M.I.T. Laboratory for Computer Science — April 28, 1979

Computer Systems Research Division — Request for Comments No. 171

Copying Complex Structures in a Distributed System

Karen R. Sollins

Attached is the paper I have submitted to the 7th SOSP.

Copying Complex Structures in a Distributed System

Karen R. Sollins

Laboratory for Computer Science
Massachusetts Institute for Technology
545 Technology Square
Cambridge, Massachusetts 02139

April 27, 1979

This research was supported 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.

Copying Complex Structures in a Distributed System

ABSTRACT

This paper presents a model of a distributed system where the universe of objects in the distributed system is divided into mutually exclusive sets, each set corresponding to a context. This model allows naming beyond the context boundaries, but limits communications across such boundaries to message passing only. Copying of complex data structures is investigated in this model, and semantics and algorithms are presented for three candidate copy operations. Of particular interest is a new operation copy-full-local which copies a complex data structure to the boundaries of the context containing the object.

Key words and phrases: copying, sharing, distributed system, message passing, strongly typed objects.

Copying Complex Structures in a Distributed System

1. Introduction

Many aspects of computing are based on the ability to copy information. The foremost of these is parameter passing by value; in distributed systems, it is the only way to pass parameters between program modules executing at different nodes. Since these parameters may be abstract objects whose actual representations are complex data structures, copying in this kind of environment is a non-trivial matter. The second area is a more general sharing where copies of some objects will be maintained at several nodes. Of course, if the object is mutable, it brings up immediately the question of mutual consistency of these copies; however, this is not the subject of this paper. Finally, copying is needed to move an object from one location to another; this is different from the previous, in that after an object is moved, there is still only one instance of the object in the system. Each of these and possibly other areas require the ability to copy. Each also requires other mechanisms, which have, in general, been topics of research. This paper concentrates only on copying, in particular copying complex data structures[10].

In addition to the problems for which copying is a part of the solution, there are a number of interesting problems that must be addressed in developing semantics and algorithms for copying. For example, consider the situation in which a structured object is being copied. Of interest here are those components that are contained by naming in more than one other component, in other words shared by other component objects. A decision must be made as to whether or not those shared components are copied only once, once for each containing object, or once for each pointer to the object. Another question that must be answered is whether or not more than one kind of copy operation is needed, and, if so, what the semantics of the different operations are. In order to address these problems, a model is necessary. This paper will begin with a model of a distributed system which combines aspects of recent work on naming objects, distributed systems, and strongly type object based systems. The paper will then return to the issues surrounding copying, and, finally, present a solution to them consisting of the semantics of three copy operations, and algorithms describing how they might be implemented. In particular, the paper will present a new copy operation, copy-full-local which copies a complex data structure to the boundaries of the domain containing it. The model will allow naming beyond such domain boundaries but the communication between such domains is only by message passing. For a more detailed treatment of this subject, the readers are referred to the full report.

2. The Distributed System

The model of a distributed system used in this research has been influenced strongly by the work of Saltzer[9], Liskov et al.[4,5], and Svobodova et al.[11] In Saltzer's work every object is associated with a context or naming environment; all the names or pointers in an object are resolved with respect to the context specified for that object. The purpose of contexts in Saltzer's work is to achieve what he terms modular sharing. A number of ideas from the work in CLU of Liskov et al.[4,5][1] have influenced this work. First, the work on CLU presents a strong justification for abstractions or strongly typed objects and type extension. Second, the CLU syntax and approach to modularity in programming has provided a basis for implementation of a number of the most important procedures for this research; most of these appear in the full report[10]. CLU also provides approaches to the semantics of copying, the copyl and copy operations for arrays and records. Both arrays and records can be complex structures. The copyl copies only the top level of one of these structures, while the copy copies the complete structure. The third strong influence on this research is the work on distributed systems of Svobodova et al.[11] The model of a distributed system in that work assumes guardians communicating only by message passing. The universe of entities in this model is divided into two kinds of entities, active, which are called processes, and static, called objects. A guardian is composed of one or more processes and the local address space (the directly accessible objects) of those processes. The local address spaces of guardians are mutually exclusive sets of objects. A process or object can refer directly only to objects within the same guardian. Across guardian boundaries only processes may be named.

The model of a distributed system used in the research reported in this paper contains aspects of the models mentioned above. The model assumes the hardware of the system to be a network of computers, each computer having its own private memory or namespace for objects. The single namespace in a

1. Although the article in Comm. of ACM on CLU[4] provides much of the knowledge of CLU used for this paper, it does not contain all the details. For those the only source is "The CLU Reference Manual"[5].

computer provides neither enough flexibility in naming objects nor enough protection in accessing objects. Thus, this work first develops a model of a context that facilitates finer partitioning of the namespace, and takes into account the other works mentioned above.

Each computer or node in the distributed system supports one or more contexts. As with guardians, the universe of objects on a node form disjoint sets, each set corresponding to a single context. Thus the context defines the local private memory or namespace. In order to provide flexible control of sharing and to limit error propagation, the only means of communication between contexts is by passing messages. This constraint allows enforcement of arbitrary degrees of protection at the context boundaries. It does not eliminate the possibility of sharing an object across context boundaries, but does limit the means of access to that object; if an object is known beyond the boundary of its local context, the only means of operating on the object is by passing the name of such a foreign object in a message requesting that some operation be performed on the object in the containing context. The user will see a collection of contexts with messages flowing between them.

In line with the work on abstractions in CLU, the model assumes strong typing for objects and supports extended types. Thus, every object has one type for its lifetime; there are predefined base types provided by the system, with a provision for creation of extended types composed of both base and extended types. A base type object contains a value, and an extended type object contains a list of names of component objects. Such an extended type object contains only names local to the context in which it resides. The context translates these names into one of two different kinds of names: a low level name for a local object (this is discussed further, below), and a globally unique name for a foreign object. In order to contain a foreign object the containing object uses a local name, and the context provides a translation into a name that is unique throughout the whole distributed system. Since contexts are assumed to be uniquely named throughout the system, globally unique naming can be achieved by combining context and object name.

There must be supporting mechanisms for this model of contexts. For the purposes of this paper, only the message and storage handlers are of concern. The message handler must be able to (1) pass messages between contexts local to a single computer, (2) pass messages from a local context out into the network, and (3) receive messages and see that they are delivered to the correct local context. The message handler transforms messages passed between contexts into the kinds of messages that can be passed through the network hardware. The message handler contains information about low level protocols. It is quite possible that the low level messages of the network do not correspond to the high level message objects (or <u>images</u>, as will be presented below) discussed in this paper. These high level messages may be buffered and sent in groups, or split into smaller packets. Whatever is done by the message handler at such a low level is hidden from the contexts and users. The storage handler, as its name indicates, oversees storage of objects. For each object stored in the node, it provides a unique name in order that the physical object may be accessed (through the storage handler). Each storage name is known to a single context and associated with the local name assigned to that object by that context.

There are three complementary views of the context. First, as it was initially presented, it appears to the user to be a namespace. A context is an environment in which local objects exist and can name each other using only names local to the context in which they reside. Second, an extension of this view leads to classifying contexts as virtual nodes in an abstract network, where the nodes can communicate only by sending messages. Third, contexts also can be viewed as typed objects; a context has state and a name and can be considered to be of type context. It is useful to consider contexts to be objects at times and as such to make use of the operations defined for the type context; this is more apparent in the implementations presented in the full length report on this work [10] than in this paper. The view of contexts taken in this work is a combination of the three views.

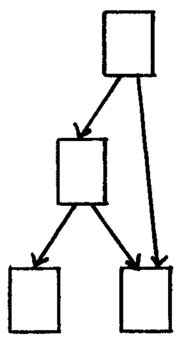
In consideration of the model of a distributed system that has been presented here, it appears that there are a number of issues to be addressed in copying objects between contexts. These will be discussed in the following section. Within a context, copying appears to be less difficult, and if it can be achieved across context boundaries, it certainly can be achieved within them. Therefore, this paper concentrates on copying between contexts.

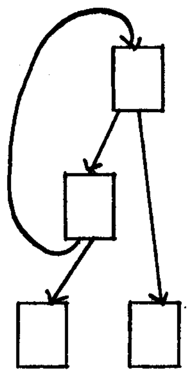
3. Issues and Goals in Copying

As mentioned in the introduction to this paper, there are a number of issues related to copying that must be considered before deciding on the semantics of copying. The most important in this research is the question of whether or not to maintain sharing of component objects. Although a more common concern is sharing among processes or users, this research concentrates on sharing among objects. In the model assumed for this research, objects can have arbitrarily complex structures; they also can be contained recursively.

The simplest question is whether maintenance of sharing would be necessary in copying objects if recursion were not allowed, but sharing components were, as in Figure 1(a). If sharing does not occur in a copy where it does in the original, the behavior of the copy may be different from the behavior of the original object under the same conditions. Now, considering the more complex structure that includes recursive containment of components such as the structure in Figure 1(b), it becomes even clearer that such sharing must be copied in order to terminate a copy operation which copies the complete structure. Finally, the most complex situation is that in which such sharing of components occurs across context boundaries, as in Figure 1(c). In this case, a new dimension has been added to copying, because of the limitation of communicating only by message passing. In all three situations, depending on the means of guaranteeing a consistent copy of the data, deadlock is possible. In the third case, such a deadlock would involve more than one context. If the structure were part of distributed data base, with components of the structure in many contexts, the problems of copying without regard to sharing in the structure appear to be unmanageable. Therefore, the primary goal of the copying described in this paper is to maintain any sharing that exists in the object being copied.

Closely related to the question of sharing is the question of exactly what the semantics of copying are to be. The standard meaning of copying is to create at another location another version of the object being copied, the copy having the same behavior as the original. Now the question is what is meant by "the same behavior". CLU provides two answers to this, as previously mentioned. The *copy* operation copies the complete structure, although sharing relations are not maintained. The other copy operation provided by CLU is *copyl*. This operation copies only the top level of the structure, copying pointers to all the components of the original. In fact, the *copy* operation is defined by calling *copyl* on the original object, and then calling *copyl* for each component, moving through the structure until all the components have


(a) Non-recursive sharing


(b) Recursive sharing

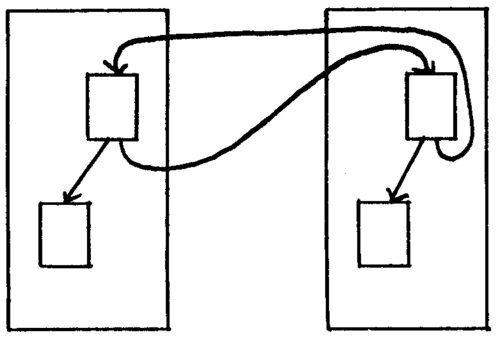

(c) Recursive sharing across context boundaries.

Figure 1 Examples of sharing of components within a data structure.

been copied. Copy provides the standard semantics for copy by copying all of the object, and copyl allows for creation of specially tailored copying, in which not all the components are to be copied. In this research the operations similar to copyl and copy are copy-one and copy-full.

The model of the system presented in this paper is much more complex than that of CLU, and, as mentioned previously, data structures can be more complex. As a result, this research has led to a third kind of copy operation: the copy-full-local. The copy-full-local operation copies to the boundary of the context containing the original object. Figure 2 is an example of this. Only those components directly connected to the top level of the structure and in the original context are copied. This copy operation complements the other two in such a way that the three provide the user with a great deal of flexibility in copying complex data structures across context boundaries.

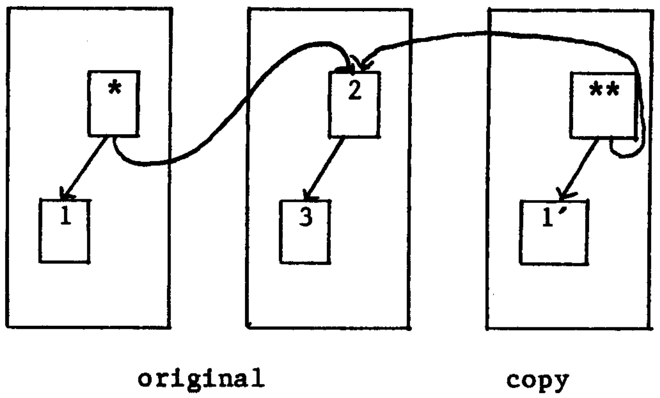


Figure 2 An example of the copy-full-local operation. The object labelled with * is copied into the object labelled with **, and the components labelled 1 is copied into 1′. The component labelled 2 and 3 are not copied.

It must be pointed out that a variety of copying algorithms have been developed by other people. These include those developed simply as copying algorithms (for example both Clark [2] and Fisher [3]) and those with particular functions in mind such as garbage collection (for example McCarthy [6,7], and Baker [1]). Although these works must be considered, they present a common problem. They all use the copy that is being created as part of the workspace needed to generate the copy. If copying is to be performed across context boundaries, such use of the copy implies increased message passing. Because of the cost in time and greater possibility of failure due to the need for cooperation between contexts, this paper presents an alternative approach that avoids these problems.

A final point is that the model assumes objects of arbitrary structure; this includes arbitrary size. Since objects can be very large, it may be impossible to create an image of the complete structure of an object before sending the pieces to the receiver. The receiver may have similar space limitations. Therefore it should be possible to process the images of components separately.

In light of the discussion of this section, four goals are set for developing algorithms and implementations of the three copy operations: (1) any sharing that exists in the original structure must be maintained; (2) economy of mechanism by using a single approach in all three operations is desirable; (3) since all communication between contexts is by message passing, the amount of message passing necessary should be limited; (4) it should be possible to send and receive component images separately.

4. Copying

This section presents the algorithm for achieving copying by sending and receiving images, followed by an example of performing a copy-full on an object that contains components in two contexts. The purpose of this example is provide a clearer picture of the mechanism needed to achieve the copy operations described in the previous section. The succeeding section will discuss the implementation and indicate the simplicity of including copying when implementing a type manager or cluster.[1] The procedure for sending a copy of an object to another context is similar for all three copy operations. When it has been decided that an object is to be copied, the first step is to create a message-context in the sending context. A message-context is an object that is growable and has only a short lifetime. It is a mapping between the indices of its entries and the values of those entries. An entry is created as follows: each name in the original object is examined to find its full name, {context name, local name} pair. This becomes an entry in the message-context if it is not there already. A message-context is a mapping between indices and full names. The entry associated with index 0 is the full name of the top level object being copied. Once the message-context exists, an image of the object is created. An image contains the indices of those entries in the message-context containing the full names corresponding to the local names of the components of the original object. The header of the image of each component includes the index of the entry in the message-context that contains the full name of the corresponding object. The header also contains

1. The term type manager is used throughout this paper to be synonymous with the CLU term cluster.

the type of the object from which the image was created. When an entry for the original object has been made in the message-context and an image of it has been created, the image is ready to be sent. At this point an image of the next object named in the message-context that is to be copied is created in the same manner as the top level object using the same message-context, thus adding entries to the end of the message-context when necessary. This is repeated until an image has been created and sent for every object named in the message-context that is to be copied. For a copy-one operation, only the top level object is copied. Once this image has been sent, an image of the message-context must also be sent, in order to create the correct entries in the receiving context for the names in the object being copied. For a copy-full, once images for all the components have been created and sent, nothing more needs to be sent. The message-context is of no more use. Finally, for a copy-full-local operation, all the components that are in the sending context will be copied, and a partial image of the message-context containing the indices and entries for the foreign components must be sent.

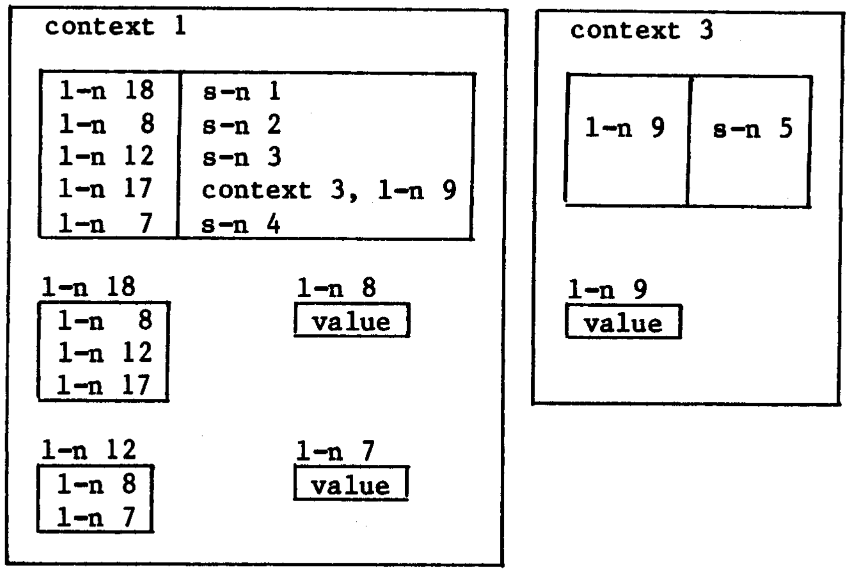
As indicated above, the image created for each object copied will have a two part header. One part is the index of the object's name in the message-context. This would not be necessary if we could guarantee that all messages would be received in the same order they were sent; however, such an assumption is unnecessary.[1] The other part of the header is the type of the particular object to which the header is attached. Again this would not be necessary in most cases assuming that messages were received in the order

1. This assumption would put additional burden on the lower level protocols, and since the overhead of sending the index is low, such an assumption is not considered necessary. It also would limit the usefulness of spraying messages (in particular images) down multiple paths.

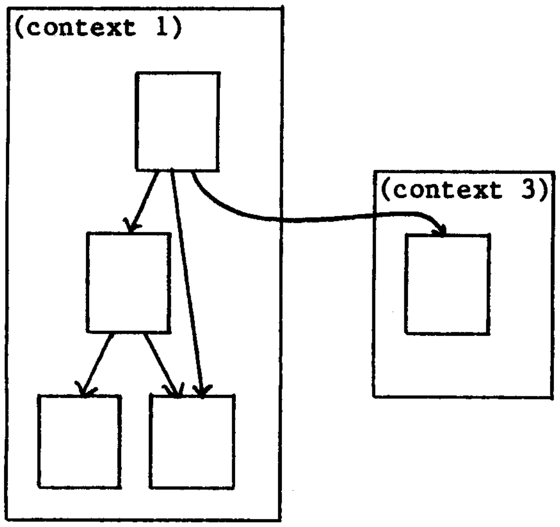
sent, because if the order of arrival were predictable and the types of the components already known, as the images arrived their types would be known. However, if the receiver is expecting an object of the CLU type <u>any</u>), the arriving image had better have its type attached to it, in order that the receiver can invoke the correct type manager. In addition, the components of an image header provide redundancy that can be used for reliability.

For a better understanding of the algorithm, an example of the copy-full operation follows. For examples of all three operations, see the full report on this research[10]. The copy-full operation is the most encompassing of the three copy operations, and as such uncovers problems not encountered with the other two. First, the problems associated with shared components appear. (This is also a problem in the copy-full-local operation, although it is not a problem in the copy-one.) The message-context maintains all such sharing. The second consideration is the problem of handling foreign components. (This is not a problem in either of the other operations.) In the case of the copy-full operation, there are problems associated with acquiring a copy of a foreign component, as well as sharing components across context boundaries. In order to maintain such sharing across context boundaries, a copy-one operation should be performed on any foreign component. This means that the sending context will acquire only the top level of the foreign component plus the names it uses. By this means the message-context will discover all sharing, even that involving foreign components.

The object to be copied in this example is depicted in Figure 3. For the remainder of this discussion the term <u>l-n</u> is used for local name and <u>s-n</u> is used for storage name. Also, contexts are depicted as containing objects, as



(a) The names in an object, its components, and the relevant contexts.



(b) Block diagram of the structure of the object l-n 18 of (a)

Figure 3 An example of an object. The abbreviation l-n is used for local name and s-n for storage name. A context here is represented as a node in an abstract network, containing objects, and as a namespace containing a mapping from local names to either storage names or globally unique names. Those objects containing local names are of extended type, and those containing the word value are of base type.

a node of an abstract network would, while at the same time providing name translation as a namespace would. In the example, the object l-n 18 will be copied using the copy-full operation from context 1 into another context, context 5. Figure 4 depicts the message-context and images created in the sending context while performing the copy-full. The message-context is created with an entry for {context 1, l-n 18}. In context 1, l-n 8 is first looked up and found to be local to that context. Hence its full name is {context 1, l-n 8}. This entry is created in the message-context and since it has index 1, a 1 is put into the first position in the image of l-n 18 being created for sending. Then the full name is found for l-n 12 in context 1, and, since it is not already in the message-context, a second entry is made, and another index is put into the image. Now, when l-n 17 is followed, rather than a storage name in the context, there is another {context name, local name} pair. This, then, is used as the full name for the entry in the

---

message-context

0	context 1, l-n 18
1	context 1, l-n 8
2	context 1, l-n 12
3	context 3, l-n 9
4	context 1, l-n 7

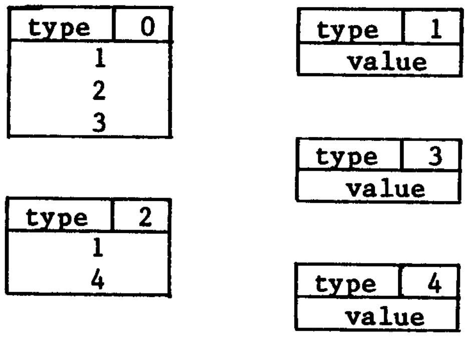


Figure 4 For the copy-full operation images 0, 1, 2, 3, and 4 will be sent, but no image of the message-context need be sent in copying {context1, l-n 18} of Figure 3. Again, l-n is an abbreviation for local name.

message-context in the same way as the other full names. Once this has been completed, and the image has a header containing the type and index 0, the image can be sent. Now, the next entry in the message-context, {context 1, l-n 8}, is considered and an image of that object is created as with the first. It is of a base type, and therefore its image will contain a value. Again, a header will be attached to the image, this time containing the type of this object and an index of 1 (which is the index of its entry in the message-context). Now this image can be shipped. Once an image of l-n 8 has been created, the image of the next entry in the message-context can be created. This next object is {context 1, l-n 12}, which is of an extended type. It contains a list of two names. The first is l-n 8. When the full name is found for this, {context 1, l-n 8}, and it is compared with the entries already in the message-context, it is discovered that there already is an entry for that object. Its index is used in the image of l-n 12, but no new entry is made in the message-context. Now the next name in l-n 12 is handled. It is found to have a full name of {context 1, l-n 7} which is not yet an entry in the message-context, so an entry is created and the index of 4 is used. Once the header containing the type of l-n 12 and an index of 2 have been attached to the image of l-n 12, this step of the operation is complete. The next object to be copied is {context 3, l-n 9}; a copy of this must be acquired from context 3. Once that has been done, an image can be created for this object having in its header of the name of the type of {context 3, l-n 9} and an index of 3. The copy operation from context 3 must be a copy-one, although for an object of base type as in this case, it makes no difference.

There are several issues that need mentioning here. First, context 1 will not keep the copy of {context 3, 1-n 9}. If such a copy were kept in the sending context, the copy-full operation would have permanent side-effects on the sending context; this is clearly undesirable.[1] Second, there may be a problem with acquiring that copy from a foreign context. It will, at least, cause some delay; at worst, it may be impossible, causing the original copy-full to fail. The copy-full-local operation solves this problem.

To resume the example, assume that the copy-one on {context 3, 1-n 9} into context 1 has been completed successfully. The object {context 1, 1-n 7} will be processed next. This is another object of a base type. The value will be copied as with 1-n 8, and the header attached. Now, considering the message-context, it becomes evident that all the objects named in it have been copied and their indices attached to them in their headers. Therefore no part of the message-context needs to be sent to the receiver of the copy, and it is expendable.

Once an image has been sent from a context, the message handler must determine how to find the receiving context. If the receiving context is on the same node, the network need not be involved. The images passed out of the sending context are passed directly to the receiving context. If the receiving context is not on the local node, the message handler must prepare each message for transmission through the network to the correct node.[2] At

1. Of course, copy-full operations will have temporary side-effects when there are foreign components.

2. This work does not deal with the communication protocols of the network, although of course the message handler must know about them. The copy operations can know nothing about these protocols nor about the degree of reliability they provide.

the other end of the transmission, the message handler at the receiving node will receive the messages from the transmission medium and pass them to the receiving context, assuming a receive command is pending in that context. Whether or not the network was involved, it is in the receiving context that the images created by the sending context are used to create the actual copies of objects. The receiving procedures will be presented as a set of cases each to be handled differently, as there are so many possible orderings of the arrivals of the parts of a copy, and processing is to begin as soon as a receive command has been issued and at least one image has arrived. As has been mentioned previously, the order in which the images arrive is unpredictable, and they are processed in whatever order they become available at the receiving site.

Each piece of a copy must be identifiable as part of that copy and labelled with its own type and index if it is a copy of a component or the fact that it is a message-context or a part thereof, if the copy was a copy-one or a copy-full-local. The procedure is as follows.

1. When the first image (component or message-context image) is ready to be processed, a local receiving message-context is created. It will contain, in addition to the index for each object, the local name for that object once that name has been determined.

2. When the message-context image arrives, its entries are processed sequentially. When an entry is processed, the receiving message-context is first checked. If there is a local name there associated with the index of that entry, this local name is used to find the location in the local context to place the full name carried by the message-context image. If there is no local name in the receiving message-context for that entry, the context must find a local name to refer to the foreign object, this entry is created in the local context, and an entry is created in the receiving message-context for the appropriate local name having the appropriate index.

3. When a component image arrives, the receiving message-context is checked for a local-name to be used for the new object. If a

reference to the arriving component has already been received in another image, a local name will have been assigned. If not, one must be requested from the context. Using the appropriate local name, the image is transformed into a copy of the original object. If the object is of a base type, its value is taken from the image. If is of extended type, each name is picked up out of the image. Using this name as an index into the message-context, a look up is done. If either that object's image itself has arrived previously, or another reference to that object has arrived in yet another image, then there already will be an entry in the receiving message-context containing a local name for the reference. This will be used in the copy of the component being created. If there is no local name for the reference yet, the context must provide one. Thus an entry will be created in the receiving message-context, having the appropriate index and the local name provided by the context. Also an entry must be made in the context, although no object will be assigned as yet; i.e., there will be a local name in the context having no other name (either storage or full name) associated with it.

4. Images are received until there are no entries in the context that do not have storage names or full names associated with them. At this point, the copy has been completed and the receiving message-context is no longer needed.

Figure 5 depicts context 5, the receiving context in the example discussed earlier. In the figure all the copies of the components have been created. Also the receiving message-context is complete, but has not yet been deleted.

This section has presented an algorithm for achieving the copy operations and that meets the goals described earlier. This description may leave the impression that these copy operations are too complicated, but the next section will point out how little work is involved in including such operation in a type manager. Most of the work can be done by procedures that can be provided in each context when the context is created.

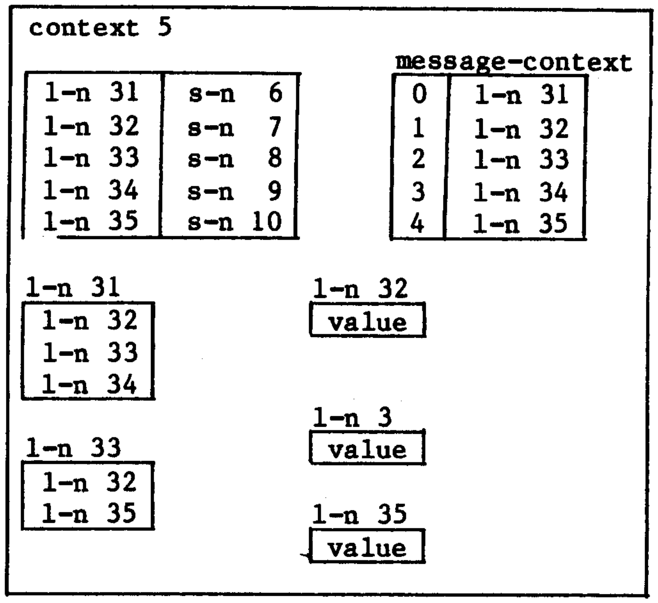


Figure 5 The results in context 5 of a copy-full on {context 1, l-n 18} of Figure 3 to context 5 before the receiving message-context has been deleted. As with previous figures, the context contains a mapping as well as objects. Also, again, l-n is an abbreviation for local name and s-n, for storage name.

5. Implementation

This section presents only an overview of an implementation; a sample implementation of images, message-context and the sending and receiving protocols can be found in the full report [10]. The implementations are in a CLU-like language. The algorithm as described in the previous section is quite involved. Therefore one of the strong forces in developing the implementation was to hide as much as possible from both the user of the copy operations and the creator of new types for which copy operations are to be defined. The functions of the message-context are (1) to find sharing among components, (2) to keep track of which components should be and have been copied, and (3) to provide those references that must be copied in the cases where only a partial copy (copy-one or copy-full-local) is being performed. The user and programmer do not need to know about these functions or the message-contexts that provide them. In contrast, the creation of images should be type dependent. This means that the type implementer must know about images, and must provide a create-image operation for his type. There also must be a create-image for the type of each component to be copied.

As mentioned previously, each context will be equipped with three procedures named generic-copy-one, generic-copy-full, and generic-copy-full-local. In order to define one of the copy operations for a particular type, the programmer simply writes the operations including the create-image operation as has been done for copy-full and create-image in Figure 6. The appropriate generic copy procedure invokes message-context$create[1] passing it the name of the top level object being copied and the name of the copy operation being performed (copy-one, copy-full, or copy-full-local). The message-context is viewed as the essence of the copying of the top level object since it contains the names of the object and the copy operation. Therefore, the generic copy operation next invokes message-context$send to send the message-context. The message-context$send operation oversees the creation of images for the components to be copied by invocation of the create-image operations of the appropriate type managers followed by the image$send operation. When all the components have been copied as determined by the particular kind of copy operation, an image of the message-context is created and sent if that is appropriate. At this point the message-context$send can return, as can the

1. In CLU message-context$create is the invocation of the create operation of the message-context type manager or cluster.

```
copy-full := proc (object-name: foo, receiver-name: any);
generic-copy-full (object-name, receiver-name);
     return ();
     end copy-full;


create-image := proc (object-name: foo);
     image-name: image :=image$create ("foo");
     for component: any in components-of-object-name do
           . . .
          for the name of each component of the object create an entry in
             the image
           . . .
          end;
     return (image-name);
     end create-image;
```

Figure 6 The copy-full and create-image operations of the type foo. The syntax used in these procedures is similar to that of CLU. The type of the receiver of such a copy has not been specified and therefore any has been used for it.

generic copy procedure invoked, and finally the original copy operation. In performing a copy-full, a foreign component may need copying. In this case, a copy-one of the foreign component must be invoked through the message-passing facility. In order to receive, similar sorts of procedures must be provided. It should also be mentioned that it is trivial to modify the above procedures to include also local copying within a context, with no extra burden on the programmer.

An interesting result of the requirement of retaining sharing relations in copying is that copy-full, unlike copy of CLU, cannot be composed of multiple calls on copy-one (copyl in the case of CLU). The reason for this is that each invocations of copy-one created a new message-context and it is the single message-context of copy-full that achieves the retention of sharing. This point is more apparent in the full report[10] than in this paper.

Although no reference has been made to it yet, the operations discussed above depend on a synchronization mechanism in order to guarantee that a copy is consistent within itself. If a large object is being copied using a copy-full or copy-full-local operation and more than one process is running in the sending context, there must be some form of guarantee that components do not change during the copy operation. The apparently simplest approach is locking, but this immediately raises up the specter of locking a whole or large part of a database. It also requires an extra traversal of the structure. A much more serious problem is deadlock; there has been work on deadlock detection and avoidance at a single site, but the problem becomes quite costly with attempting to lock foreign components in the case of a copy-full operation. An approach developed by Reed[8] appears to provide a better solution to this problem. Reed proposes that when mutable objects are modified, new versions of them are created and time-stamped. Thus, as long as the older versions are saved, it is possible to refer to and use a consistent version of the object. This idea solves the problem of deadlock as well as that of making an additional pass of the structure.

6. Conclusions

In conclusion, this paper presents two major points. The first, and most important, is the new copy operation, copy-full-local. This operation copies a complex data object to the boundary of the context or domain containing that object. In the situation in which the universe of objects is divided into mutually exclusive sets or contexts and where the only means of communication between those contexts is message passing, the copy-full-local operation complements the semantics of the two kinds of copy operations provided in CLU. One of these operations copies only the top level of a structured object, and the other, the complete structure. The copy-full-local operation lies between these two extremes in function.

The second conclusion to be drawn from this research is that most of the mechanism to support the three kinds of copy operation can be embedded in the contexts by providing three generic copy operations. They can be invoked directly by the creator of a type with an argument of the names of the object to be copied and the intended receiver. The only requirement is that there must be a create-image which will create an image of an object (this being the only base type capable of being sent) must be defined for the type of every component (including that of the original object) being copied.

There are many issues related to this research that have not been addressed in this paper. A partial list follows:
(1) Exception handling must be an integral part of operations such as the copy operations presented here. Message-context can be an aid in backing out of a copy operation that fails, although there may be a problem finding images used in copying in both the sending and receiving contexts.
(2) In order to achieve the operations that perform partial copying (copy-one and copy-full-local) the types of all objects to be copied must have the same representation in both contexts.
(3) Globally unique naming of contexts and the types involved in copying is an assumption in this research. Whether this is feasible and reasonable needs further investigation.

(4) Specializing copying to different types needs attention. This can be achieved in the type specific create-image operations. Whether or not it is useful must be considered.

These and other points related to this work are addressed in greater depth (although not necessarily solved) in the full report on this work[10]. In particular, the most interesting of these and a topic of much current and probably future research is the issue of exception handling, which is closely related to the work on atomic transactions.

References

[1] Baker, H.G. Jr., "List Processing in Real Time on a Serial Computer", Comm. of ACM 21, 4 (April 1978), pp. 280-294.

[2] Clark, D.W., "List Structure: Measurements, Algorithms, and Encodings," Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pa., August 1976.

[3] Fisher, D.A., "Copying Cyclic List Structures in Linear Time Using Bounded Workspace," Comm. of ACM, 18, 5 (May 1975), pp.251-252.

[4] Liskov, B.H., et al., "Abstraction Mechanisms in CLU," Comm. of ACM 20, 8 (August 1977), pp. 564-576.

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

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

[7] McCarthy, J., "Recursive Functions of Symbolic Expressions and Their Computation by Machine", Comm. of ACM 3, 4 (April 1960), pp.184-195.

[8] Reed, D.P. "Naming and Synchronization in a Decentralized Computer System", M.I.T. Laboratory for Computer Science Technical Report TR-205, Cambridge, Mass., September, 1978. (Also Ph.D. thesis, Department of Electrical Engineering and Computer Science, M.I.T., September, 1978.)

[9] Saltzer, J.H., "Naming and Binding of Objects," Lecture Notes in Computer Science 60 (Ch. 3), Springer Verlag, New York, 1978, pp. 99-208.

[10] Sollins, K.R., "Information Sharing in a Distributed System," M.S. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. to be completed.

[11] Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems: Structure and Semantics," M.I.T. Laboratory for Computer Science Technical Report TR???, Cambridge, Mass., March, 1979.