COPYING COMPLEX STRUCTURES IN A DISTRIBUTED SYSTEM

Karen Rosin Sollins

May 16, 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

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

This empty page was substituted for a blank page in the original document.

COPYING COMPLEX STRUCTURES IN A DISTRIBUTED SYSTEM

by

KAREN ROSIN SOLLINS

Submitted to the Department of Electrical Engineering
and Computer Sceince

on May 16, 1979 in partial fulfillment of the requirements

for the Degree of Master of Science.

ABSTRACT

This thesis 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, algorithms, and sample implementations 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.

Thesis Supervisor: Liba Svobodova
Title: Assistant Professor of Electrical Engineering
and Computer Science

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

ACKNOWLEDGEMENTS

There are three people without whom I could not have written this thesis. I wish to thank Professor Liba Svobodova for the diligence with which she has read and understood the many drafts of this thesis. Her ability to help me extract and clarify my ideas has been invaluable.

Mike and Peter, my husband and son, have provided the moral and emotional support I needed through these many months, particularly the last two. They above all had the confidence in me that I could and should do this work.

In addition, I wish to thank Dave Clark, Allen Luniewski, Jim Peterson, Dave Reed, and Jerry Saltzer for helping me to clarify my ideas and writing. I wish to thank my parents, Axel and Kathy Rosin, and sister, Susanna Bergtold for their confidence in me. Finally, I wish to thank all the members of the Computer Systems Research Group for being themselves and providing a nurturing environment in which to do research.

CONTENTS

FIGURES

Chapter One

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. 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 objects. Each also requires other mechanisms,[1] which have, in general, been topics of research. The research reported here has concentrated only on copying, in particular copying complex data structures.

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

---

1. For example, if several copies of a mutable object exist in a system, a requirement may be that these copies be maintained in mutually consistent states.

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 chapter will introduce the model of a distributed system used in this research. Using the model, a discussion of the problem to be solved and an introduction to the solution will follow. Research related to this work will then be surveyed, concluding with the plan for the thesis.

1.1 Model of a distributed system

The model of a distributed system used in this research assumes the hardware of the system to be a network of computers, each computer having its own private memory or namespace for objects. Since a single namespace in a computer provides neither enough flexibility in naming objects nor enough protection in accessing objects, this work first develops a model of a *context* that facilitates finer partitioning of the namespace.

Each computer or node in the distributed system supports one or more contexts. The universe of objects on a node form disjoint sets, each set corresponding to a single context. Thus the context defines

FIGURES

protection of groups of objects where a group is a subset of the set of all objects on a particular machine.

The system model recognizes two kinds of entities, active and passive. The active entities are called processes, and can be executing in no more than one context at a time. Since processes are not of primary interest in this research, no further assumptions are made about them. The passive entities are objects. All objects have three attributes, value or state, name, and type. Every object has a value associated with it. An object will have the value of empty associated with it when it is created. Object values are of one of 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. At the level of contexts, every object will have at least one name, and will have exactly one in its home context. (As mentioned previously, an object may be nameable from a foreign context, and in order to do this the foreign context must assign to the object a name that is local to the foreign context.)

The third attribute of an object is its type; every object is of exactly one type for its whole life. Type is a description of those characteristics that a collection 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 the type manager per object that may be

considered an integral part of the object, or there may be an overseer of a particular type) that insures that only certain operations can be performed on the objects being maintained by it. In this work, we are assuming a single overseer or type manager for all the objects of a type at a particular physical node. Except for the most primitive types called base types, which are provided by the system to each context, every type is defined in terms of other types; the representation of such a type is in terms of the representations of other types, and the operations provided by a type are defined in terms of operations on the component types. The types that are not base types are known as extended types. An extended type object contains a list of the names of its component object. Such an object contains nothing but names local to the context in which it resides. The definitions of extended types form a network of definitions that must be based in the final analysis on the definitions of the base types provided by the system.

Several supporting mechanisms for this model of contexts are necessary. These mechanisms form the kernel. For the purposes of this research, only the message handler and storage manager are of concern. Figure 1.2 depicts this situation. 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

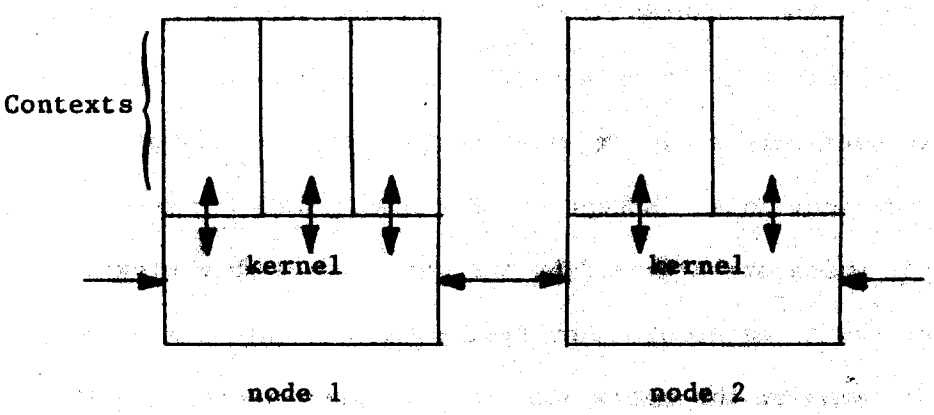


Figure 1.2 A model of the communication within and between nodes of the distributed system.

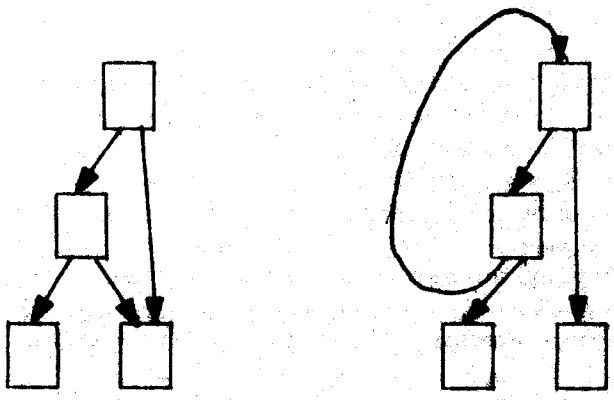
---

quite possible that the low level messages of the network do not correspond to the high level message objects or images which will be discussed later in the thesis. 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 manager, 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 manager). Each storage name is known to a single context and associated with the local name assigned to that object by that context.

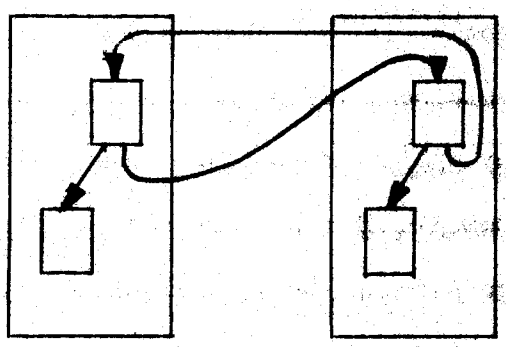
1.2 The problem

The problem that this thesis investigates is copying complex structures within the model that has been sketched. The complex structures in this case are objects of extended type, and the copying of particular interest here is copying across context boundaries. As was mentioned, copying is needed for a number of reasons. This research is a study of how to provide such copying: what the semantics of copying should be, and how to achieve them. In order to investigate copying further, we have set ourselves four goals: (1) any sharing that exists in the original structure must be maintained; (2) economy of mechanism by using a single approach in all copy operations defined (there will be three) is desirable; (3) since all communication between contexts is by message passing, the amount of message passing should be limited; (4) it should be possible to send and receive component images separately. Each of these is discussed below.

The first goal to be discussed is the retention of sharing among components when copying an objects. Although a more common concern is sharing among processes or users, this research concentrates on sharing within an object. In the model assumed for this research, objects can have arbitrary structure, including recursive containment. 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 1.3(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,



(a) Non-recursive sharing (b) Recursive sharing



(c) Recursive sharing across context boundaries.

Figure 1.3 Examples of sharing of components within a data structure.

---

considering the more complex structure that includes recursive containment of components such as the structure in Figure 1.3(b), it becomes even clearer that such sharing must be copied in order to terminate a copy operation which copies the complete structure. Sharing across context boundaries, as in Figure 1.3(c), adds a new dimension to the problem of copying. It does not introduce any new reason for maintaining sharing, however, recursive structures are much more difficult to detect across context boundaries. Thus, there is even a greater need for a mechanism that detects such sharing.

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 as in Figure 1.1.

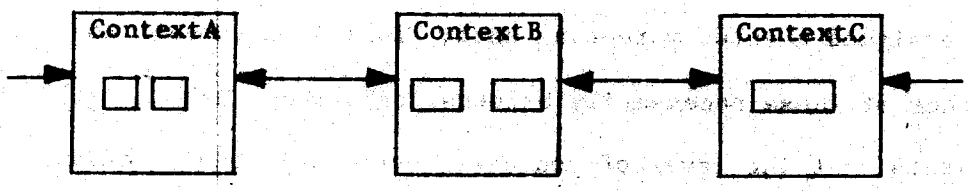


Figure 1.1 Contexts containing objects and communicating by message passing

This model of a context provides protection at a level not generally provided in computer systems. It is common for a system to enforce protection of the system as a whole; the requirement of passwords is one such mechanism. At the other extreme, individual objects are frequently protected; two common mechanisms to achieve this are capabilities and access control lists. Contexts allow for

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 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 need 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, allowing data structures to cross context boundaries. 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 1.4 is an example of this. Only those components directly connected to the

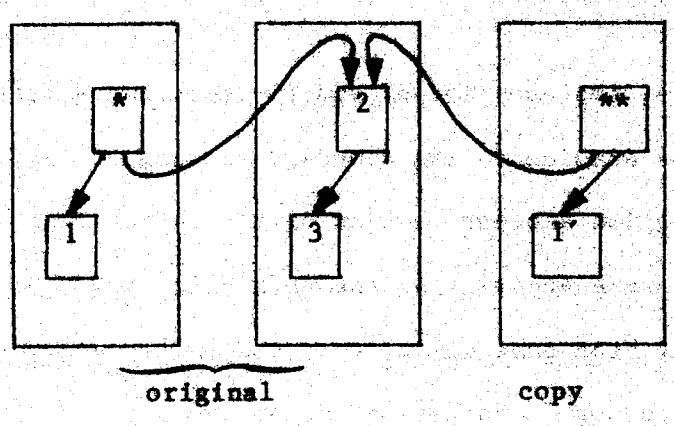


Figure 1.4 An example of the copy-full-local operation. The object labelled with * is copied into the object labelled with **, and the component labelled 1 is copied into 1'. The components labelled 2 and 3 are not copied.

top level (directly or through other local components) 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.

1.3 Related work

The model of a distributed system used in this research has been influenced strongly by the work of Saltzer[18], Liskov et al.[10,11], and Svobodova et al.[19] 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.[10,11] 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. CLU also provides approaches to the semantics of copying, the copy1 and copy operations for arrays and records, as mentioned previously. Both arrays and records can be complex structures. The third strong influence on this research is the work on distributed systems of Svobodova et al.[19] 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 directly; objects can be named indirectly by using tokens, external names for objects, passed to other contexts by the context containing the object. The model used in this research is very similar to that of Svobodova et al., except that this work is concerned only with objects, not with processes.

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 [3] and Fisher [5]) and those with particular functions in mind such as garbage collection (for example McCarthy [12,13] and Baker[1]. Although these works must be considered in a development of yet another copying algorithm, 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, for the purposes of this research an alternative approach was chosen that avoids these problems.

The external marked database developed by Bishop[2] provides much of the mechanism in his copying garbage collection for areas that our message-contexts provide here. (Message-contexts will be discussed at length in Chapters 3 and 4.) In our case the sending message-context is the repository of the names of objects that have been copied (it also has other functions) and the receiving message-context holds the names of the new objects containing the copies of the various components, in copying from the original object into an image and from an image into the copy in the receiving context. Bishop achieves this in one phase because he is not copying across naming boundaries.

1.4 Plan for the thesis

The remainder of this thesis can be divided into two parts. The first is a further amplification of the model of the distributed system: this is encompassed in Chapter 2. The second contains the discussion of the copy operations proposed as a solution to the problem of copying complex structures; Chapters 3 and 4 present this material.

Chapter 2 discusses in greater detail the nature of contexts. Three complementary views of contexts are presented: (1) the context as a naming environment, (2) the context as a node in an abstract network, and (3) the context as an object. All three views are used throughout the rest of the thesis.

Chapter 3 introduces the three copy operations. The mechanisms for the copy operations meeting the goals discussed earlier are presented in this chapter. This is then followed by a description of the algorithms

for sending and receiving in the contexts between which the copying is being done.

Chapter 4 investigates in greater detail two new types of objects, proposed in order to achieve the copying discussed in Chapter 3. It is then recognized that the simplest approach to providing copying for typed objects is to provide generic operations or procedures that can be invoked by individual type managers. Possible implementations of the important operations are then presented. One conclusion to be drawn from this work is that most of the mechanisms needed for copying can be provided by the system to the individual contexts, in the form of the generic operations, and that therefore including the type specific copy operations in particular type managers is not very difficult.

Chapter 5 is the concluding chapter of the thesis. It summarizes the thesis, and then discusses possible directions for further research related to this work.

Chapter Two

Contexts

Contexts can be viewed as several different, but complementary, classes of entities. As they were first presented, they appear to the user to be namespaces. 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. An extension of this view leads to classifying contexts as nodes in an abstract network. The nodes can communicate only by sending messages. It is also possible to extrapolate from the brief discussion in Chapter 1 to the point where contexts are considered to be typed objects themselves. Their behavior should be strictly circumscribed; their structure and the operations defined on them must be carefully specified.

This chapter will discuss separately these three aspects of contexts. It will conclude with a brief discussion of how contexts will be viewed throughout the remainder of the thesis.

2.1 Naming environment

Names are fundamental to referring to entities in a computer system. There are situations in which the value of an entity is used for identification, such as in an associative memory; however, this has not be shown to be practical when the value of the entity has a complex

structure. Thus, we will assume that each entity must have a name in addition to its value or state.

A naming mechanism, if it is designed and implemented properly, can provide flexibility in two directions, modularity and sharing, as discussed by Saltzer[18]. The achievement of modularity in a naming mechanism means that entities can be named (contained) by other entities without concern for what names are chosen within each entity. In particular, if two objects 1 and 2 use the same name A to imply different objects, 3 and 4 respectively, then object 1 should also be able to name object 2 without causing a problem with the reference A in object 2; the reference A in object 1 will still indicate object 3, and the reference A in object 2 will still indicate object 4. As mentioned in Chapter 1, Saltzer's contexts[18] provide this facility. Our contexts are modelled after his in this respect.

The other important goal of a naming mechanism is sharing. Sharing implies that there is more than one occurrence of the name for an object or that there is more than one name for the object. In other words there is more than one object naming the shared object and therefore having some form of access to it. Since we previously determined that objects are identified by names, if several different names are used for a shared object, they must in the final analysis resolve to the same name. Thus at the time a reference is made using a particular name, the name must be resolvable uniquely, but different mechanisms can be used to provide this uniqueness of name resolution. At one end of the range, there is a mechanism such as the reference tree developed by

Halstead[6]. Reference trees provide a basis for relative naming. A reference tree for an object can be considered to be a connected acyclic graph. The nodes of such a graph represent those entities that know about the object in question. A given node knows for each object which of its immediate neighbors know about the object. Using such a graph, the object could have a different name for each arc in the graph as long as each end of each arc maintains the necessary information. It is not clear that this is a useful approach to take, but it is possible. At the other extreme, it is possible to have names that are unique for all time. An example of such a mechanism is a capability system[4]; capabilities are names that are unique for all time and unforgeable.[1] Finally, it is sufficient to provide names that are all unique at any specific time, but are not unique for all time. The standard use of physical addresses is an example of this. At any one time no more than one object can have a specific address in memory, but the same address

---

1. Some capability systems, have been proposed in which the object name within a capability is a virtual address and thus is not unique for all time. For example, Bishop uses this approach[2].

can be used by different objects at different times.[1] This last approach is assumed in our model; all the objects on a node will be given names that are unique at any given time. The management and resolution of names will be provided by the kernel of the node.

Within a node, even if the node is a personal computer, used by only one person at a time, it may be useful to be able to divide the world of objects into smaller worlds. This may be simply for convenience, or there may be more pressing reasons for it such as security or containment for verification. Thus in addition to the overall naming environment in the node, we can hypothesize smaller environments called contexts.[2] Basically, a context will provide a name resolving ability for names known in the local environment into those names unique to the whole node. The whole of a node will be divided into contexts, such that every object will be in exactly one context.

---

1. As a matter of fact, in the Multics system, there are names of all three degrees of uniqueness. A segment that is shared by two or more processes, probably will be known by a different segment number in the KST or Known Segment Table of each process; thus there will be different names for the same segment. At a different level in naming the segment, when a page of it is in primary memory, if two processes want to access that page, their different names for the information they want (different because of the different segment numbers) must resolve to the same physical address. On the other hand, if the segment is not used for a period of time it may be moved from primary memory, and the physical space used for something else; the physical address now may be an address of a page of a different segment. Finally, each segment has a unique name by which it can be recognized. These last names are capabilities; they are unique for all time, and unforgeable. They are part of the information about a segment in an entry in a KST. (Such a capability exists in addition to the full path name of the segment which is a reusable name.) For a detailed discussion of the Multics system see Organick[15].

2. Since this work is to a large extent based on Saltzer's work on naming[18], the term "context" was adopted.

When an object is created, part of the creation operation is the assignment of a name local to the context in which the object is being created to that object. The context is the repository for the knowledge about whether or not a particular object exists within its domain. As long as the context knows the local name and the storage name that is associated with it, the object exists. Since it is the local name that determines whether or not an object exists, and since the local name has no meaning outside of the context boundaries, objects cannot move from one context to another. An object can be copied into another context but the resulting copy is a different object (even if the original object is destroyed).

There are a number of reasons for using local names in contexts. The first is that autonomy in naming is desirable, and often necessary, if the distributed system can be partitioned or a node can be detached from the system while continuing operation. If a centralized naming mechanism were used, it would have to be accessed every time a new object were created. If, on the other hand, the available namespace for objects were divided, in particular, along context boundaries, each context could assign locally the name for a newly created object. By combining this with a globally unique context name, globally unique naming can be achieved for objects. The second reason for using local names for objects is in order to save space. Since the model of the distributed system contains the assumption that there will be many

contexts at least one per node and probably more, the namespace for objects will be partitioned and therefore the names can be smaller.

As mentioned in Chapter 1, all objects are typed. An object of base type can be considered to contain its values, while one of extended type, any extended type, can be viewed as a list of names of the component objects. Since an object will reside in the same context for its whole lifetime, the names used for the components can and, by assumption, will be names that are local to that context. Permitting objects of extended type to contain only local names provides a simpler and more elegant model than allowing two different kinds of names, depending on whether or not the named component is local or foreign. The simplification is conceptual as well as in implementation. In addition, using only local names allows for the possibility of using capabilities provided by the local context as additional protection beyond what might be provided by protection constraints imposed on message flow at the context boundary. Therefore, an object of extended type contains only a list of local names.

The function of the context is to resolve the names used by the objects of extended type. In those cases where it is desirable, containment by naming foreign components should be available, that is, objects that reside in another context. Of course, since, as was stated in Chapter 1, communication between contexts can only be done using message passing, the names of foreign components can only be received in messages. It is also the case that such foreign components can be accessed only by sending a message to the correct context containing a

request to perform a single operation on the object. If names of objects can be passed outside the bounds of a context, objects can be shared across context boundaries.

Now, it was stated that names within objects are only local, resolvable by the local context. This means that contexts must be able to contain (map from local names into) two forms of names. One form, as already stated, is the node-wide name to be resolved by the kernel of the local node. We will call this a storage name. The other is the foreign name that needs further resolution; the current context is not capable of such name resolution. This kind of entry will consist of the name of the foreign context and a name that is local to that foreign context. The implications of this form of containment for sharing have been mentioned in Chapter 1 and will be explored further later.

2.2 Abstract networks

We now have arrived at the following situation. We have a node within a distributed system. The naming environment that it defines contains objects that are all uniquely named. From the point of view of the user this world of objects is composed of partitions which we call contexts. An object exists in exactly one context. Each context has the ability to name the objects it contains independently of all other contexts. All communication among contexts is exclusively by means of message passing. Thus our contexts are taking on the appearance of nodes in a network, resembling the abstract network postulated in the recent work done by Svobodova et al.[19]

Contexts allow for two types of protection. First, they provide a simple means of limiting error propagation. Second, they allow implementation of arbitrary protection constraints on the context; authorization to have message processed and operations performed in one's behalf within a context can be constrained to any desired degree. The second type of protection makes the first possible. As long as messages are not sent outside a context, any errors that may occur inside the context will remain contained within it. If errors cause messages to be sent, providing contexts with the ability to protect themselves to any desired degree means that they can protect themselves from external errors.

Drawing on the comparison of contexts and nodes of a network, if two processes must communicate, it is necessary to consider whether or not they are running within the same context. A process executes procedures, and since all procedures are objects and exist within some context, the process must be by definition executing a procedure from within a context. (We will avoid a discussion about whether or not the context in which a process runs is fixed for the life of the process or not.) Now if two processes are executing within the same context, they can communicate through a shared data object. This is not to say that this is the most desirable form of communication, but that it is available. On the other hand, if two processes in separate contexts wish to communicate, they have to do it by message passing. We are viewing contexts as abstractions of nodes, and have postulated that processes communicate between nodes by sending messages through the

communication medium. Thus sharing an object across context boundaries exaggerates the differences between the two kinds of sharing; if an action is to be performed on object 1, which is local to context A, from context B, (1) a request can be sent to context A for the action to be taken at context A or (2) a request can be sent for a copy of object 1 to be sent to context B in oder that the action be taken on the copy. These two forms of sharing have existed in situations where direct access was possible from both sites, but message passing accentuates the differences.

2.3 Contexts as objects

As mentioned previously, the contexts must be nameable by each other. It was stated in Chapter 1 that an object has three attributes, name, type, and value or state. In light of this definition it is possible to consider that contexts are objects, in the same way that other types of data are objects. There is something inherently different about contexts though; the domain of the names they can contain is different in nature from those contained in data or procedure objects. The latter two contain only names that are local to the context in which the objects exist. A context, on the other hand, contains storage names for those objects that exist within it, and pairs of names (name of another context and name to be resolved within that other context) for those objects that are known to objects it contains, but are not local to the context. Thus, context is a special type of object. It must be a basic type since it provides one of the interfaces between the user and the kernel. We will see later that parts of the

kernel must be able to access parts of the context type manager. In Chapter 4 we will discuss those operations for the type context that we will need to achieve the copying discussed in Chapters 3 and 4.

2.4 Summary

This chapter has discussed three distinctly different possible views of contexts. As will become clear in Chapters 3 and 4, we will use all three simultaneously. A context contains the object we wish to share by copying. In order to achieve the copying, it is necessary to perform some operations on contexts as objects and sometimes request that contexts send messages to each other to acquire foreign components as part of copying. Thus, we will slip among the different views of contexts without being explicit about it.

Chapter Three

The Copy Operations

In Chapter 2 we developed a better idea of what a context is. In particular we can imagine contexts to be nodes in an abstract network. Inside each such node is a namespace containing objects. As mentioned in Chapter 2 containment and sharing of components can occur across context boundaries. It is also the case that procedures can be invoked, requiring parameter passing, across context boundaries. Finally, multiple copies of an object for reliability and accessibility must be considered. In all these cases copying must occur when context boundaries are crossed. Therefore, the semantics of copying needs investigation.

Copying must be clarified; if a copy of an object is to be created, it must be indicated precisely in which ways the original and the copy are the same and in which ways they are different. Clearly, the values should be the same. But also, the behavior should be as similar as possible. In other words, if an object and its copy are in the same state and the same sequence of operations is performed on both, they should be in the same state afterwards. This means that any sharing that occurs in the structure of the original should also occur in the copy.[1]

---

1. As we will see later CLU[11] currently does not do this.

As mentioned previously, we will provide several different copying facilities. In a sense, the most basic copy operation is what we will call copy-one. This copies just the top level of an object of extended type. The other copy operations could in essence be built up out of copy-one operations, by explicitly requesting copy-one for each component object. The second is the most encompassing, copy-full; it involves copying the whole object, the complete structure. The third is something between the two, copy-full-local. It involves copying just that part of the object that is local to the context containing the object itself. There operations will be discussed in detail further on in this chapter and in Chapter 4. Consideration of the apparent relative usefulness of the three operations is postponed until Chapter 5.

There are a number of goals to keep in mind, while exploring copying mechanisms. First, since there will be more than one type of copy operation, we should economize on mechanism, and attempt to provide a single mechanism to achieve all the copy operations. Second, since all passing of information from one context to another only occurs through messages, the mechanisms should keep down the quantity of separate pieces of information that must move between the two contexts, in order to keep the number of messages under control. Thus, the representation of several components can be packed together in a single message. On the other hand, it seems useful to copy an object piecemeal. There are three reasons for this. First, this will help reduce the amount of buffer space needed at both ends of the message

passing facility. Second, it will allow processing at the receiving end to overlap with sending. Third, it may reduce the amount of information that may need to be retransmitted, since the bigger the message, the higher the possibility of an error. Both of these become important when a large amount of information must be passed during a copy operation.

It must be remembered that since we are assuming that all objects are typed, an object can only be manipulated through use of operations defined for its type. Therefore the copy operations must be defined for each type of object that may ever need to be copied; on the other hand, a different kind of copy, an internal one (create-image) which will be discussed later, is sufficient for types that are and will be only components.

The chapter has the following plan. Section 1 provides a brief description of the copy operations that exist for the basic types of RECORD and ARRAY in CLU[11] since our copy-one and copy-full are based on them. It also discusses other copying algorithms. Section 2 introduces the algorithms developed in this research. Sections 3 and 4 develop the details of the algorithms for the sending and receiving contexts involved in a copy. A detailed example is presented in these two sections.

3.1 Existing copying algorithms

As we have mentioned previously, CLU[11] provides a good base for discussing copy operations for extended types. CLU is a strongly typed language. This brings with it the implication that all operations are

type specific. This means that there are no generic operations that can be used on an object. On the other hand, copy operations are defined for most of the basic types of abstractions and type generators. The two types that have interesting copy operations are arrays and records. These are actually generators of infinite classes of mutable types of objects.[1] (This means that they can be used to generate types based on any other types.) For each, array and record, there are two distinct copy operations, copyl and copy. The semantics (and implementation) of the array$copyl are the same as those of the record$copyl. The same is true for array$copy and record$copy. Thus it suffices for the remainder of this discussion to use the terms copyl and copy.

The simplest way to describe the behavior of the two copy operations is to give an example. Figure 3.1 depicts a mutable object in CLU. The object contains two parts, the header, containing the description of what is to follow (specifically, the reptype, which indicates the form of the representation of the object, and the length), and the actual representation of the object. This figure depicts an object that is a list of references to other objects. A reference is composed of several flag bits, something under 10 bits to describe the type of the object named by the reference (this actually is an index into a table of pointers to descriptions of types), and the address of the object. The copyl operation creates a new object of the same type having all the same references. In other words, what is returned by the

---

1. We are proposing in this thesis three additional mutable basic types, contexts, message-contexts, and images. The latter two will be discussed in detail in this and the next chapters.

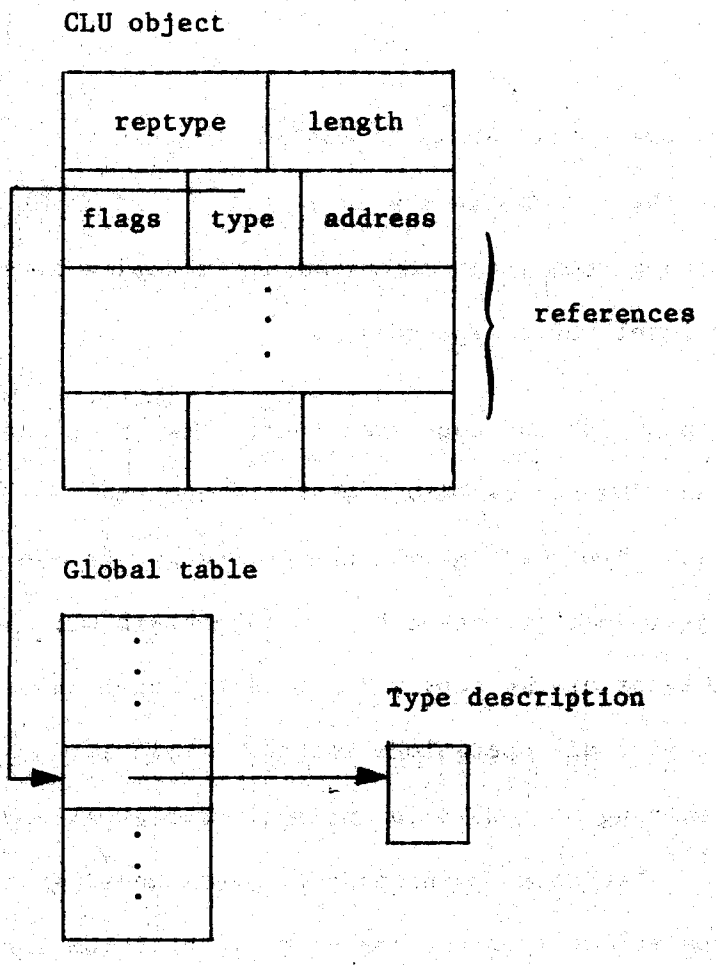


Figure 3.1 A mutable CLU object of extended type. The header contains the reptype, in this case references, and the length, in this case the number of references. The representation of the object is the list of references that follow the header. The only place in which the type of an object is stored is in a reference naming the object.

---

copyl operation is a new reference having the same type as the original, but a different address, and the object at this address has the same contents as the original object, i.e. the new object points to all the same objects the original does. The copy works as follows. First, a copyl is performed on the object to be copied. Then each reference is picked up from the new object, and a copy operation is performed on this

component object.[1] For each component, as it is copied, the new reference is used to replace the old one in the copy of its containing object. This process of copying components continues until copies have been made of all the lowest level basic type objects.

There are several problems with the copy operation. The first one is a semantic problem. If sharing exists within a record and the record$copy operation is used, this sharing will not be present in the newly created object; an object that is shared by two components will be copied twice. Thus the behavior of the copy may not be the same as that of the original object under all operations for the particular type. In order to achieve sharing that will be copied, a different copy operation must be implemented that takes cognizance of where sharing is to occur. The second problem arises from the implementation of the CLU environment in general. The lifetime of an object is no longer than the lifetime of the process that created it. A copy of the object can be saved in some form in secondary storage, but if the process that created the object dies and a new process wants to retrieve the information, it will by definition be in a new object. The name used to identify an object is unique at a given time by virtue of its containing an address. When the state or value of an object is stored or saved, all the addresses are modified so as to be relative to some base address attached to the entity being stored. Thus the names used by a process for objects can never get into secondary storage. When an object is

---

1. This description conforms to the implementation of CLU on the DEC20 system at the Laboratory for Computer Science, MIT.

retrieved from secondary storage, it will be given a new name or reference (address) based on its new position in primary memory. Now the object really has become a new object having the same structure as the old one and which might be considered to be a complete copy of the original. In this thesis, the assumption has been made that an object can have an existence beyond that of the process that may have created it. Therefore, the object must have a name that is not tied to the creating process, such as an address in the primary memory allocated to that process. If the name is not tied to a physical address, we can arrange the naming mechanism and its interface to the storage mechanism so that the physical location of an object can change without changing the value or content of the object.

In addition to the copying provided in CLU, other copying algorithms must be examined before devising one to fit the particular needs of this research. One approach that must be considered is the copying done by various garbage collecting mechanisms. An important such algorithm is that suggested by McCarthy[12] and then later used in LISP 1.5[13], MACLISP[14], and other list processing systems. This algorithm passes over the information three times, first marking all cells still accessible, second compacting or moving all the accessible cells into contiguous storage, thus adding all the inaccessible cells to the free list of available storage, and finally updating all the pointers, so they point correctly to the cells that have been moved. There are two problems with this approach. First, because the algorithm requires three successive complete passes over the structure, one in the

old location, one to move the data, and one in the new location, we would not be able to achieve much overlapping of processing. Second, this algorithm requires many more messages than necessary as will be seen later. Another approach to garbage collection has been developed by Baker [1]: real-time garbage collection. Again, as with the algorithms mentioned above, the original object and components are used to store the name of the copies. If we were to use an approach such as this, additional message passing would be necessary.

On the other hand, Bishop has developed a mechanism similar to ours[2] for his compacting garbage collector. For simplicity he does not modify the original object being copied, but rather maintains an external marked database that maps the names of objects into the new copies of these objects. An entry in this data base for a particular object indicates that it has been copied and provides the name of the copy. In our mechanism, the message-context provides a similar function, although it also maintains the list of those objects to be copied. The reason for this is that Bishop follows each path to its end, thereby copying the lowest level components first, in fact, and ending with the top level object. In this thesis one of the goals is to send images as quickly as possible, not invoking the copying recursively on components; therefore the message-context is the means of retaining the information about which components need copying.

Other algorithms for copying list structures have been developed by Fisher[5] and Clark[3]. The purpose of these algorithms is to copy an object of arbitrary size in a workspace of bounded size. In both cases

in order to achieve such a goal both the original object and the copy are utilized by changing the values in each multiple times. These algorithms have, from our point of view, problems similar to those of the garbage collection algorithms. Thus, we found it necessary to develop our own mechanism for copying in the situation in which all communication takes place through messages, and where it is desirable or even necessary to send pieces of the copy separately in separate messages.

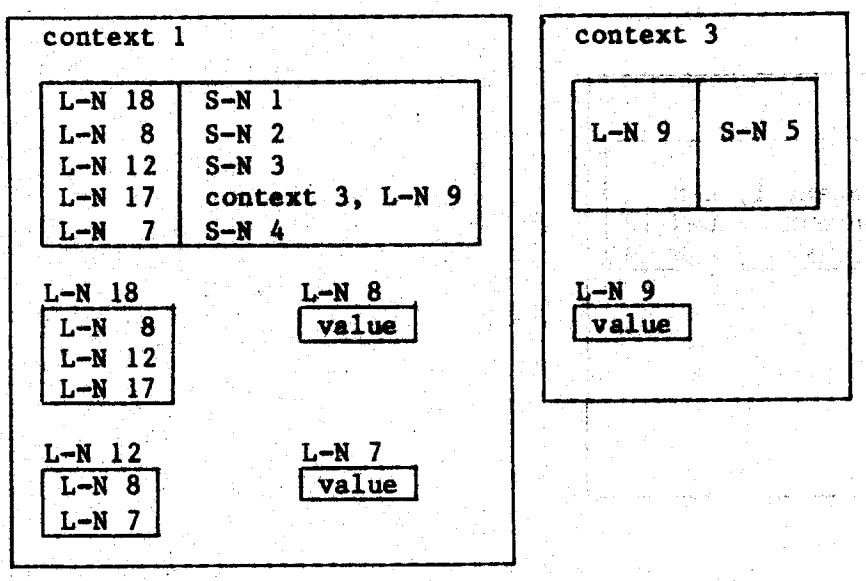
3.2 Proposed copy operations

This thesis will provide three varieties of copy operations. Two of them are very similar to the two provided by CLU as discussed in the preceding section. Two problems were brought up in relation to CLU, first, that CLU does not recognize any sharing within an object, and, second, that, as can be seen in the naming mechanism used in CLU, an object has no existence without the process that created it. We are assuming that an object has an existence tied to its context instead. It is the context that determines whether or not an object exists.

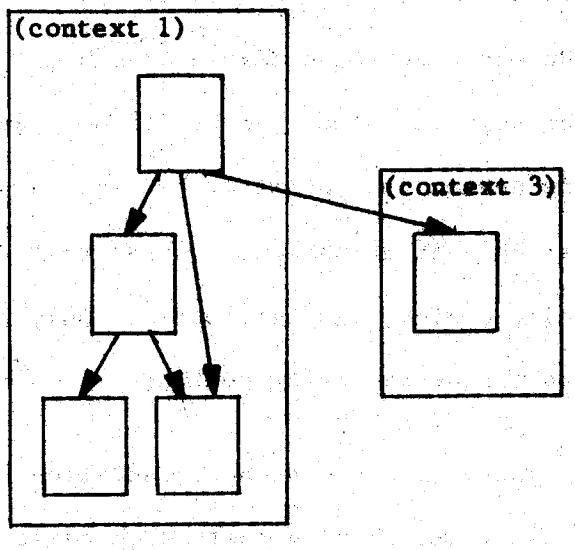
As we have discussed previously, paired with each local name in a context will be a name of one of two kinds: a full name pair of the form {context, local name}, or a storage name that uniquely identifies the object to the storage manager in order that the object can actually be accessed. Also, as mentioned previously, when an object is shared (by naming) by two components of another object which is being copied that the sharing should not be lost in making the copy.

We will call the two copy operations that are modelled on CLU copy-one and copy-full. The third copy operation is the copy-full-local. This operation is the same as the copy-full except that only the original object and those components of it in the same context as the original object will be copied, while for the foreign components only the names will be sent. Again, the best way to explain the details of these operations is to consider an example.

Let us first consider Figure 3.2(a). Throughout the remainder of this thesis the abbreviation "L-N" will be used for "local-name" and "S-N" will be used for "storage-name" in naming objects during the discussion of examples and figures. We wish to copy the object in context 1 having a local name of L-N 18 to context 5. Figure 3.2(b) shows the structure of the object L-N 18 as a block diagram. Now, in order to perform a copy-one operation on L-N 18, to create a copy in context 5, four names local to context 5 must be chosen (here L-N 31-34). Figure 3.3 depicts what will be in context 5 after the copy-one operation; there will be a copy of L-N 18 of context 1 and for each local name used in the copy in context 5 there will be a reference back to the original component. Thus the first name, when followed through, points to L-N 8 in context 1, the second, to L-N 12 in context 1, and the third to L-N 9 in context 3. The first two can be resolved to storage names in context 1, but the third can only in context 3. Figure 3.4 presents the copy-full on L-N 18 of context 1. In this case all the components have also been copied, and five local names are needed in context 5. Now, there are no references back to the original objects,



(a) The names in an object, its components, and the relevant contexts. The contexts contain mappings between local names and storage or full names as well as objects."L-N" and "S-N" are abbreviations for "local-name" and "storage-name" respectively.



(b) Block diagram of the structure of the object L-N 18 of (a)

Figure 3.2 An example of an object.

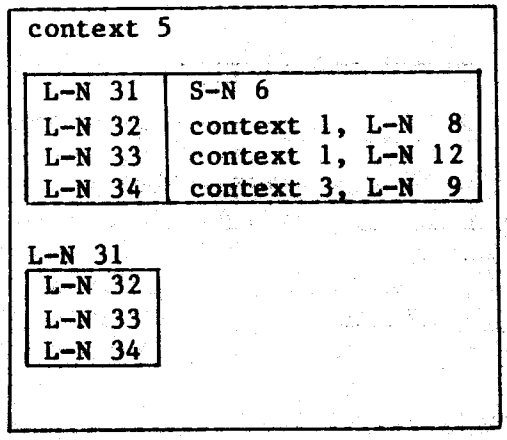


Figure 3.3 The results in context 5 of a copy-one on (context1, L-N 18) of Figure 3.2 to context 5. The context contains objects as well as a mapping between local names and storage or full names. "L-N" and "S-N" are abbreviations for "local-name" and "storage-name" respectively.

---

but also we have lost the fact that one of the components was in a context separate from the rest. On the other hand sharing has been maintained. Figure 3.5 depicts the copy-full-local on L-N 18 of context 1. Here again five local names are needed in context 5, but the component that was in context 3, since that is not the context that contained the object originally being copied, was not copied. Only the name of that object has been passed to the receiving context.

At each physical node in the system, there must be in addition to the set of contexts residing there a kernel that supports such basic functions as message passing between contexts, communication with the hardware network underlying the system, storage management, and allocation of other physical resources that are shared among the processes running in different contexts on the same node. A kernel will

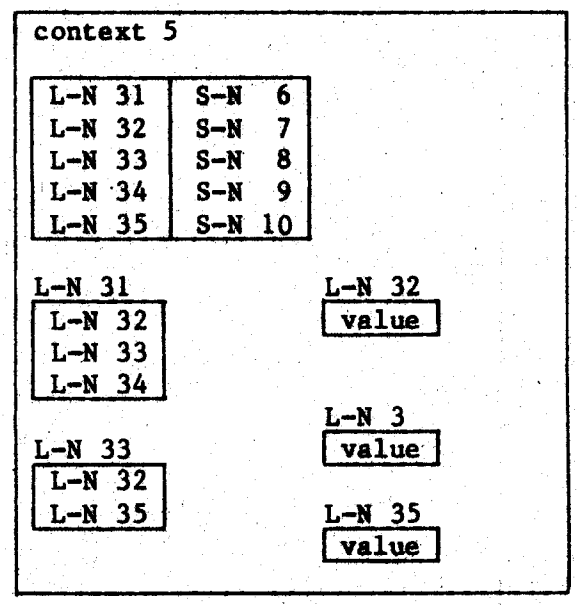


Figure 3.4 The results in context 5 of a copy-full on (context 1, L-N 18) of Figure 3.2 to context 5. The context contains objects as well as a mapping between local names and storage or full names. "L-N" and "S-N" are abbreviations for "local-name" and "storage-name" respectively.

---

also provide mechanisms for enforcing security constraints of the contexts it supports.

In copying an object from one context to another, images are created within the sending context as previously described. They are then passed to the kernel of the sending context. We will postulate a message handler that deals with all the problems of passing messages among contexts on the local node and into and out of the network for the

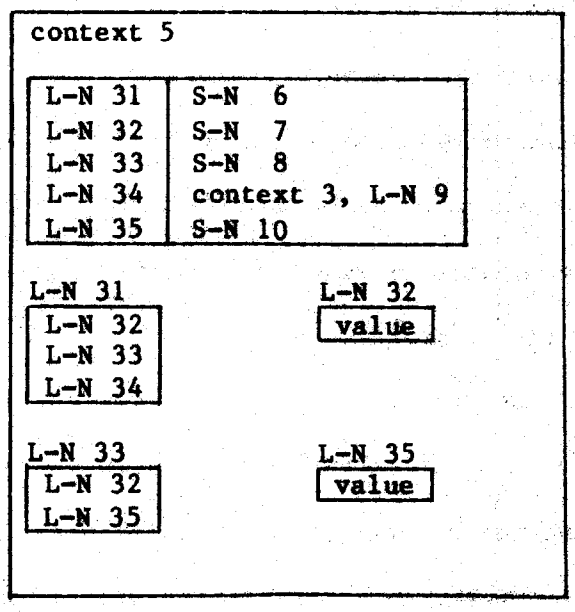


Figure 3.5 The results in context 5 of a copy-full-local on (context 1, L-N 18) of Figure 3.2 to context 5. The context contains objects as well as a mapping between local names and storage or full names. "L-N" and "S-N" are abbreviations for "local-name" and "storage-name" respectively.

---

local contexts.[1] 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 at all. The messages passed out of the sending context will simply be passed directly to the receiving context. If the receiving context is not on the local node, the message handler

---

1. We are assuming not only that the architectures of all the nodes are the same, but also that the specification and implementation of the extended and base types of objects that can be copied are the same on all machines. By this we mean that the representation of an object of extended type will be composed of the same component types on all nodes between which the object can be copied. The problems caused and avoided by such a restriction will be discussed in Chapter 5.

must prepare each message for transmission through the network to the correct node.[1]

3.3 The copying algorithms

The procedure that will be followed will be similar for all three types of copy operations. When it has been decided that an object is to be copied, the first step will be to create a message-context. A message-context is an entity that is growable and will have only a short lifetime. It is a mapping between the index of an entry and the value of that entry. An entry is created as follows: each name in the original object will be examined to find the full name, {context name, local name} pair, for it. This will become an entry in the message-context if it is not there already. The entry associated with index 0 will be the full name of the top level object being copied. Meanwhile an image of the object will be created having in place of each name in the object the index of the entry in the message-context containing the full name of the component object. The image of each component will have attached the index used in the message-context. Each object will also have the type attached. When an image of the original has thus been created and an entry for it has been made in the message-context, it is ready to send. At this point an image of the next object named in the message-context is created in the same manner

---

1. This work does not deal with the communication protocols of the network, although of course the message handler must know them. The copy operations can know nothing about these protocols nor about the degree of reliability they provide. We will discuss reliability at a later point.

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. The message-context will provide the names of those objects to be copied as components. For a copy-one operation, the copying is only performed on the top level object. Once the image of the object 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 references must be sent.

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 would be too restrictive.[1] The other part of the header is the type of the particular object to which the header is attached. Again this should not be necessary in most cases assuming that messages are received in the order sent. The reason for this is

---

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.

that if the order of arrival is predictable and the types of the components are already known, as the images arrive their types will be known. However, if the receiver is expecting an object of type any, the object being received must have its type attached to it, in order that the receiver can hand it to the correct type manager. In any case, even if we could ignore the reasoning just followed for including both parts of the header, they can be justified on the grounds that they provide redundancy that can be used for reliability.

We will now examine some examples for a better understanding of the algorithms. The object to be copied again will be L-N 18 of Figure 3.2. Figure 3.6 depicts the copy-one operation. The message-context is set

---

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

type	0
1	
2	
3	

Figure 3.6 For the copy-one operation, the images of object 0 and the message-context (without its first entry) will be sent in copying {context1, L-N 18} of Figure 3.2. The abbreviation "L-N" is used for "local-name".

---

up with the entry for the object being copied. 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 put into the message-context

an 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, it is discovered that rather than a storage name in the context, there is another {context name, local name} pair. This, then, is used as the full name to put into the message-context in the same way as the other full names. The header for the image of object L-N 18 contains both the type and a zero. Now, the image and the message-context can be sent (in separate messages, if desired, as long as there is some means of telling the receiver that the two really belong together).[1]

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 was not a problem in the copy-one, although we will see it also in the copy-full-local operation.) We want to be sure that all such sharing is maintained if that is desired. The message-context will do this for us. Second, we must consider the problems of handling foreign components. (This is not a problem in either of the other operations.)

---

1. Some optimization could be done here. First, since, only one object is being copied the zero in the header is unnecessary. Second, if no component names the original object, the entry for it in the message-context need not be sent. Third, we really do not need to send the message-context separately. Instead, we could use the full names for the references, thus including the message-context information in the image of the object.

In this case, in addition to the problems associated with acquiring a copy of a foreign component, we also must be careful to maintain sharing components across context boundaries. In order to do this, a copy-one operation should be performed on any foreign component. This means that only the top level of any foreign component plus the names it uses will be acquired. By this means the message-context will discover all sharing, even that involving foreign components.

The copy-full operation is exemplified in Figure 3.7. Again, as

---

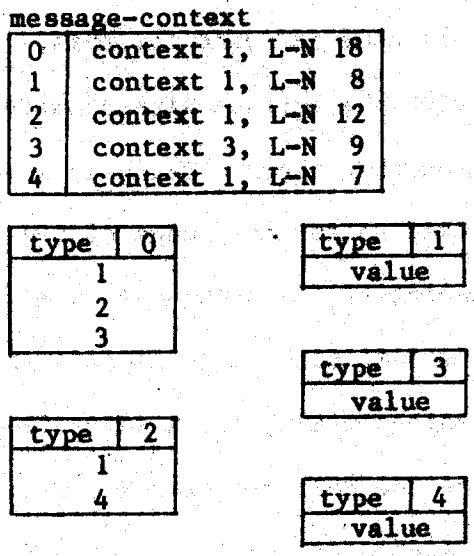


Figure 3.7 For the copy-full operation images of objects 0, 1, 2, 3, and 4 will be sent, but no image of the message-context need be sent in copying {context1, L-N 18} of Figure 3.2. The abbreviation "L-N" is used for "local-name".

---

in the copy-one, the message-context is created with an entry for {context 1, L-N 18}. Also, again, an image is created of L-N 18. Once this has been done, and the header of type and index 0 have been

attached to this image, it can be sent off. Now, the next entry in the message-context, {context 1, L-N 8}, is picked up and an image of that object is created as with the first. It is of a base type, and therefore its value will be copied. Again, the header will be attached to it, 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, we can pick up the next entry in the message-context. This is {context 1, L-N 12}, which is an object 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 made in the message-context, it will be discovered that there already is an entry for that object. Its index is picked up for 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 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, the copy of {context 3, L-N 9} will not be kept in context 1. If such a copy were kept in the sending context, we would have a situation in which the copy-full operation would have side-effects on the sending context; this is clearly undesirable.[1] Second, there may be problems 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. It is for this reason, and we will discuss it further later, that we have added the copy-full-local operation.

To resume our example, we will assume that the copy-one on {context 3, L-N 9} into context 1 has been completed successfully. We now can proceed to {context 1, L-N 7}. This is another object of a base type. The value will be copied as with L-N 8, and the header attached. Now when we look at the message-context, we see that all the objects named in it have been copied and their indices attached to them in their headers. Therefore we do not need to send any part of the message-context to the receiver of the copy, and the message-context is expendable.

As was mentioned before, the final copy operation is the copy-full-local. This seems to be particularly useful when one cannot or does not want to involve other contexts. An example of the copy-full-local operation is depicted in Figure 3.8. It is quite similar to the copy-full operation. First, the message-context is

---

1. Of course, copy-full operations will always have temporary side-effects.

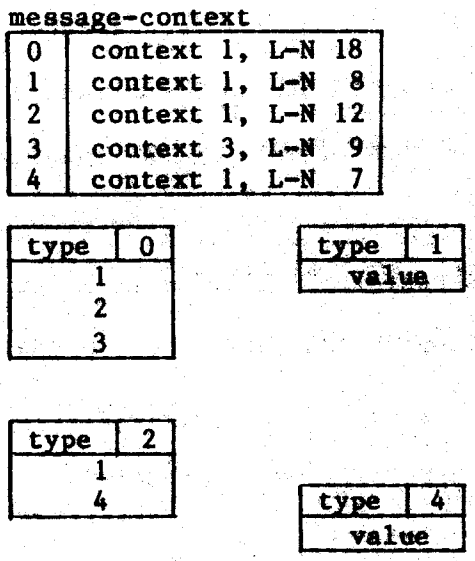


Figure 3.8 For the copy-full-local operation images of objects 0, 1, 2, and 4, and a partial image of the message-context containing the fourth entry [3, {context 3, L-N 9}], will be sent in copying {context 1, L-N 18} of Figure 3.2. The abbreviation "L-N" is used for "local-name".

---

created. The images of L-N 18, L-N 8, and L-N 12 are created. When it is discovered that the next entry in the message-context {context 3, L-N 9} names an object in a foreign context, the image for this object is not created, but the entry in the message-context is marked for future reference. Finally, the image of L-N 7 is created. Any time after each image has been created, it may be sent. An image of a partial message-context must also be sent containing all those entries in the message-context that were marked as not copied. Once all this has been sent, the message-context can be deleted and the sender has finished his part in the operation.

3.4 The receiver

As mentioned previously, the message handler for the sending context will be passed images from the sending context, and pass them to the receiving context. If the receiving context is on the same node in the distributed system as the sending context, the two contexts will make use of the same message handler. If the receiving context is on another node, the sending message handler will pass the images out into the network; a foreign message handler will take care of them. Whether or not the network was used, it is in the receiving context that the images created by the sending context must be used to create the actual copies of objects. We will present the receiving procedures 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 we want processing to begin as soon as a receive command has been issued and at least one image has arrived.

We must be able to identify each piece of a copy as part of that copy. Each piece will be labelled with its own type and its 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. As each 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 it 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.

When the message-context depicted in Figure 3.9 (a) is added to Figures 3.3, and message-context in Figure 3.9 (b) to Figures 3.4 and 3.5, we can see the receiving contexts for the copy-one, copy-full, and copy-full-local operations after all the images depicted in Figures 3.6, 3.7, and 3.8 respectively have been received.

message-context

0	L-N 31
1	L-N 32
2	L-N 33
3	L-N 34

(a) The message-context that must be added to Figure 3.3 in order that it depict the receiving context after it has received the information sent in Figure 3.6, the copy-one.

message-context

0	L-N 31
1	L-N 32
2	L-N 33
3	L-N 34
4	L-N 35

(b) The message-context that must be added to Figures 3.4 and 3.5 in order that they depict the receiving context after it has received respectively the information sent in Figures 3.7 and 3.8, the copy-full and copy-full-local.

Figure 3.9 Message-context in the receiving context. The abbreviation "L-N" is used for "local-name".

Chapter 4 will explore in greater detail the support that must be provided to achieve what has been discussed this far. In particular, it will investigate the types message-context and image and how they can be used to provide those facilities the user needs while hiding what the user does not need to know. Chapter 5 will compare the three copy operations and point out problems and some interesting possibilities for further research in similar directions.

Chapter Four

Additional Mechanism for Copying

In Chapter 3, we discussed algorithms for copying to be used in an environment of contexts as described in Chapter 2. We must now explore the implications of these algorithms in terms of what new basic types are needed in contexts, what mechanisms are needed as supports below the level of the contexts in order to achieve such copying between contexts, and the interdependencies among these entities. We will also extend the copy operations to include local copying.

4.1 Message-contexts and images

Two special types of objects were used in Chapter 3 to describe the copying operations that must be defined within the contexts: message-contexts and images. These two types are basic types and therefore provide an interface with the lower level or kernel of the node supporting the context. This section clarifies their characteristics by describing the operations defined for these types. Language constructs similar to those of CLU[11] will be used for this purpose.

As mentioned previously, message-contexts are similar in many ways to contexts. Each is a mapping from one kind of names, local to and unique within the context, to other kinds of names. Message-contexts are used specifically for preparing images when copying an object. A

message-context, as used in Chapter 3, is a mapping between the indices for entries in the message-context and the contents of those entries, that is, the full names for objects. We will modify this definition slightly later in this chapter when discussing an optimization for local copying within one context. A message-context not only must keep track of the component objects, but also must do some other bookkeeping. First, it must remember how many indices have been used. Second, it also should remember which components have been copied and which have not. We will depend on the message-context to provide the name of the next object to be copied. In order to do this, the message-context must remember which type of copy operation it is handling. The message-context must also oversee the sending of an image of a partial message-context in the cases of the copy-one and copy-full-local operations. Finally it must self-destruct.

Ten operations are needed for the message-context. Some of these are used only for local copying, and therefore will not be used until later in the chapter. The message-context operations are as follows:

1. create (object-name, op-name) returns (message-context-name): takes as arguments a local name of a local object and the name of a copy operation (copy-one, copy-full, copy-full-local, receive), creates a message-context, and returns the local name for that message-context.

2. delete (message-context-name): takes as an argument a local name for a local message-context, deletes it from the context, and returns nothing.

3. next-send (message-context-name) yields (object-name, context-name, indexl, object-local): is a CLU-like iterator. On each invocation with the same message-context-name it produces another object-name from the message-context until it has exhausted its supply, so that the name of each object to be

copied has been given to the invoker exactly once. For each object name produced it also returns the name of the context containing the object, the index of the object in the message-context, and a boolean which is true if the object is local and false if the object is foreign. If copy-one is being executed, only the name of the first entry in the message-context will be returned. If copy-full is being executed, all the names will be returned. If copy-full-local is being executed, the names of all the local objects will be returned.

4. create-image (message-context-name) returns (image-name): takes the name of a message-context and creates an image of it to be sent to another context. This transformation uses the information about the type of copy operation being done using this message-context, and the name of the local context. The image created will have its type specified as message-context. The index field can have any value since it is not used for this kind of image. If the copy operation is copy-one, all of the entries in the message-context except the first will be copied into the image. If the operation is copy-full, this is an error, because no message-context image is sent, and this should have been discovered before this operation was invoked. Finally, if the operation is copy-full-local, only those entries for which the context is not the local context will be copied into the image along with their indices. This then is the image that is sent to the receiver.

5. send (message-context-name, receiver-name) : takes as arguments a message-context name and the name of a receiver, to receive the copy. This operation manages the copying; it does the following for each component to be copied including the original object. The index in the message-context of the object is found. If the object is foreign, a copy of it is obtained. Now, regardless of whether the object is foreign, its type is found, create-image is invoked for that type. Then the index can be added to the image header, the names in the image translated into indices using the message-context, and the image sent. If the object was foreign, the copy of it created locally will be deleted. Once all this has been done for each object, an image of the message-context can be created and sent if that is appropriate. See the later sections of this chapter for more details of this operation.

6. local-send (message-context-name, receiver-name): takes as arguments local sending and receiving message-context. It generates the appropriate kind of copy of the object named in the first entry of the sending message-context. It is invoked when the receiver is local to the sending context, and achieves for local copying what the combination of send and receive

achieve for distant copying. A sample implementation of it is presented later.

7. name (message-context-name, index) returns (object-name): takes as arguments an index, and a message-context name and returns the name associated with that index in the message-context. If there is no such name, the context is requested to provide a local name which later will have associated with it an object. This operation is used in receiving

8. receive (message-context-name, sender-name): takes as arguments a receiving message-context and an identifier for the arriving copy (sender-name). This operation is the reverse of send; it receives images and manages the creation of the components of the copy using the images. It appears in an example later in the chapter.

9. receive-image (image-name, message-context-name) takes as arguments an image of a sending message-context and a receiving context and updates the receiving message-context and the context with the contents of the image. This is used only in the receiving context.

10. next-receive (message-context, sender-name) yields (image-name, type1, index1): takes as arguments the name of a receiving message-context and a sender, from whom a copy is coming. It is an iterator that yields the name of an image to be received, and the type and index that have been extracted from the image header. It provides the reverse function from next-send.

As long as message-contexts are used only for sending and receiving copies of objects, these operations are sufficient.

The other new type is image. We have discussed to some extent the use and form of an image, but more must be said. An image has a header, and is of variable size. It has twelve operations defined on it as follows:

1. create (type) returns (image-name): takes as an argument the type of the object for which an image is being created. The operation creates the image, which will grow as local names and values are stored into the image. This operation returns the local name of the image that has been created.

2. store-name (image-name, index, next-name): takes as arguments the name of an image, an index into the image, and a local name for an object to be stored there. These names will be transformed later. This operation returns nothing. It is used only in the sending context.

3. store-value (image-name, index, value): takes as arguments the name of an image, an index into the image, and a value to be stored there. Such a value will not be transformed before sending the image. This operation returns nothing. It is used only in the sending context, but not in the examples presented here.

4. store-index (image-name, message-context-index): takes as arguments the name of an image and an index which the operation will store in the header of the image. This operation returns nothing. It is used only in the sending context.

5. translate-out-name (image-name, message-context-name): takes as arguments an image and a sending message-context, and uses the message-context and sending context to transform any names stored in the image by the store-name operation into indices, adding entries to the message-context when necessary. This operation returns nothing. It is used only in the sending context.

6. send (image-name, receiver-name): takes as arguments the names of an image and a receiver for the image and passes them to the message handler. This has the effect of deleting the image from the context. This operation returns nothing. It is used only in the sending context.

7. receive: is the reverse of send. It is not invoked in any of the sample programs, but is included here for completeness. It would be invoked by message-context$next-receive.

8. translate-in-name (image-name, message-context-name): takes as arguments the names of an image and a receiving message-context and translates all the indices put into the image by translate-out-name into local names in the receiving context using the message-context and receiving context. This operation returns nothing. It is used only in the receiving context.

9. get-next-name (image-name) yields (next-name, index): takes as an argument the name of an image in the receiving context, for which translate-in-name has been invoked and returns one at a time the local names in the image, each with its index in the image. This operation is an iterator. It is used only in the receiving context.

10. get-next-value (image-name) yields (value, index): takes as an argument the name of an image and returns one at a time each value in the image with its index in the image. This operation is an iterator. It is used only in the receiving context, but not in the examples given in this thesis.

11. transform-names (image-name, message-context-name, receiver-name): takes as arguments the names of an image, a sending message-context and a receiving message-context in the same context. This operation translates the local names in the image into the appropriate local names for a local copy operation. This operation returns nothing. It is used only for local copying.

12. delete (image-name): takes as an argument the name of an image, and deletes the image from the local context. The operation returns nothing. It is used when receiving a copy.

These are the operations needed for sending and receiving images of objects. Images are the only base type objects that have the send operation defined for them.

4.2 Layering in a node

Now that we have a better idea of the two new basic types that are the basis of the interface between contexts and the system, we can explore the layering again in more depth. At the system level, we need to clarify the function of two entities: the storage manager and the message handler. In the kernel of a node, objects are named by their storage names. Storage names are used by the storage manager to name uniquely every object that the storage manager must handle. It is not clear that storage names have to be unique over all time, although they obviously should be unique at any one time. A storage name must not appear in more than one context at a time, because that would imply direct sharing of the object; two contexts would have an alternative form of communication to message passing. Depending on whether or not

storage names are to have other functions such as providing some of the security wanted, they may be capabilities, but they also may be just physical addresses (if objects are not physically moved), or possibly names that hide the physical locations of the objects, but provide no security (in other words they are forgeable). Names that are local to a context have no meaning at this level; they are only strings or numbers or very simple entities that hopefully will not be manipulated in any way that is damaging to the objects and contexts.

The storage manager is the interface between physical storage and all the rest of the system. There is no reason that all the nodes in a distributed system need to have the same implementation of the storage manager. The storage managers will be protected from each other by the message handlers and type managers on the various nodes. It is these that must agree and cooperate, not the storage managers. The storage manager may be quite cleverly written to take advantage of all sorts of optimizations, such as sharing immutable objects between contexts by providing separate storage names that map into the same object. It may use the types of objects to help in utilizing space to greater advantage or reduce time requirements for moving objects between different levels of memory.[1] The implementation of the storage manager, however, is not of concern in this thesis.

---

1. These optimizations, however, lead to a cyclic dependency that is undesirable from the point of system verification, so the developer of such a storage manager would have to take extra precautions. See Janson[9], Chapter 3, section 3 for a detailed discussion of cyclic dependencies and many of the problems related to them.

The other important entity is a message handler. As mentioned above, the message handlers must cooperate. The message handler will take the storage name of an image, and the name of the receiver, and see that the message handler for the receiver receives the image. Assuming the receiver is at another node, the sending message handler will prepare and send the image through the network. Thus the message handler will have to know network addresses of all relevant contexts and the network protocols for sending images. As far as the information about locations of contexts is concerned, this information can be kept in a table that is internal to the message handler; it is information about which nothing else in the node should need to know. Alternatively, a protocol such as the one suggested by Reed[16] could be used to find the receiver name by interrogating directories at different nodes. The network protocols will be discussed no further than to say that the various message handlers must agree upon them.

If we were to parallel the sending and receiving in the case in which the two contexts are on the same node, the same message handler would be used for the two, but it would call on the image type manager in the receiving context to create a new image in the receiving context. The image object that was created in the sending context should be deleted. The image should never appear to be in two places at once.

On top of the kernel containing the storage manager and the message handler are the contexts. As discussed in Chapter 2, contexts are namespaces, the only namespaces available to the user of such a system. Figure 4.1 depicts one view of this arrangement. When a context is

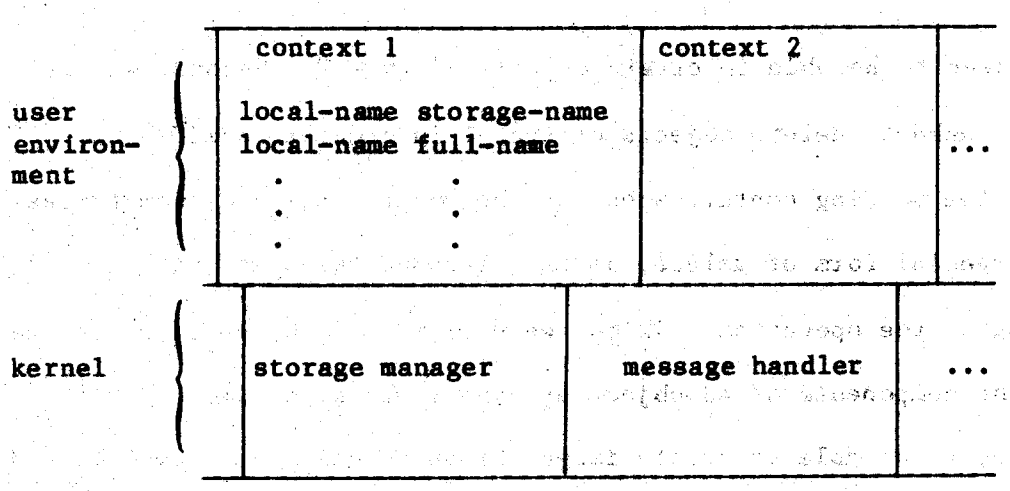


Figure 4.1 Layers in the system (on one node).

created, it will have a certain number of local names preassigned to important objects, such as the type managers of all the base types including the context type manager, and all other resources that, in the final analysis, must be shared among the contexts (for example the kernel and all the hardware that is to be used by more than one context). It is also possible that a context will need to have a local name assigned to itself, to do some forms of name translation, or receive responses from type managers, for example.

4.3 The details of sample copy operations

In this section we will present the details of the copy operations for a type manager of a specific extended type. We will demonstrate this on an example of a hypothetical type, T; we assume that objects of type T are mutable and can be created with a value of nil. There are several other assumptions we need to make about objects of type T.

First, we need to be able to create objects of type T. Second, we will also find a need to delete objects of type T in order to avoid side effects in the sending context when copying under certain circumstances. This is a special form of delete, as is discussed below in the description of the operation. Third, we will need to be able to get the names of the components of an object of type T one at a time. Fourth, we will need to be able to create images based on objects of type T, and receive images and translate them into objects of type T. Fifth, we will need to be able to assign components of an object of type T one at a time. This means that it must be possible to create an object of type T with a value of nil, and insert into it components one at a time. This implies a need for storing a component into an object of type T.

To achieve the copy operations the following supporting operations will be assumed for type T. (The operations copy-one, copy-full, copy-full-local, and receive-copy will be discussed and exemplified later.)

1. create () returns (object-name): takes no arguments, but creates an object of type T and returns a local name for it.

2. delete-copy (object-name): takes the name of an object of type T, calls on the context to delete that local name and all the local names contained in the object from the context, and returns nothing.

3. get-next-name (object-name) yields (next-name, index): takes as an argument the name of an object of type T and returns one at a time the names and indices within the object of each of its components. This operation is an iterator.

4. create-image (object-name) returns (image-name): takes as an argument the name of an object of type T and returns an image of that object containing local names that will be translated

later. The image may also contain values that will not be translated.

5. receive-image (image-name, object-name): takes as arguments an image name and an empty object of the type any. There must be only names local to the receiving context or values in the image. The state of the image will be put into the object. This operation returns nothing.

6. store-name (object-name, index, next-name): takes as an argument the name of an object, an index to a component of the object and the name of that component to be installed. This operation returns nothing.

This is a list of only those operation needed in type T in order to achieve the copying. It says nothing about what other operations there may be in type T.

A number of additional details must be specified. We will assume only two operations on contexts:

1. request-copy (object-name, context-name) returns (new-name): takes as arguments the two components of the full name of a foreign object, obtains a copy (copy-one, to avoid any loss in sharing) of the object, and returns the local name for the newly created local object, which is a copy of the foreign object. It guarantees that a new local name is assigned to each non-local subcomponent name.

2. local (object-name) returns (boolean): is a test operations that takes an object name as an argument and returns T if the object named is local to the context, and F otherwise.

In order to implement the procedures described below, some modifications are needed for the CLU type any. This work assumes two operations on the type any: (1) type, which produces the actual type of the object in the any object, and (2) force, which forces the any object to the object inside the any. CLU provides no operations for the type any, although it does provide a force built-in function. Another type that is assumed

in this work is type. No operations are needed for it in the sample implementations in this work.

Now that we have a better understanding of the relevant aspects of contexts, and the full complement of operations available for images and message-contexts, we can consider the details of a possible implementation of the three forms of copying discussed in Chapter 3. We will begin by noticing the similarities among the operations. In particular, the message-context appears to be a focal point. Since the message-context contains the name of the copy operation being performed (copy-one, copy-full, copy-full-local) and the identity of the top level object being copied, it contains enough information to be at the core of all three copy operation. The message-context$send operation provides this central function on the message-context. The message-context is created containing two pieces of information, the name of the original object being copied and the type of the copy operation. (Later message-contexts will also be created with "receive" as the name of the operation using them.) Message-context$send invokes the create-image operation of the type manager for each component to be copied. When all components that should be sent have been sent, an image of the part of the message-context naming those components not sent is created and sent. After some cleaning up the copying is complete.

It is important to keep in mind that the tools provided for the system users should be as simple as possible and should not contain any mechanism for which there is no apparent need on the particular level of abstraction. Message-contexts may well fall into this category, but

they can be hidden from the creator of an extended type manager or cluster. A method of achieving this is to make available three generic operations or procedures named proc-copy-one, proc-copy-full, and proc-copy-full-local. These procedures will simply see that message-contexts are properly created and sent. There is one other place at which the programmer might come into contact with message-context; when the images are sent, they contain only names generated by the message-context, yet the creation of the image of an object of extended type should be controlled by the extended type manager. The reason for image creation being in the type manager is that what actually is sent should be type specific. There may be information which is node specific, that the receiving type manager will have to acquire later. There may be components such as temporary workspace that it would be a waste to send, and perhaps for security reasons should never be sent. Whatever the reason, image creation should be under the control of the type manager. For this purpose, the programmer will be required to write a create-image operation, which will see that the image is created and write values and only names local to the context into the image using the image$store-value and image$store-name operations. The message-context$send operation will later, unbeknownst to the programmer, invoke image$translate-out-names, using the appropriate message-context. Thus the programmer never knows of the existence of the message-context.

The only other pieces of code the programmer must write are definitions of which copy operations are to be defined for the type. These operations will do nothing but invoke the appropriate generic copy operation passing along the parameters. Figures 4.2 and 4.3 provide a possible coding of the procedures described in this section for objects such as the top level object used in the examples in Chapter 3. They are written in a subset of a language based on the conventions of CLU.

```
copy-one = proc (object-name: T, receiver-name: any);
     proc-copy-one (object-name, receiver-name);
     end copy-one;

copy-full = proc (object-name: T, receiver-name: any);
     proc-copy-full (object-name, receiver-name);
     end copy-full;

copy-full-local = proc (object-name; T, receiver-name: any);
     proc-copy-full-local (object-name, receiver-name);
     end copy-full-local;

create-image = proc (object-name: T) returns (image);
     image-name: image := image$create ("T");
     for (next-name: any, index: int) in get-next-name
        (object-name) do
          image$store-name (image-name, index, next-name);
          end;
     return (image-name);
     end create-image;
```

Figure 4.2 Operations in the T type manager

Figure 4.4 presents an implementation of message-context$send; since message-contexts are base type objects, the message-context type manager with all its operations is provided in every context. Message-context$send is somewhat involved. It iterates over all the

```
proc-copy-one = proc (object-name: any, receiver-name: any);
     message-context-name: message-context :=
        message-context$create (object-name, "copy-one");
     message-context$send (message-context-name, receiver-name);
     message-context$delete (message-context-name);
     end proc-copy-one;

proc-copy-full = proc (object-name: any, receiver-name: any);
     message-context-name: message-context :=
        message-context$create (object-name, "copy-full");
     message-context$send (message-context-name, receiver-name);
     message-context$delete (message-context-name);
     end proc-copy-full;

proc-copy-full-local = proc (object-name: any, receiver-name: any);
     message-context-name: message-context :=
        message-context$create (object-name, "copy-full-local");
     message-context$send (message-context-name, receiver-name);
     message-context$delete (message-context-name);
     end proc-copy-full-local;
```

Figure 4.3 The generic copy operations or procedures provided to each context by the kernel.

---

names in the message-context. While this is happening, additional entries are made into the message-context by the image$translate-out-names operation. For each object, the message-context$send operation requests a copy of the object if that object is not local to the sending context. Once there is a local copy of the object (if the object was local no additional copy will have been created), the type of the object can be determined and the appropriate create-image operation can be invoked. This operation will create an image containing possibly a subset (this will be discussed later) of the same local names that were in the object itself. Therefore, the programmer does not need to know about message-contexts in order to write the create-image operation. Once the image has been created, the

```
send = proc (message-context-name: message-context, receiver-name:
   any);
     for (object-name: any, context-name: context, index: int,
        object-local: boolean) in next-send (message-context-name)
        do
          if (object-local equal F) then
               new-name: any := context$request-copy (object-name,
                  context-name);
               object-name := new-name;
               end;
          type1: type := any$type (object-name);
          image-name: image := type1$create-image (any$force
             (object-name));
          image$store-index (image-name, index);
          image$translate-out-name (image-name,
             message-context-name);
          image$send (image-name, receiver-name);
          if (object-local equal F) then
               type1$delete-copy (any$force (object-name));
               end;
          end;
     if (op-code (message-context-name) not equal "copy-full") then
          image-message-context: image := create-image
             (message-context-name);
          image$send (image-message-context, receiver-name);
          end;
     end send;
```

Figure 4.4 The send operation of the message-context type manager.

---

index in the message-context of the object from which it was created can be placed in the header of the image. Also, the names in the image must be translated from local names to indices in the message-context. This may involve creating new entries in the message-context, and therefore also may involve the context. After this translation has been completed the image can be sent. If a copy of the object was acquired from a foreign context, the copy will now be deleted, and the whole procedure can begin for the next component. When images have been sent for all the components to be copied, if the operation was not a copy-full, an

image of the message-context must be sent. This completes the message-context$send operation.

4.4 Preservation of sharing

An interesting situation now exists with respect to the copying of foreign components. At the context level, we can control how much sharing within an object we wish to worry about across context boundaries. If we wish to maintain all sharing regardless of context boundaries, the context will request a copy-one of the foreign component. We will use Figure 4.5 as the basis of further discussion.

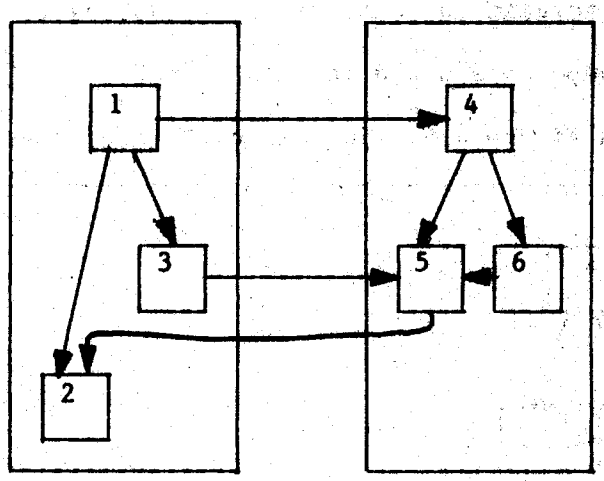


Figure 4.5 An example of sharing across context boundaries. The numbers in the boxes represent values of objects.

When a context requests only copy-one for each foreign component, exactly one object image and a message-context image will be acquired for each foreign component. Thus the message-context for the whole copy

operation will keep track of all possible sharing even across context boundaries (because for every component the globally unique name is found). The result is that the structure in the receiving context will be exactly that in Figure 4.5 except that it all will be in one context. The problem with this is that a request must be sent out for a copy of every foreign subcomponent of the original foreign component named in the object being copied. If instead it is decided that we care about most sharing but are willing to trade losing some in return for the saving in time and messages, a copy-full-local can be used instead. In this case, we will lose the identity of subcomponents of a foreign component that are local to that foreign context. Thus requesting a copy-full-local of the foreign components would lead to a final structure of the form depicted in Figure 4.6. In this case, many fewer

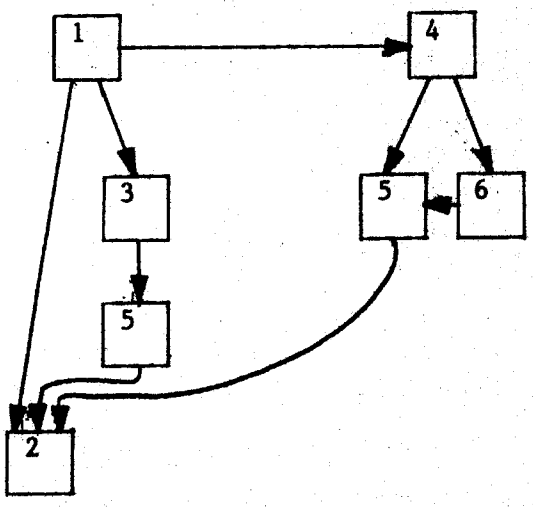


Figure 4.6 The resulting structure of a copy of the object shown in Figure 4.5 when copy-full-local is used across context boundaries. The numbers in the boxes represent values. Thus we can see clearly where extra copying has taken place.

messages will be required if the foreign components are large (that is, have many subcomponents), but if there is much sharing across context boundaries, there will be a greater expense in terms of space needed for the additional copies of the subcomponents. If there is no interest in maintaining sharing across context boundaries, the copy-full operation can be invoked. In this case, only sharing that is local to a context or in which two local components name the same foreign component will be preserved. Figure 4.7 provides an example of the structure in the receiving context for the case in which copy-full is used to request copies of foreign components in order to prepare images. Using the copy-full may have serious drawbacks although in many cases it may save

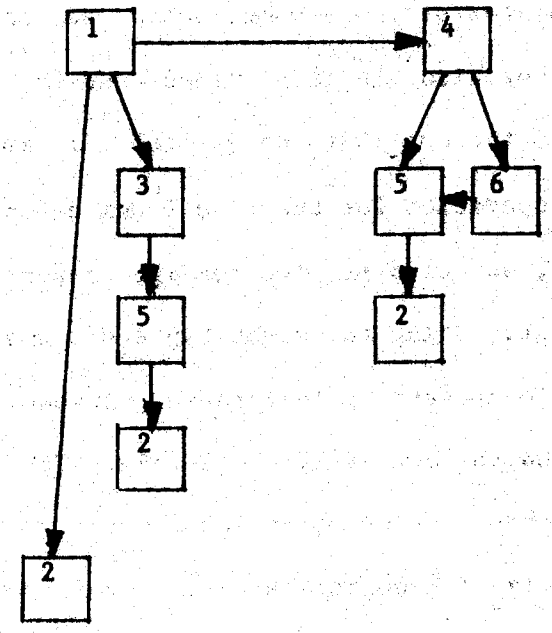


Figure 4.7 The resulting structure of a copy of the object shown in Figure 4.5 when copy-full is used across context boundaries. The numbers in the boxes represent values. Thus we can see clearly where extra copying has taken place.

much in time and many messages. The problem is that the foreign component may contain foreign components, which may contain foreign components. If such a structure has loops not only across context boundaries, but across node boundaries, the infinite recursion might be very difficult to discover, and even more difficult to handle. Thus, although this may be a very tempting approach because of its simplicity, it is probably something that ought to be avoided.

There is an issue that has not yet been addressed, although it was considered in determining the operations earlier in this chapter. When a local copy is made of a foreign component, in order to create an image that will be part of a copy-full operation, such a local copy must not have any side-effects on the local context. In other words, not only must the copy itself be deleted, but also the local names used to identify any foreign components of the copy must be deleted. It is for this reason that the delete-copy operation for the type T was defined to delete not only the object itself, but also all the foreign components of the copy, from the local context. Using the other types of copying when requesting copies of foreign components solves these problems, the copy-full-local to some extent, and the copy-full completely. The reason for this is that by using these, fewer or no local names will be associated with full names of foreign subcomponents before copies of them are acquired. As we have seen there are other tradeoffs. Perhaps the decision as to which form of copying is used in requesting copies of foreign components should be left to the person on whose behalf the context is created.

4.5 The receiving end

The operations needed to receive an object of type T are similar to those for sending except that rather than three kinds of operations there is just one. When the bits representing the sent images arrive over the network, the message handler receives them and places them in something called a pseudo-image. The message handler must extract from incoming messages identifiers to be used in assembling the images belonging to the same object. When the receive request has been issued by the appropriate type manager, in our case type T, the process of creating the copy in the appropriate context can begin.[1] The implementation of receiving is similar to sending; again, a generic operation is provided to be invoked by the receive-copy operations of particular types. Again most of the control is in the message-context, in this case in message-context$receive. Now, the iterator used to drive the whole operation of receiving is next-receive which yields images acquired from the message handler. The operations of interest for this thesis are receive-image and receive-copy for type T, the generic receive proc-receive, and message-context$receive. For an implementation of these see figures 4.8, 4.9, and 4.10.

It is in the create-image and receive-image operations of type T that the decision as to what is copied and how it is made. These two

---

1. As a check that the copy was performed correctly, perhaps type checking ought to be done on the complete structure. The message-context can be used for this to avoid any loops. This is simply a matter of checking that all the components of each component are of the correct types.

```
receive-copy = proc (sender-name: any) returns (T);
     object-name: T := create();
     proc-receive (object-name, sender-name);
     return (object-name);
     end receive;

receive-image =proc (image-name: image, object-name: T);
     object-name := create ();
     for (next-name: any, index: int) in image$get-next-name
        (image-name) do
          store-name (object-name, index, next-name);
          end;
     end receive-image;
```

Figure 4.8 The receive-copy and receive-image operations of the T type manager.

```
proc-receive = proc (object: any, sender-name: any);
     message-context-name: message-context :=
        message-context$create (object-name, "receive");
     message-context$receive (message-context-name, sender-name);
     message-context$delete (message-context-name);
     end proc-receive;
```

Figure 4.9 The generic receive operation or procedure.

---

operations together provide the type specific qualities of copying. It is here that we can decide not to copy some components without causing the whole copy operation to fail. For instance, if some component of an object is context specific, the create-image operation might generate a special signal or value to the receive rather than the name of a component even in a copy-full. The signal must be interpreted by the receive-image, so that it will be able to fill in the appropriate component. This is just an example of the reasoning that might occur.

```
receive = proc (message-context-name: message-context, sender-name:
   any);
     for (image-name: image, type1: type, index1: int) in
        next-receive (message-context-name, sender-name) do
          if (type1 not equal "message-context") then
               image$translate-in-name (image-name,
                  message-context-name);
               object-name: any := name( message-context-name,
                  index1);
               type1$receive-image (image-name, any$force
                  (object-name));
               else receive-image (image-name,
                  message-context-name);
               end;
          image$delete (image-name);
          end;
     end receive;
```

Figure 4.10 The receive operation of the message-context type manager. It is similar to the message-context$send operation.

This completes the discussion of copying across context boundaries, but there is still one more form of copying that must be discussed.

4.6 The local copying operations

The last situation that must be considered is when the copying is done within a single context. For this we will use much of the mechanism already in place, modifying it where necessary. The semantics of the copy-one, copy-full, and copy-full-local for the local situation should be the same. As a matter of fact the operations can be invoked by using the same operation names. The only changes needed are changes to pieces of code that the programmer never sees, in particular the generic operations and message-context$send. To make these operations work for a copy within a single context, it is necessary to simulate the

important parts of both the send and the receive sides of a copy operation, handled in a single operation, message-context$local-send. For this we will use two message-contexts, although we will see later that this is not always necessary. When the copy operation is invoked, the receiver-name will be the local name of the copy that is to be created. The first message-context will remain the same as previously, associating with the index used for a component the full name of the component. The second message-context will be used to associate, for each component, the index that it had in the first message-context with the local name for the copy, if that is appropriate. Thus the first entry will be the name passed as the receiver-name. At this point, the receiver-name will be reassigned to contain the name of the second message-context. For any component that will not be copied (as with some components in copy-one and copy-full-local), the local name of the original will appear in the second message-context.

Since, as we said before, the sending message-context can be thought of as representing sending of the copy, the receiving message-context, or in this case the second message-context, can be thought of as representing the reception of the copy. Therefore, making the second message-context the receiver-name and placing in it the local name of the new object are reasonable. The procedures that must be changed to achieve this are the generic copy operations and

```
proc-copy-full = proc (object-name: any, receiver-name: any);
     message-context-name: message-context :=
        message-context$create (object-name, "copy-full");
     receive-local: boolean := context$local (receiver-name);
     if receive-local then
          second-message-context: message-context :=
             message-context$create (receiver-name, "copy-full");
          receiver-name := second-message-context;
          message-context$local-send (message-context-name,
             receiver-name);
     else message-context$send (message-context-name,
        receiver-name);
     end;
     message-context$delete (message-context-name);
     if receive-local then
          message-context$delete (receiver-name);
          end;
     end proc-copy-full;

message-context$local-send = proc (message-context-name:
   message-context, receiver: any);
     for (object-name: any, context-name: context, index1: int,
        object-local: boolean) in next (message-context-name) do
          if (object-local equal F) then
               new-name: any := context$request-copy (object-name,
                  context-name);
               object-name := new-name;
               end;
          type1: type := any$type (object-name);
          image-name: image := type1$create-image (any$force
             (object-name));
          image$transform-names (image-name, message-context-name,
             receiver-name);
          new-object: type1 :=  any$force( name (receiver-name,
             index1));
          type1$receive-image (image-name, new-object);
          image$delete (image-name);
          end;
     end local-send;
```

Figure 4.11 The proc-copy-full modified to take into account local copying and the message-context$local-send procedure. Proc-copy-one and proc-copy-full-local are identical to the proc-copy-full except for the creation of the message-contexts where the appropriate operation name must be used.

message-context$local-send must be created. (This could be included in message-context$send, but for ease in understanding the programs has not.) Figure 4.11 depicts these revisions.

There is a great deal of mechanism here to achieve something apparently simple. There are several reasons for this. First of all, one of the primary goals of this work was to retain any sharing in the structure; message-contexts are needed to do that. Second, using the mechanisms already in place to perform distant copying economizes on mechanism. Third, as mentioned previously, message-contexts can and should be hidden from the programmer in creating copy operations. We use images again and have only modified routines that will be provided by the system for the programmer. He only needs to think about copying. Thus, although the mechanism appears complex from the system point of view, the programmer's job has been simplified.

As mentioned earlier, there are some possible optimizations. One has already been included in Figure 4.11. If we were to duplicate strictly what is done at a distance, we would have two images, one for sending and one for receiving. We have elided these two into one. A second is that it should be apparent that for the copy-one operation the two message-contexts will be identical except for the original entry. Hence, only one message-context is necessary. Actually, for copy-one we could get away with none, and simply perform a bit by bit copy from the original to the copy of the object.

4.7 Additional issues

This section addresses several additional issues that arise in the implementation of the contexts and communication between contexts.

1. Global naming for contexts and types of objects.

A problem with globally unique names of any sort is that they imply at least cooperation among the entities needing to make use of name generation and possibly a loss of autonomy for those entities. One approach to generating globally unique names is to provide a single name server. This certainly can be made to guarantee unique names. Unfortunately, no new names can be acquired by a name client when he is detached from the name server. A solution to this is to distribute the name server, by partitioning the namespace and providing each potential client with a piece or subset of the whole namespace. This is what has been done for objects in the model used in this research. Each context has a part of the namespace of the whole distributed system. By combining the locally unique name of an object with the globally unique context name, objects can be assigned globally unique names. But this is based on the assumption that contexts have globally unique names. The same procedure of partitioning the context namespace by nodes of the distributed system could be used, so that a node could be detached from the system and still be able to create new contexts. Now, the nodes need to be globally uniquely named. At some point the process of dividing the namespace must stop and there must be dependence on a central name server. It is quite reasonable to expect this at the level

of naming the nodes, because this may very will be encoded into the hardware interface to the communication network supporting the distributed system.

The assumption of global names has also been made for types. The situation here is a little different though. There is a certain amount of negotiation that must occur in order for two contexts to agree that they both have correct versions of the type manager. There is no reason that part of this agreement cannot be to agree on an external name for the type. Neither context needs to use the external name internally, but both must know it in order to translate it into the local names. It is still p ssible that a centralized name server may be necessary.

2. Uniformity among machines in defining types copied between machines

We have assumed that when an object is copied from one context to another, not only will there be the right set of type managers or clusters at the receiving context to receive the copies of the object and its components, but also that the type managers will be defined in terms of the same component types. In other words the representation of any type that is to be copied will be the same in terms of its component types in the contexts between which it will be copied. It may not have been completely apparent that this assumption was made, but as long as we permit the partially copying operations, copy-one and copy-full-local, component types must be the same. Let us reconsider the object L-N 18 of Figure 3.2. Let L-N 18 be of type T1. In context 1, let its third component be of type T2. If in context 5 (the

receiving context) an object of type T1 is implemented as having the third component of type T3, we have a problem. For the copy-one and copy-full-local operations the copy should have the third component pointing to an object of type T3, but has a component of type T2. A more difficult occurs in cases in which the representation of a type has different numbers of components in different contexts. Thus whether or not a component (or several components) need to be converted into a different type can be determined only by examing other components. To solve this a different approach to copying would have to used, one with much less overlap.

There is also a more subtle problem when creating the copy at the receiver (in the cases where several images are passed, copy-full and copy-full-local) if the various representations of a type are different in terms of type of components in different contexts. Sometimes when a component of a specific type e.g. T2 above, is received it will be transformed into an object of another type e.g. T3 as above and other times not. Whether or not this should be done may not be known until all the images have been received and processed. Much reprocessing may need to be done. As with the previous point made above, some degree of autonomy is at stake if type representations must conform to each other.

3. Sharing code between contexts on the same node

This problem can be broken into two problems depending on whether or not the code in question is pure code or not. If the code can be impure, each context must have its own copy of every piece of code, in

particular type managers. If this were not the case, there would be another means of communication between contexts besides message passing, and that has been excluded from our model. If, on the other hand, code can be guaranteed to be pure, even though two contexts may in fact have different storage names for a piece of code, the storage manager may actually map these different names into the same object representing the piece of code. In particular, at the bottom of the network of type managers, the type managers of the base types (e.g. those that may be implemented in hardware) will be pure, and therefore can be shared. As a matter of fact, those in hardware must be shared, unless there is a separate processor for each context, which seems like an unreasonably severe limitation. Looking at this problem slightly differently, we must consider whether or not immutable objects can be shared. We can conclude that this form of sharing is invisible.

4. Synchronization

Conceptually the simplest and most straightforward mechanism to guarantee consistency is locking. There are several problems with this. First, it requires an additional pass over the object in order to discover and lock all the components. Second, there is a more serious problem when components are foreign. In this case many complications arise. There is a problem of responsibility for foreign locks if they can be held by foreigners. If this is allowed, then outside forces may impinge on the autonomy of a context. Thus, for practical as well as security reasons, locking may not be the correct solution. An approach developed by Reed[17] 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 also solves the problem of locking foreign components.

5. The size of message-contexts

The size of message-context is a potential problem. One of the requirements that Fisher[5] and Clark[3] put on their copying algorithms was bounded buffer space to achieve the copy. We have traded that for a smaller number of messages and the ability to process in parallel. although we have considered the problem of the size of message-context in developing the algorithms presented here. First of all, we have eliminated, as much as possible, actually copying the message-context. Second, we expect that the system will support a larger quantity of and more useful base types than CLU[11], some of which will be larger in order to avoid having to break every object into the immutable base types of CLU. For instance, it may be useful to consider arrays and records of base types to be base types. As mentioned in earlier chapters, we consider contexts and message-contexts to be base types. This means that when a message-context is sent as part of a copy-one or copy-full-local, it need not be broken apart into smaller pieces. More research needs to be done to determine additional base types.

6. Types of component objects that should not be copied

An object of a particular type may include components that should not be copied, although the object itself may have a copy operation defined on it. The reasons for this may be numerous. For example, one of the components of a procedure may be its workspace. This certainly should not be copied. Or, a table that is to be customized for the local context is to be copied. Some of the components should be copied, but in place of others flags should be sent, so that the type manager in the receiver will insert the correct component in these spots. We have provided the hooks for handling this problem, in the form of the create-image operation that the implementer of the type manager must provide. This means that type specific image creation is performed, and therefore can be written to provide the desired flexibility.

4.8 Summary

This chapter has presented in greater detail the sending and receiving operations needed to copy an object. To this end we defined two types of objects, the image and the message-context. The image is the vehicle by which we pass the value or state of the object from the sender to the receiver. Other work[11] uses the terms encode and decode to describe the operations of creating from the original an image in order to create a copy. The message-context is the means by which we retain any sharing in the original structure in the copy. In addition, it is the means by which we avoid looping infinitely when copying cyclic structures.

The remainder of the chapter detailed how one might implement the copying making allowances for foreign components and for local copying. In order to do this, a number of operations were assumed to exist for both the context and the hypothetical extended type for which copy-one, copy-full, and copy-full-local were then defined. An important result of this chapter is that what needs to be written by the implementer of an extended type, in order to provide these copying facilities, is minimal.

Chapter Five

Summary and Conclusions

5.1 Summary

We are now at a point to review what has been accomplished in this thesis. We began with a model of a distributed system. It has as a hardware base a network of computers. Each node in this network supports a kernel, the local system software. On top of this we postulate one or more contexts at each node. A context can be viewed in several ways: as a namespace in which processes can execute, as a node in an abstract network (with communication among such abstract nodes only by message passing), and finally as objects in a world of typed objects. We also assume that the typed objects contained in contexts may not migrate among contexts. Given this model of the system, we investigated sharing. Since we do allow naming across context boundaries, sharing is possible. However, sharing of foreign components is limited to the following two ways: passing messages requesting operations to be done on the object in the foreign context, or by acquiring a local copy and performing operations locally on the copy. In the first case, the physical object is shared, and any mutations of the object caused by one of the sharers will be visible to the other. In the second case, since a copy of the object is passed, although the information content of the object at the time a copy is made is shared, the physical object is not and therefore any changes made by one of the

sharers will not be visible to the other. In spite of this, since all communication must be done by message passing, sharing by copying may be the more desirable approach for a number of reasons. First, message passing is likely to be expensive in terms of both time and space. Second, if the two contexts between which messages are passing are on different computers, since we have assumed as much autonomy as possible for the nodes and cannot predict failure of either the nodes or the communication network, we have no guarantee that the node containing the object will be available at any particular time. Thus, sharing by making a copy may be the only reasonable alternative. In any case, it certainly is an alternative that should be provided.

In order to achieve this sharing by copying, we have defined three copying operations that we think ought to be considered. The first is the copy-one, copying just the top level of the object's structure. Second, we considered the copy-full, which copies the complete structure of the object including any components that reside in another context. Finally, we have looked at a novel approach to copying, the copy-full-local which copies a complex data object to the boundary of the context containing that object, but no further. In devising a mechanism for achieving these copy operations, several goals were set. First and most importantly, we want to maintain any sharing that exists in the original structure, because we believe this to be an important part of the information contained by the object. Second, we want to economize on mechanism by using a single approach in all three operations. Third, since all communication between contexts is by

message passing, we want to limit the amount of message passing necessary; that is, copying should require as little communication back and forth between the two contexts as possible. Finally, we want to allow for parallel processing at the sending and receiving ends of these copy operations. The mechanism discussed in Chapters 3 and 4 achieves these goals. In order to do this, we have postulated two new types of objects, the image and the message-context. Now copying simply requires creating a message-context to be used to reconstruct the sharing within the structure and determine which objects are to be copied as components. The type image is the type of object that actually can be sent in a message. Thus for each object that the message-context determines must be copied, an image is created and sent. At the receiver, the reverse is done. The message-context again is the means of handling sharing within the structure and images are the objects that are received and that bear the information that is used to create the copy.

The procedures that have been developed in Chapter 4 indicate that copy operations can be implemented in such a way that the creator of a new extended type must do very little in order to provide these three operations for his type. First, he must define the operations simply as invocations of generic procedures of similar names. These procedures are to be provided in each context by the kernel. The other chore left for the programmer is to define how an image is created from an object of his type, by implementing the create-image operation for his type. Thus the actual contents of the image can be type specific, yet the

implementer need never know about message-contexts and other details of the copy operations at all. In order to receive copies, similar operations must be written by the programmer; receive, which invokes the generic receive operation, and receive-image, which transforms an image into an object of the type being implemented by the programmer. We have also shown how the mechanism can be extended to provide the three copy operations within a single context (in addition to copying across context boundaries) without requiring the implementer to distinguish between these calls.

Thus, assuming our particular model of a distributed system, this thesis developed a solution to the problem of copying. The following section will assess the relative utility of the three operations and mechanism developed.

5.2 Conclusion about the research

Now that we have developed a mechanism to solve the problem presented in Chapter 1, we must examine what has been achieved. This discussion will be divided into two parts. First we will consider the relative usefulness of the three copy operations. Second, we will consider in what ways the mechanism might be simplified if we were to relax our goals as initially stated in Chapter 1.

As stated earlier in the thesis, the copy-one may be considered the most basic of the three copy operations we have presented. In theory the other two operations ought to be achievable by a repeated application of copy-one. In practice, in order to maintain the sharing,

the programmer would have to take on the function of discovering most of the sharing from the message-context. The message-context will only discover sharing among components of a single object. Simulating the copy-full and copy-full-local using the copy-one would also involve much more message passing than we have found necessary. It is not clear how useful the copy-one operation is; if the object to be copied is of an extended type, then copying only the top level does not appear to be very useful, as the actual state of the object is still only accessible by passing more messages across context boundaries. (Of course, there may well be situations in which it is desirable to allow the names of components to be passed around without actually copying the components.) On the other hand, if the object is a base type object, there is no difference among the various copy operations; all three should have the same effect. The only difference in this case is whether or not extended types using the base types as components can have defined on them one or another of the copy operations. As will be discussed further later, in order to define the copy-full and copy-full-local operations for an extended type, it must be clear that the relevant operation is defined for each component type. This will be considered further in the discussions of verification and exception handling.

Now, when considering the copy-full operation, we find this to be what is most frequently considered to be the standard copy operation. In our model, severe complications may arise because contexts may support arbitrary authorization constraints, nodes may disassociate themselves at any time from the system, and the communication network as

well as the individual nodes may not be reliable. (We are not considering the reliability issue related to whether or not an individual message is lost or scrambled, but rather how useful the copy-full operation is if the network has a high probability of being unavailable at any given time.) The copy-full also requires the extra commitment in time and space to acquire copies of all foreign components in order to create images. Thus we are led to conclude that perhaps the copy-full is too general.

The copy-full-local is a new operation developed in this thesis that appears to strike a middle ground. It approaches a solution to the above criticisms of the other two operations. Assuming some or most of the components of an object are in the same context, most of the state of the object can be copied. In addition, if the full state of the object is the objective, there is a savings in number of messages (copy requests) and message-contexts (one returned for each copy request) over those needed if only copy-one operations are executed. At the same time if the foreign components are unavailable for whatever reason the copy-full-local operation does not fail where the copy-full would. Of course, if instead we have the situation in which most or all the component objects are in other contexts, perhaps on other nodes, then the copy-full-local may not represent much of a saving to the receiver of the copy. Since the other components must also be requested in this case, it will be necessary to access the foreign components anyway. It also means that some sharing in the structure may be lost, because once images have been created, the globally unique identities of their

originals are no longer attached to them. On the other side, in this case of widely dispersed component objects, the copy-full may be more expensive in terms of use of resources and time, since each of the foreign components really will be copied twice. We suspect this is an unusual situation, but the only true test is experience. Thus we recommend that all three operations should be available in a system similar to the one we have modelled, although we suspect that the copy-full-local will be the most useful.

The mechanism presented appears to be fairly complicated. It is worth considering whether it could be simplified if we relaxed some of our goals for the copy operations. Our goals or constraints on the copy operations were listed in Chapter 3 and again earlier in this chapter. The most important one was to maintain sharing among components. We have already discussed relaxing this in acquiring foreign components to create the appropriate images for a copy-full. If we were to eliminate consideration of any sharing, we must consider whether we could eliminate message-contexts. The answer is that we could not entirely. It would still be necessary to pass to the receiver the names of components not copied in the copy-one and copy-full-local operations and these would have to be collected somewhere. The problem is that objects only contain names local to their containing context for a number of reasons discussed in Chapters 3 and 4. If only names local to the sending context are sent in images, the naming network would become much more complex. Foreign components of an object might become inaccessible because one of the intermediate contexts was unavailable, when, in fact,

the context containing the component was available. On the other hand, using only the globally unique names would solve this probem. Instead this would cause a waste of space. Using either local names or globally unique names causes another problem; it allows the local names for the components within the sending context outside the bounds of that context. For security reasons, this may be undesirable. There is also the problem of circular lists or recursive containment. If that were not to be handled by message-contexts, there would have to be some other mechanism. It is possible that if locking were used as the synchronizing mechanism, it could also be used to detect circularity. Unfortunately, as we have mentioned, there are other problems with locking. Thus it is quite likely that, even if we were not to consider sharing, message-contexts would provide the simplest approach to solving these other problems. A relaxation of the second and third goals of economizing on mechanism and limiting the number of messages needed to copy a component would only lead to a more complex mechanism because of the nature of the model with which we are working. The final goal of allowing images of components to be sent, received, and processed separately, if relaxed, might allow for some simplification, although not at the sending context. A simplification would occur in type checking the structure as it is created rather than needing to wait until all the components have been created. (This type checking was not included in the procedures presented in Chapter 4, since it is necessary only for reliability, an issue not addressed in this thesis.) The mechanism we have presented would still need images and message-contexts. At the receiver, the components could be processed in

the most convenient order, which is probably the order in which they were sent. This would simplify those functions provided by the system. On the other hand, the message handler might have to be more complex and certainly would need more buffer space, since it would have to collect and order all the appropriate pseudo-images before the execution of the receive command could start. This approach would simplify receiving a complete copy (copy-full) in those cases in which the representations of the type are different in the sending and receiving contexts. It would not be of much help in the case in which the representations are different, but only a partial copy is occurring. The tradeoffs are such that it is not clear there is any benefit to be had from relaxing this goal. Thus we are led to conclude that the copy operations as defined in this thesis solve the problem presented within the realm of the model postulated.

5.3 Suggestions for further research

Since we have used CLU as a basis for much of our work, it is reasonable to consider the possibility of including the proposed operations in CLU. As CLU stands currently, it is based on a different model from ours. It assumes a user environment in which there is a single process and a single namespace (address space). However, work is currently progressing in the direction of extending CLU for a distributed environment.[1] Operations similar to the proposed copy

---

1. This work is taking place at the Laboratory for Computer Science, M.I.T., Cambridge, Mass. under the direction of B. Liskov and D. Reed, however, there is as yet no published work.

operations of this thesis need to be considered to facilitate sending values of abstract objects between processed in such environments.

This thesis has dealt with some parts of the copying problem in a distributed system. There are related areas that need research; generalizations of the work presented here are also possible. In accordance with this, we have a number of suggestions. They fall into three categories: items 1, 2, and 3 address additional details that have to be solved when implementing the scheme presented in this thesis, items 4, 5, and 6 are extensions to the work, and item 7 is a generalization.

1. This research has addressed only the problem of sending typed pieces of information. It is clear that there are other entities that can be in messages names, commands or requests, additional control information, type information, to name a few. Further research into what kinds of messages, other than images, is needed. This must be done in the context of a more detailed model of the distributed system.

2. We have mentioned very little about verification. Much verification should take place at compilation time of type managers. The receiving context should be able to verify that all received components are of appropriate type, possibly even check value ranges. In the environment of autonomous nodes, it is important to do some run-time checking. As mentioned, if a copy-full or copy-full-local is defined for a type, it had also better be defined for its components. This can be checked at compilation time for the local type managers. Now, there are two

possible interpretations of a type being the same in two different contexts. In the first case, in addition to the type being composed of the same component types, all the same operations are defined. In this case, even for a copy-full which allows foreign components, type checking can all be done at compile-time. In the second case, two types being considered the same means that they have the same representation in terms of component types, and the operations of one are a subset of the operations of the other. In other words, for those operations that are defined on both, the operations have the same effect, but not all operations need be supported in every context. The effect of this is that during a copy-full operation, run-time type checking for the availability of operations must be done in the foreign contexts. Although, as previously discussed, a type should be composed of the same component types at each site between which copying is to occur, this does not guarantee that copying is defined for a specific type at each node on which on the type occurs. Regardless of the definition of two versions of a type manager being the same, permission to copy a particular object must be checked. This can only be checked at run-time. Work must also be done on guaranteeing that two type managers at different nodes really are implementing the same type if they claim to be, regardless of which definition of being equivalent is used. This is easy if type managers are written in a high level language and are simply distributed to and installed (compiled) at individual nodes. This is a great imposition on nodes and directly threatens their autonomy and ability to operate while disassociated from the distributed system. It is clear that work must be done in this area.

3. We have not addressed any issues of exception handling, except obliquely. We have pointed out several places at which exceptions might occur: the context boundary (insufficient authorization), unavailable operations (discovery that a particular operation is not defined for a component type in a foreign context), an unavailable node (the node has been detached from the system), an unreliable network. It may be difficult to distinguish some of these, but thought must be put into what to do when exceptions occur.

4. It might be quite useful to be able to pass images to a context without requiring that the context use them to create copies, but rather be able simply to assemble a collection of images for storage or passing on to a third context. In this case what may be needed in addition to the receive command at the receiving site is a command that would imply just collecting images. Possible uses for such a facility might be to support a file server or back up storage.

5. This thesis has explicitly excluded the issue of moving objects. We have assumed that an object resides permanently within one context. If it is necessary to create the appearance that an object has moved, this should be handled at a higher level by creating a copy of the object at the new location, deleting the original, and using a higher level name to point, first, to the original object and then, after the "move" to the new object. There are problems associated with moving objects. One is the question of resolving names of objects. Since in our model the name contains the name of the context, there would have to be some policy and mechanism for how to resolve outstanding references to the

moved object. Further questions relate to security policies, such as if there is an outstanding reference to the object before it moved, should that reference be updated, who has the right to update it, when should this occur, and the list goes on. This is an area for much more research.

6. It might be interesting to extend the approach used in this thesis as follows. Each time a new entry is made in the sending message-context a new process is created to copy that new component. The processes all would use the same message-context, so no sharing would be lost. There would be a master process associated with the message-context, and one for each component to be copied. In this way, much more parallelism might be achieved if the hardware could support it. If processes are not expensive, not much has been lost in overhead, while allowing for as much parallel processing as possible. Of course, activities involving the message-context would have to be synchronized, but that could be managed by the process associated with the message-context. This approach is an extension in the direction pursued by Atkinson, Hewitt, and Baker [7,8].

7. The approach we have taken in this thesis to copying is to translate every object into an image. Images are the only objects that can be sent in messages. This approach can be generalized so that we have, instead of images, message-images, display-images, printer-images, etc. In other words, for each physical device there is a form in which it expects information. This can be used to create the appropriate abstractions as we have done for the network by creating our images.

This should simplify the task of transferring objects to other devices. The programmer must specify which operations are to be defined for his type and write the operation to transform one of his objects into the image appropriate for the device. At this point the programmer's responsibilities should halt and the system should take over. This puts responsibilities where they belong.

One of the most important considerations in looking to the future will be to learn more about how this kind of model would be used (how it relates to the characteristics of real distributed applications) and to assess the costs (performances) of the operations proposed in this thesis. It is possible that experience will indicate that different operations or even a different model is needed. The research presented in this thesis must be tested by experience and the proposal of alternatives.

References

[1] Baker, H.G., Jr., Actor Systems for Real-time Computation, M.I.T Laboratory for Computer Science Technical Report TR-197, Cambridge, Mass., March 1978. (Also Ph.D. Thesis for the Dept. of Electrical Engineering and Computer Science, M.I.T. Cambridge, Mass. March, 1978.)

[2] Bishop, P.B., Computer Systems with a Very Large Address Space and Garbage Collection, M.I.T. Laboratory for Computer Science report TR-178, Cambridge, Mass., May 1977. (Also Ph.D. Thesis for the Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., May 1977.)

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

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

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

[6] Halstead, R.H., Multiple Processor Implementations of Message-Passing Systems, M.I.T. Laboratory for Computer Science Technical Report TR-198, Cambridge, Mass. January 1978. (Also S.M. Thesis for the Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. January, 1978.)

[7] Hewitt, C. and Baker, H., "Laws for Communicating Parallel Processes," Proc. IFIP Congress 77, North-Holland Publishing Co., New York, August 1977, pp.987-992.

[8] Hewitt, C. and Atkinson, R., "Specification and Proof Techniques for Serializers," IEEE Transactions on Software Engineering SE-5, 1 (January 1979), pp. 10-23.

[9] Janson, P.A., Using Type Extension to Organize Virtual Memory Mechanisms, M.I.T. Laboratory for Computer Science Technical Report TR-167, Cambridge, Mass. Setpember 1976. (Also Ph.D. Thesis for the Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., August 1976.)

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

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

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

[13] McCarthy, J., et al., LISP 1.5 Programmer's Manual, 2nd edition, The MIT Press, Cambridge, Mass. 1965.

[14] Moon, D.A., MACLISP Reference Manual, Project MAC. Massachusetts Institute of Technology, Cambridge, Mass., December, 1975.

[15] Organick, E.I., The Multics System: An Examination of Its Structure, The MIT Press, Cambridge, Mass., 1972.

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

[17] 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 for the Department of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass., September 1978.)

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

[19] Svobodova, L., Liskov, B., Clark, D., Distributed Computer Systems: Structure and Semantics, M.I.T. Laboratory for Computer Science Technical Report TR-215, Cambridge, Mass., March 1979.