.A., Levin, R. and Pierson, C., "Overview of the Hydra operating system development", *Proc. Fifth Symposium on Operating Systems Principles*, 131, November 1975.