DISTRIBUTED COMPUTER SYSTEMS: STRUCTURE AND SEMANTICS

Liba Svobodova

Barbara Liskov

David Clark

March 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 number N00014-75-C-0661

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

TABLE OF CONTENTS

Acknowledgement

This report is an outcome of a series of discussion meetings that brought together people from different research groups from within the Laboratory for Computer Science, in particular the Programming Methodology group and the Computer Systems Research group. In addition to the authors, Irene Greif participated most actively in these meetings; her comments often exposed the need to clarify our approaches and conclusions. All other participants contributed in their own way through our lively discussions. Jerry Saltzer, who was on a leave of absense during the period covered by this report, provided valuable comments on an earlier presentation of this material.

Keywords and Phrases: Abstract Types, Distributed Systems, Information Sharing, Inter-Process Communication, Programming Languages, Protection, Reliability.

DISTRIBUTED COMPUTER SYSTEMS: STRUCTURE AND SEMANTICS

Abstract

This report describes on ongoing project in the area of design of distributed systems. The goal is to develop an effective programming system that will support well-structured design, implementation, maintenance and control of distributed processing applications. This programming system combines a powerful high level language and operating system features, and addresses the underlying system problems that affect the reliability and security perceived on the application level. The report presents a conceptual model of distributed computation, and, in the context of this model, discusses our approaches to inter-node communication and cooperation, reliability, and protection. One of the basic goals of our project is to allow the application programmer to work with application-oriented entities. Thus, inter-node messages, error handling and protection constraints will all be expressible in application oriented terms. The report concludes with some examples of the language constructs and an outline of the future research under this project.

1. INTRODUCTION

Computer systems should reflect the structure and needs of the problem to which they are being applied. For many applications, a distributed computer system represents a natural realization. For both technical and economic reasons, it is likely that for many existing applications, distributed computer systems will replace conventional computer systems built around a large central processor, and that new applications will emerge based on distributed information processing.

The area of "distributed systems" has become a popular source of systems research projects. This trend has been supported mainly by the rapidly falling cost of computing hardware and the increasing power and flexibility of mini and microcomputers. However, many research efforts in this area seem to miss the most important aspect of the revolution in the hardware costs and power: the steadily decreasing entry cost of acquiring and operating a free-standing, complete computer system encourages lower-level units within a large organization to acquire their own computers that consequently will operate somewhat independently and autonomously from one another. The administrative autonomy is really the driving force that leads to acquisition of local computers dedicated to the applications of a particular organization unit. However, it is necessary to anticipate that these autonomous computer systems will have to be at least loosely coupled into a cooperating confederacy that serves as the information system of the organization.

The basic technical problem in these emerging systems is to provide coherence in communication among the nodes in a computer network while these nodes retain their administrative autonomy. Technically, autonomy appears as a force producing incoherence: one must assume that operating schedules, loading policy, level of concern for security, availability, and reliability, update level of hardware and software, and even choice of hardware and software systems will tend to vary from node to node with a minimum of central control. Further, individual nodes may for various reasons occasionally completely disconnect themselves from the confederacy, and operate in isolation for a while before reconnecting. Yet to the extent that agreement and cooperation are beneficial, there will be a need for communication of signals, exchange of data, mutual assistance agreements, and a wide variety of other internode interaction. We hypothesize that one-at-a-time ad hoc arrangements will be inadequate, because of their potential large number and the programming cost in dealing with each node on a different basis.

The move toward distributed systems will be dictated not just by their "naturalness", but also by the many technical advantages they offer over centralized systems. These advantages include the following:

Availability. Availability of information can be increased by replicating it at several nodes. This arrangement not only increases the access bandwidth to the information, but in case of a failure of one of the nodes or some communication link, the information remains accessible.

Protection. Distributed systems provide a better environment for protecting information stored in the system and for coping with run-time errors resulting from hardware failures or residual design and implementation errors. These advantages arise from the actual physical separation of independent or loosely coupled computations and information that belongs to different users. The physical boundaries of individual nodes provide "firewalls" that (if properly designed) will prevent spreading of errors originating in a particular node to the rest of the system and protect information stored at individual nodes from unauthorized access or modification by other nodes. As the most severe protection measure, a self-contained node can be guaranteed privacy during some sensitive operation by physically detaching it from the rest of the system.

Expandability. As more users join the system or new services are added, it is not necessary to make any physical replacements in a distributed system. Rather, one or more new nodes need to be added to the system; if the system is designed properly, it may be possible to accomplish this without interrupting the service of the existing system. Thus, distributed systems offer a potential for a more gradual and smoother growth than systems with a large central processor.

Thus, there are many sound reasons why applications should be implemented as distributed systems. However, while it has been successfully demonstrated that it is not very difficult to interconnect remote computers at the electrical and information bit level, the effective utilization of such a network at a higher software and applications level is still missing. The project discussed in this report is aimed at solving the technical problems that hinder the development of applications for distributed systems. In particular, the goal of this project is to develop an integrated programming language and operating system to support well-structured design and implementation of distributed applications.

1.1 Distributed Systems Of Interest

The distributed systems considered in our project can be described loosely as organizations of highly autonomous information processing modules, called nodes, which cooperate in a manner that produces an image of a coherent system of a certain defined level. Autonomy is the key characteristic that eliminates most multiprocessor organizations from this class of distributed systems. Certainly, a distributed system has more than one processor, since it has at least one processor in each node. However, in a distributed system, the nodes are highly independent, each having its own primary memory, possibly some secondary storage, and its own interface through which it communicates with its environment (e.g. user terminals, sensors). The individual nodes are connected by a communication network; the communication delay may be highly variable and unpredictable. The communication network might be a long-haul network such as the ARPANET [ROBE70], a local area network [CLAR78], or a suitable combination of these two types. Each node has access to its own memory only; that is, inter-node communication is possible only by explicitly exchanging messages, not through shared memory. Finally, physical (geographical) reorganization of the nodes and the communication network is assumed not to impair the system's functionality; the only change might be in the system's performance.

1.2 Comparison Of Our Approach With Related Work

The assumption of autonomy of the nodes that compose a distributed system is the most important ingredient that distinguishes our work. However, once autonomy is assumed, the next issue that arises is to devise techniqes that permit programs running on the autonomous nodes to communicate in a coherent fashion. We are aiming at a high level of coherence that is application-independent but permits communication among the nodes in application-oriented terms. This high level of application-independent coherence distinguishes our approach from other work that is based on the assumption of autonomous nodes. Most work has either provided a very low level of coherence (e.g. the ARPANET) or has provided coherence within a specific application (e.g. the NSW works manager [MILL77]). There is some work related to ours in progress at Xerox PARC, but again this work is focussing on a very specific application -- office automation.

The problem of simultaneous update, making an identical or a logically related change at several sites, has received considerable study [GRAY78, MONT78, REED78, ROTH77, STEA76, TAKA78, THOM76]. However, we remain unconvinced that a solution to this particular problem is crucial to our research. Rather, we view our system as providing an environment in which any one of several simultaneous update algorithms can be implemented as needed. This point distinguishes our work from SDD-1 [ROTH77], for example, since that project assumes a very particular technique for implementing simultaneous update. SDD-1 also makes very restrictive assumptions about the autonomy of the nodes of the system.

Distributed systems have only lately become a focus of programming language research. In the past, programming languages have mostly not addressed concurrent programs. More recent languages (e.g. Concurrent Pascal [BRIN75], Modula [WIRT77]) have had features for concurrency, but within the context of a single processor: these languages are based on the assumption that programs interact through shared memory, which is not consistent with the concept of autonomous nodes with private memory. There is related work at Oxford [HOAR77], the University of Rochester [FELD77] and at MIT [DENI75, HEWI76], but this work does not place strong emphasis on integrating the language and operating system features.

Indeed, we feel that our emphasis on integration of language and system is a further key factor in our work that distinguishes it from other related work. Much of what distributed programs do falls into what is usually considered to be the systems area, including such topics as synchronization of access to shared information, and protection. However, programs are written in a programming language, and proper primitives in that language can greatly influence the structure of programs. By integrating the two areas we expect to achieve a greater impact on the construction of distributed systems than could be accomplished in either area separately.

2. STUDY OF APPLICATIONS

It is essential that the mechanisms we develop to support construction of distributed applications cover the real distributed processing problems. To this end, we have studied a number of applications, both by direct observation [SVOB78A, SVOB78B] and by surveying related work as discussed earlier. This study was hampered by the lack of _existing_ distributed systems; for example, banking systems are not yet distributed, although a distributed system is being planned. Therefore, we had to supplement our study by sketching designs for future systems.

Several different classes of distributed activities have been identified:

Invocation of remote servers. A message is sent to a remote node instructing some server at the node to perform a certain operation; a reply (requested information or an acknowledgement if no data is to be returned) confirms that the operation has been performed. The mail system in the ARPANET is an example of this type of application.

Atomic Transactions on Distributed Databases. To preserve the integrity of a database, it may be necessary to provide a mechanism that guarantees that either all updates specified by a transaction will be performed, or none, no matter how the transaction fails.

Distributed Data Processing. If the large quantities of data residing at different nodes are processed, a problem may arise even if no updates are performed, which is to minimize the data moved between nodes in order to perform the desired operation. An example is query processing in a distributed database system.

Distributed Problem Solving. This describes systems where the cost (overhead) of maintaining a centralized global view of the system state and control is prohibitive. In such systems, each node knows only a partial state of the system and has to make intelligent guesses, from the information received from other nodes, about the rest of the system. An example of such an application is a dynamic routing algorithm for store-and-forward networks.

Distributed Programming System. This is a distributed version of a general purpose time sharing system. The assumption is that it is not possible to restrict in advance the modes of sharing among users. It is necessary to communicate both data and programs, but from the point of view of the mechanics of the actual exchange of information this type of system could be included in the first category.

The distribution can take place along two main lines, based on functional separability or on the non-uniform distribution of the use of databases. Functional distribution means that different nodes support different services. Such systems seem natural for control of industrial processes, where different nodes control different parts of a process, or in such systems as aircraft, where different nodes process information from different sensors. However, this approach seems also to be advantageous in service sectors such as banking [SVOB78A].

Database distribution characterizes systems where an individual processor supports the same services but on a different part of a database. A typical example is a bank with many branch offices. Each branch has its local accounts, but it should be able to serve a bank's customer whose account is at another branch. Since such remote requests are much less frequent than manipulation of the local accounts, partitioning of the bank's accounts database (that is, maintaining accounts on a computer at their local branch) is a natural approach.

It must be said that the division between functional distribution and database distribution is not clean; in most cases, a distributed system will to some extent include both. The latter case, however, implies an <u>integrated</u> database, while in the former case (functional distribution) the databases used by individual servers are much more independent. The functional distribution is the more general case. In either case, a distributed database represents a special problem, the need to enforce consistency constraints that span several nodes. It is not clear how often this problem actually arises, but it cannot be ignored.

It can be concluded that the basic paradigm in the class of distributed systems that our project is addressing is the invocation of remote servers. This can be viewed as a communication protocol of much higher level than, for example, the host-to-host communication protocols currently employed in he ARPANET. The implementation of such high level protocols, however, may need to differ, depending on the type of application, and possibly on the efficiency and reliability requirements of the application. Therefore, we should not aim to design such high level protocols, but instead develop a set of <u>tools</u> that facilitate design and implementation of such protocols.

Finally, an application study by d'Oliveira [DOLI77] revealed an important result: there are strong pressures toward decentralization for sociological and political rather than technical reasons. We infer from this study that decisions about the distribution of information among the various nodes will be made for external reasons that only the application itself can specify. Thus, the application builder must have control over and understand the placement of information.

3. THE TARGET OF THE PROJECT

To summarize, we view a distributed system as a collection of autonomous nodes that communicate only by information exchange over the communication network that connects them. In such a system, at least two levels of coherence must be enforced. One level is the application level itself. The second level is the set of internode communication protocols that facilitate the physical exchange of information (packets of bits). But there is a large gap between the application and the low level communication protocols. Usually, this gap results in a rather ad hoc implementation of the application.

Our target is an intermediate level, called the programming system, which will support a well-structured design, implementation, maintenance and control of distributed applications. This level is more than a programming language in a traditional sense. Rather, this level is envisioned as a set of tools that include primitives found in conventional higher level languages such as Pascal or PL/1, but also primitives normally assumed to be part of an operating system, for example, long-term storage and cataloging of information or control of protection safeguards. Thus, this programming level will integrate the programming language and the operating system. More strongly, this level will integrate a programming language and a distributed operating system.

The design goals for the programming system include:

Aim for as high a level as possible, but application independent. Our system is intended to be used to implement many diverse applications, for example, both command and control systems and administrative systems like inventory control systems. To adequately support such a class of applications, the language should be as high level as possible but general purpose. One need that all applications share is the ability to exchange potentially quite sophisticated messages.

Support well-structured programming. Since our primary motivation is to ease the task of the application programmer, we feel that the embedded language should borrow from existing language work, in particular building on languages such as CLU [LISK77] and Alphard [WULF76], which aid in the production of well-structured programs by providing powerful abstraction mechanisms. Of particular importance is the data abstraction, which consists of a set of objects together with a set of operations that provide the only means for manipulating those objects. Data abstractions have been investigated so far mainly in the context of centralized processing. We believe that they will be even more useful in the type of distributed systems assumed in our work, because they provide a powerful tool in organizing a coherent structure where the data of the application and the allowed distributed sharing is described in application-oriented terms, independent of the idiosyncracies of the individual autonomous nodes.

Since we are dealing with a distributed environment where an operation defined on the application level may require the assistance of several nodes, the language must support concurrent activities (process abstractions). Extensions of sequential languages will be necessary to achieve this. To enhance ease of use, we will keep the language as conventional and conservative as is consistent with our other goals.

Support communication in terms of abstract objects. Autonomous program units need to communciate in terms of the kinds of high level objects they manipulate. For example, the ARPANET supports one sort of "high level object", the ASCII file, but any other form of data must be transmitted as a sequence of bits and explicitly transformed from one representation to another by a user written program. The language should support communication in terms of abstract objects, regardless of the relative location of the sender and the recipient of a message. Two advantages arise from this approach. First, a clear statement can be made about the properties of data that the units depend on. Second, it is clear how to accomplish the processing that is needed to translate an object in memory into a message transportable by the communication network and vice versa: the translation is accomplished using special operations of the object's type. Note that this kind of translation is always needed; a language that requires messages to be composed of low level objects simply obscures this fact.

Allow explicit control of the application distribution. Conceptually, the target level can be viewed as an abstract network of processes where application-defined processes communicate via messages that contain high level commands, data and responses. In an ideal situation, this is all that would need to be seen by the application programmer. However, underneath this abstract network is the set of physical nodes and the communication lines that connect them. Our study of applications has indicated that the mapping of the objects used by an application into the physical set of nodes has to be made visible to the application programmer. We are assuming that objects to not move dynamically from node to node, depending on the degree of demand (such dynamic migration is often assumed in the "distributed" systems consisting of many, relatively tightly coupled, mini or microprocessors). Rather, when a specific node is chosen to be the (new) home of a particular object, an installation of the object has to be explicitly requested using commands provided by the programming system. This assumption is based on the belief, discussed earlier, that many of such placement decisions will be based on non-technical factors external to the system [DOLI77].

Support sharing. The programming level must support sharing of information represented as objects that reside at different nodes and belong to different users, where how the information is to be shared is defined by the application. An important aspect of sharing is to provide controls that regulate the patterns of sharing so that protection and synchronization constraints are properly met. It is also necessary to solve problems of naming across nodes.

Support reliable (robust) operations. Reliability is one of the most important goals of our project. A distributed system, by its very nature, provides a potential for enhanced reliability. However, to exploit this potential, the system and the application have to be properly designed. An arriving message must be tested for integrity and authenticity, using a combination of automatic system features and application dependent procedures, and there must be control over timeouts and the number of retries for messages sent but for which a reply has not been received. It is also desirable to have a means for specifying that an online backup copy is requested for an object.

Support changing patterns of use. We cannot expect an application to be written once and never modified. First, the system will grow by the addition of new nodes. Second, new patterns of use will arise involving existing or new pieces of information. Thus, we can expect synchronization and protection constraints to change with time. This change must not cause upheaval in the design of existing parts of the application.

We want to emphasize that the envisioned programming system is not intended for the end user, but for the application builder (programmer), although in some environments (such as LCS) there is often little distinction between the two classes of users. Also, it should not be necessary for all nodes in the distributed system to support the full language; each node need only support the appropriate (high level) internode communication protocol.

4. COHERENCE VS AUTONOMY

Since autonomy is such a basic property in our model of distributed systems, a natural question to ask is how much the coherence that we strive for in our project constrains the autonomy of individual nodes. The last section indicated that at least three levels of coherence are required:

1) application level

2) programming level for distributed applications

3) inter-node communication protocols

The need for coherence on levels 1 and 3 is unquestionable. However, level 2 may not be necessary, especially if the system is designed for a single specific application. Level 2 has been introduced as a result of a desire to provide an effective programming environment for implementation of a variety of distributed applications. Thus, the above question can be restated as: does this intermediate level impose any additional constraints on the autonomy of individual nodes? We believe that it does not add new constraints, but merely divides the constraints into two categories: the constraints that have to be observed by the application programmer, and the constraints that the application programmer no longer has to be aware of, since the programming system handles them for him.

Basically, autonomy has two different aspects:

Operational Autonomy. A node can operate (at least to some degree) even while it is completely cut off the rest of the system.

Administrative Autonomy. The owner of a node can exercise a certain degree of control over the use of the node even while the node participates in a distributed application.

Both of these aspects have to be considered in the design of a distributed system; both interfere with the goal of coherence.

A good discussion of the mechanisms needed to solve the problem of operational autonomy can be found in [MONT78]. Administrative autonomy is a broader and less understood issue. On the application level, administrative autonomy is associated with an individual or a group of people, users or project managers. Potentially, more than one autonomous project or service could be supported on the same node, or several nodes could form such an autonomous unit. The coherence required to join such autonomous projects or services is determined strictly by the application; there are also many situations where, on the application level, local autonomy is completely supressed. Thus, the application limits the autonomy affordable in the individual nodes. The notion of administrative autonomy needs to be defined more formally; at the present time, only some practical examples of the possible manifestations of autonomy are given:

Selection of the hardware components and their configurations. Even if the overall system has to be homogeneous at the machine level, it is still necessary to select for each node a particular processor the amount of main memory and secondary storage, etc. Such decisions can be entirely autonomous; they weill depend mostly on the type and the amount of work to be done at the particular node, rather than being dictated by the obligation to support a common high-level language.

Local Software. In addition to supporting the common programming language and the inter-node communication protocols, additional software for local needs can be developed, maintained, and run on individual nodes. Also, the implementation of the common programming language and the communication protocols can differ from one node to another, as long as the implementation conforms to the specifications. It also may be possible that some nodes need not support more than only a small subset of the common language in order to be able to handle their parts of the distributed application.

Naming. Each node can have its own independent way of naming its local objects. Lower level names may be completely inaccessible from outside the node. An application, however, may enforce a uniform one-level naming space. Alternatively, a structured name space may be used, where each name consists of the name of the node containing the object and the local name for that object. If the fact that the system is distributed must be invisible to the end user, the name space seen by the user must be one-level. Somehow, however, this type of name must eventually be translated into a structured name space to locate the desired object. Thus the extent to which the autonomy in assigning names is constrained depends on the application.

Availability of local resources. Each node can control access to its private objects (not part of the common application). Local control of objects that are part of the common application may be restricted by the application. Similarly, the need to use local hardware resources imposes constraints on the individual nodes; a node may not be free to refuse a request to perform some service in order to do some local work or disconnect for maintenance. While these constraints, due to the decentralized control of individual nodes, cannot be enforced physically at the moment of crisis, they can be enforced outside of the system, e.g. by legal means.

5. MODEL OF DISTRIBUTED COMPUTATION

This section presents a model of the universe of entities that take part in a distributed computation. We assume that each entity has an identity that is permanent; an entity can be referred to by giving its name. We are not concerned here with all aspects of the behavior of entities, but rather limit our attention to questions concerning the locations of entities within the network and how entities can refer to other entities.

5.1 Types Of Entities

The basic model of distributed computing provides two different kinds of entities: processes and everything else. A process is active, and is thought of as being the execution of a sequential program. Non-process entities, which we will call objects, are passive, i.e., they do not originate any activity. Examples of objects are integers, arrays, stacks, procedures, etc. Objects have a state (value) that may change. If the state can change during the object's lifetime, then the object is mutable.

A process can communicate with another process by sending it a message. We assume that the syntax and semantics of message passing is independent of the nodes of residence of the two communicating processes (although certain optimizations can be performed by the system if both processes reside at the same node). A process can use an object by performing (invoking) an operation on it (or by invoking it if it is a procedure); again, the semantics of invocation is the same regardless of the nodes of residence.

The decision about whether the entities used in the model of computation are all uniform or whether the model distinguishes different classes of entities is a fundamental one. Basically, the uniformity concerns the ways in which entities may be used (and may use other entities). In our model, two basic primitive operations are used: invocation and message passing. We intend that the semantics of invocation be distinct from message passing; the following sections will clarify the reasons for this distinction.

An example of a computational model in which all entities are uniform is the Actor System [HEWI76]. In this system, every entity is an actor, and an actor is used by sending it a message that is also an actor. There is only one basic primitive, message passing, so our model seems more complicated. However, we believe that it is more natural than the actor model and will therefore be easier for programmers to understand. If programs built out of actors are examined, it is clear that there are "data-like" actors, "procedure-like" actors and "process-like" actors. We believe these differences are fundamental and should be reflected in the language in its semantics.

5.2 Location Of Entities

The universe of entities is spread across the physical nodes that make up the network. One important question concerns the location of entities: is an entity permanently located at a particular node, or can it move from node to node?

To make a decision here, we must consider several issues:

i. Earlier we discussed our conclusion, based on an analysis of applications, that the application programmers must be able to control the location of entities. Note that, at the least, this conclusion precludes automatic relocation of entities by the system, although relocation under program control would still be possible.

ii. We are assuming that nodes are autonomous and possibly heterogeneous. Even under program control it is possible to move an entity to an autonomous node only if that node is willing to accept it. Furthermore, if that node is different from the current home node of the entity, considerable translation may be needed to effectively move the entity.

Therefore, we believe that entities should have a <u>permanent location</u> at some node in the network. An entity comes into existence at some node (when it is created) and remains at that node until it is destroyed. Moving an entity can be accomplished by having a program create a new entity and letting it "take over" from the old one; however, the relationship between the two entities is not recognized by the system, and represents a higher level concept of identity than that introduced above.

One consequence of this decision is that it will be easy for the system to create unique names for entities and to interpret entity names, since the node of residence can be part of the name.

5.3 Restrictions On Referring To Entities

An entity may refer to another entity by using or containing its name. For example, a process will have local variables that may contain the names of other entities (both processes and objects); as the process executes, it can use these names. A data object is represented by some storage (at its node of residence), and some of this storage may contain names of entities (again both processes and objects).

In our model, the universe of objects is divided into mutually exclusive sets. A process is associated permanently (that is, for its lifetime) with a single specific set. The objects in this set are <u>private</u> to the process; the set forms the <u>local address space</u> of the process.

We have chosen the following restrictions on how the entities of the model can refer to each other:

i. a process can refer directly only to its private objects

ii. an object can refer only to the objects that belong to the same local address space.

There are no restrictions on referring to processes: both processes and objects can name other processes.

The above restriction can be enforced as follows: messages can contain the names of processes but not the names of objects. A model obeying this restriction is illustrated in Figure 1. The nodes labeled Pi are processes, while nodes labeled Oi are objects. Two kinds of directed arcs are shown. A solid arc from entity x to entity y means y is a process and x names y, while a dashed arc means y is an object and x names y. A process may (ultimately) refer to an object in the course of its execution if there is a path from the process to the object consisting entirely of dashed arcs. The set of all such objects form the local address space of the process.

Our further restriction is that all objects in the same local address space must reside on the same physical node. Alternatively, a process could be allowed to perform an operation on an object whether that object resided at the same node or not. Invocation of an operation on a remote object can be made to work,* but has the disadvantage that what appears to be a simple invocation will involve internode communication, and therefore, can take a long time, and may even have slightly different semantics. In particular, a local invocation has two possible outcomes:

i. the operation completed successfully

ii. some error occurred.

For an operation on a remote object, however, another kind of outcome is possible, in particular, if no reply is received from the remote node, it is simply not known what has actually happened. The message to or from the remote node could have been lost, or the node could have failed before, after, or in the middle of processing the request.

* The invocation must take place at the object's node, since as discussed in the preceeding section the object cannot move to the invoker's node.

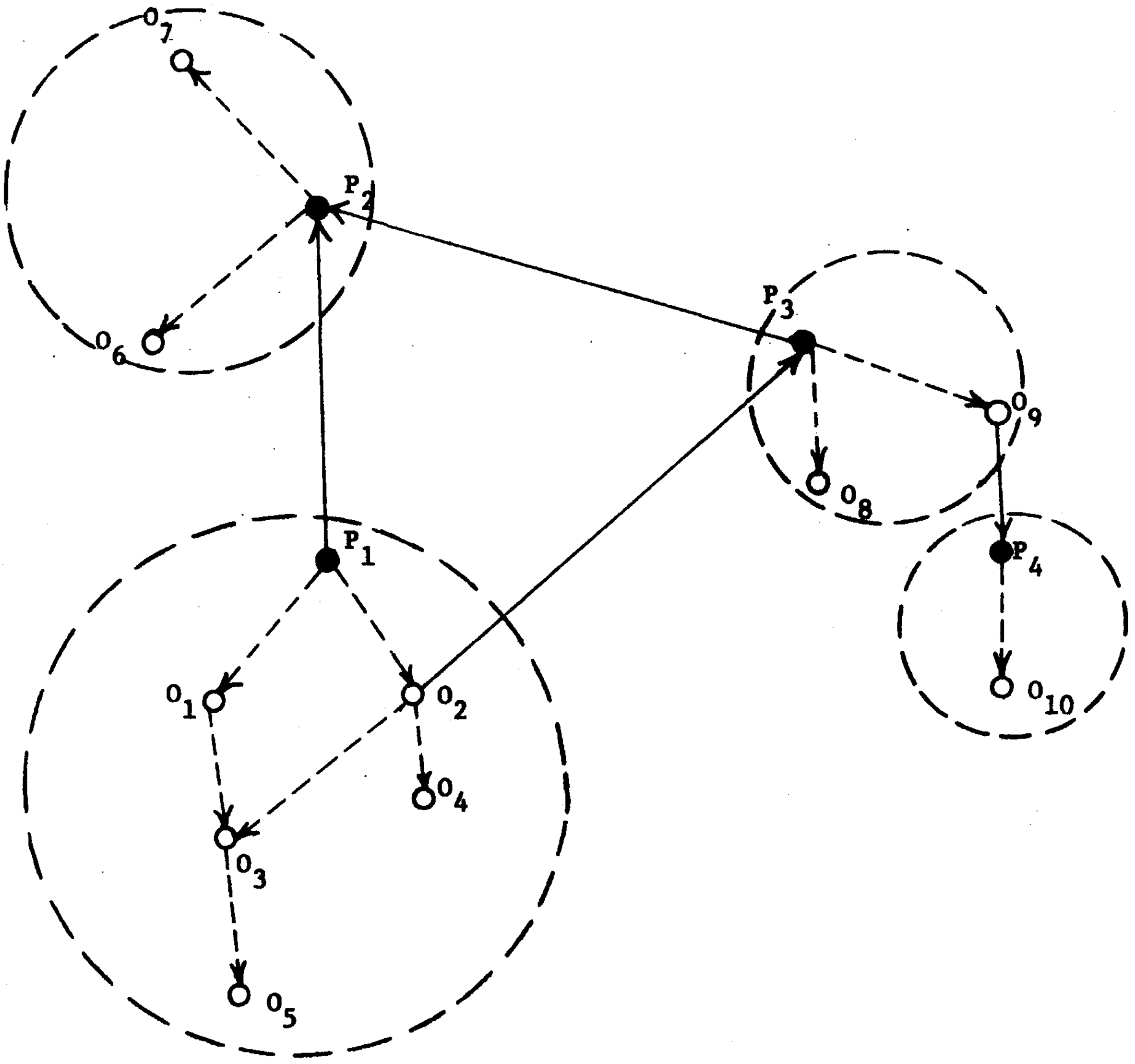


__Figure 1__: **Example of possible relationship of processes and objects.**

5.4 Guardians

The model that we have developed thus far represents an abstract network. Processes are the nodes of this abstract network; each process, analogously to a physical node, has its private memory and can communicate with other processes only by sending messages.

The abstract network model has several advantages:

i. The programmer organizes the locations of entities by considering where to locate the abstract nodes, (e.g. each process with its local memory). This seems easier than worrying about each entity individually.

ii. Operations are always invoked locally. This is simpler to implement than remote invocation, and also avoids some arbitrary time delays.*

iii. Management of storage for objects (e.g. garbage collection) can be done locally on each node.

Although two processes cannot refer to the same object, they can <u>share</u> an object if they both name the process that can refer to the object. Such a process will be called a <u>guardian</u>. A guardian may guard one or several objects; its job is to synchronize possibly concurrent requests to perform operations on the guarded objects. In Figure 1, P2 is a guardian for O6 and O7, which are shared by P1 and P3.

The abstract network model requires two extensions to be useful. First, the requirement that local address spaces of processes are disjoint may need to be relaxed. There are two reasons why it might be desirable to have several processes in a guardian. First, the processes could provide

* Of course, the operation activation might send a message, e.g. to some process whose name was contained in one of the input objects to this operation.

additional concurrency, which could be used to improve response. Second, a process could provide continuity. If a user and the guardian need to have a conversation, providing a process for the user to interact with would be a natural means of realization. The processes within a guardian would share objects directly. A guardian would be defined using a special syntactic construct, something like a serializer [HEWI77], that specifies the processes making up the guardian and their intercommunication; all the processes in the guardian would reside at the same node.

Second, in the case of a guardian that guards several objects, some efficient mechanism is needed that permits a user process to specify to the guardian the particular object of interest,* and for the guardian to determine that the object so specified is one it guards. Often no special mechanism is needed for this. For example, a query sent to a data base guardian names a record (or records) in the data base by means of a high level name. However, it appears that sometimes efficient, low level names are needed. Consider a guardian of a disk that provides logical tracks. When a user requests some data to be stored on the disk, the manager returns the logical track number where the data is stored. Later, the data can be read by sending the logical track number back to the guardian. However, the logical track number must be one created by the guardian, and also must still exist at the time of the read request.

The guardian construct can model different degrees of autonomy. For example, if a process requests an operation on data that are available only through the guardian, such a request may fail since the guardian may refuse to

* This should not be interpreted as a requirement that the system ought to guarantee that an object continues to exist as long as some user can specify it.

release requested data, or in some cases may even destroy the data at its own discretion. Also, a guardian does not have to know a priori about all processes that may request operations on the guarded objects. That is, a guardian can be a general server that accepts requests from any process or a class of processes.

5.5 Summary

In their recent paper, Lauer and Needham argue that message passing and procedure calling (invocation) are essentially equivalent, and consequently either can be used as the basic and only primitive in the implementation of an operating system [LAUE78]. As follows from our model, we believe that it is advantageous to have both. One reason is the semantic difference between local and remote invocations, as discusses earlier. However, the whole concept of the abstract network is based on the distinction between and the combination of message passing and invocation. The abstract network makes it explicit when an operation is to be performed on an object that belongs to a different set, where the difference may lie in the logical function of the set, protection constraints, or administrative responsibilities. For a distributed system that supports several different applications on highly autonomous nodes, the abstract network is an appropriate model.

6. RELIABILITY ISSUES

In the future, reliability will be one of the major issues in information processing sytems. This claim is based on two observations. First, the quantity of information entrusted to a computer system is ever increasing. Second, the complexity of the operations performed by a computer is also increasing. More and more organizations and systems are dependent on computer maintained information and a failure of these computer systems can often be critical. Thus high reliability is not just a requirement for real-time systems controlling space vehicles or industrial processes failure of which would endanger human lives.

Reliability of an information processing system is not merely a question of software correctness. Hardware failures, synchronization failures, and errors of the human users must be anticipated and handled gracefully. The only way to design a reliable system is to make it "fault-tolerant", or, robust in face of a large variety of internal failures and misuse.

Distributed systems are often claimed to be inherently more reliable than systems based on a large central processor. That is, given that a distributed system is properly designed, it offers better reliability. This claim is based on several factors. First, distributed systems by their very nature provide opportunities for redundancy. Second, error propagation is restricted by physical separation of processes and resources. And finally, individual nodes in the distributed system may be less complex than a large central processor and, as a result, ought to have lower probability of failures. Basically, distributed systems have a potential for being more reliable than systems based on a large central processor. However, this potential needs to be exploited through proper design.

For the purpose of the discussion of reliability issues, the implementation of the abstract network introduced in the preceeding section is divided into two levels: the applicaton level and the system level. The system level is all the mechanisms needed to support the view presented to the application programmer (that is, the hardware and software run-time support of the programming system). The level built on the top of this level using the tools available to the application programmer is referred to as the <u>application level</u>.

Reliability mechanisms are those mechanisms that assist in detection, reporting and recovery from errors and failures. An error is an internal state of an entity that, if special steps (recovery) are not taken, will result in a failure of the entity (or, in the case of data objects, failure of an operation on the entity). Some errors can be handled entirely by the entity itself, and thus remain invisible to other entities (e.g. users of that entity); such errors are said to be <u>masked</u>. Detected errors that cannot be handled internally should be reported to the users by signalling a <u>failure</u>. Undetected errors also turn into failures; it is possible that a user can detect this kind of failure, but the problem is much more complex than with the reported failures.

To achieve reliable operations from the application point of view, both the system level and the application level have to include mechanisms for detection and handling of errors and failures. For each type of error, it is necessary to decide where it can be detected and how it should be handled. Some classes of errors, detected within the system level, can be masked, but for others a failure has to be reported to the application level. Other errors are application dependent and therefore, their detection and handling must be done at the application level. Basically, in the case of system errors, there is a gray area where a decision has to be made as to whether these errors will be masked by the system level or reported as failures to the application level. It may also be possible that an attempt to mask an error fails; it is necessary to decide if and how many times the system should repeat the attempt before reporting a failure. The important factors in these decisions are the cost and complexity of the masking mechanisms on one hand

and the convenience to the programmer on the other hand. Both sides have to be carefully evaluated.

The system must provide a means for detecting and correcting or reporting errors arising from the operation of the hardware and the software that supports the application programs. However, the system also has to provide suitable primitives for the application programmer to facilitate handling of the application specific errors and communication of the system detected errors to the application programs.

6.1 Availability And Correctness

Reliability has two aspects that, unfortunately, cannot always be separated; in particular, their solutions may conflict. One aspect is the _availability_ of the entities needed to perform a specific task. The other is the _correctness_ of the available entities; a very important special case is the integrity of the stored information.

To assure correct operation, the system and the application must be prepared to handle errors that originate in lower levels, in particular, hardware faults and possible residual bugs in the software that comprises both the system level and the application level. It is also necessary to be prepared to deal with errors whose source is the user of an entity. Since the user may be a process running on another node, these latter errors may be caused by hardware or software failures in the user's node, or they may occur because the requesting process either doesn't know how to use the requested entity properly or is trying to misuse it intentionally. Thus, to ensure correct operation of an entity, it is necessary to ensure both that the operations on that entity are performed correctly in spite of possible failures of the node on which the entity resides, and also, it is necessary to defend the entity from possible misuse by other processes.

As described earlier, processes in different guardians can communicate only by sending messages. Objects can be manipulated only within their guardian. To protect an object from misuse, it is necessary to ensure that access is indeed limited to the guardian and that all incoming messages are carefully scrutinized to determine whether the request is reasonable and the effects of performing requested operations. Within the guardian, it is necessary to provide mechanisms that will protect the resource from being damaged or lost (made inaccessible) due to errors arising from the faults in the structures that implement the guardian.

Availability is constrained by two factors:

i. the efficiency of the system, that is, the actual physical delay and queueing time in the abstract network, and

ii. failures in the abstract network.

Availability has several connotations. Firstly, it is used to indicate whether an entity is <u>useable</u>, that is, if the respective process will execute the operation requested once the request is brought to its attention (e.g. gets to the head of the queue). Secondly, it is used to indicate whether an entity can be used immediately, or whether there is a contention for the entity. Thirdly, and this aspect plays an important role in a distributed system, it may be used to indicate whether an entity is <u>accessible</u>. An entity may be useable and unused, but the path to it may be broken.

It is possible to translate all three aspects into the problem of how long it is necessary to wait for a resource. A useable entity may not be immediately available due to contention for the entity, but also due to long communication delays; if the communication path is broken, the communication delays may be unusually long, even infinite. Similarly, if an entity is unuseable, the wait time for the entity to become useable may be very long, possibly infinite. Since in a distributed system it is not always possible to determine the cause of a long delay, the system may have to respond to poor performance (due to overload) in the same way it responds to functional failures of the resources and communication paths. Thus, in a way, poor performance (due to overload) or turns into a failure!

From all three of these aspects, availability can be enhanced if several instances (copies) of an entity are maintained at different physical nodes:

i. Coping with failures. If a node fails, or communication with a particular node fails, it should be possible for processes at other nodes to continue. That means that entities provided by the failed (or inaccessible) node have to be provided by some other node(s) in the remaining operational network (each operational partition of the network).

ii. Coping with bottlenecks. Even if the nodes and the communication network of a distributed system never fail, a single instance of an entity may not provide sufficient availability. An entity may become a bottleneck; also, the communication delays, especially in a long-haul network, may be substantial, and it thus may be desirable to have a local instance of the entity (and, consequently, support multiple copies).

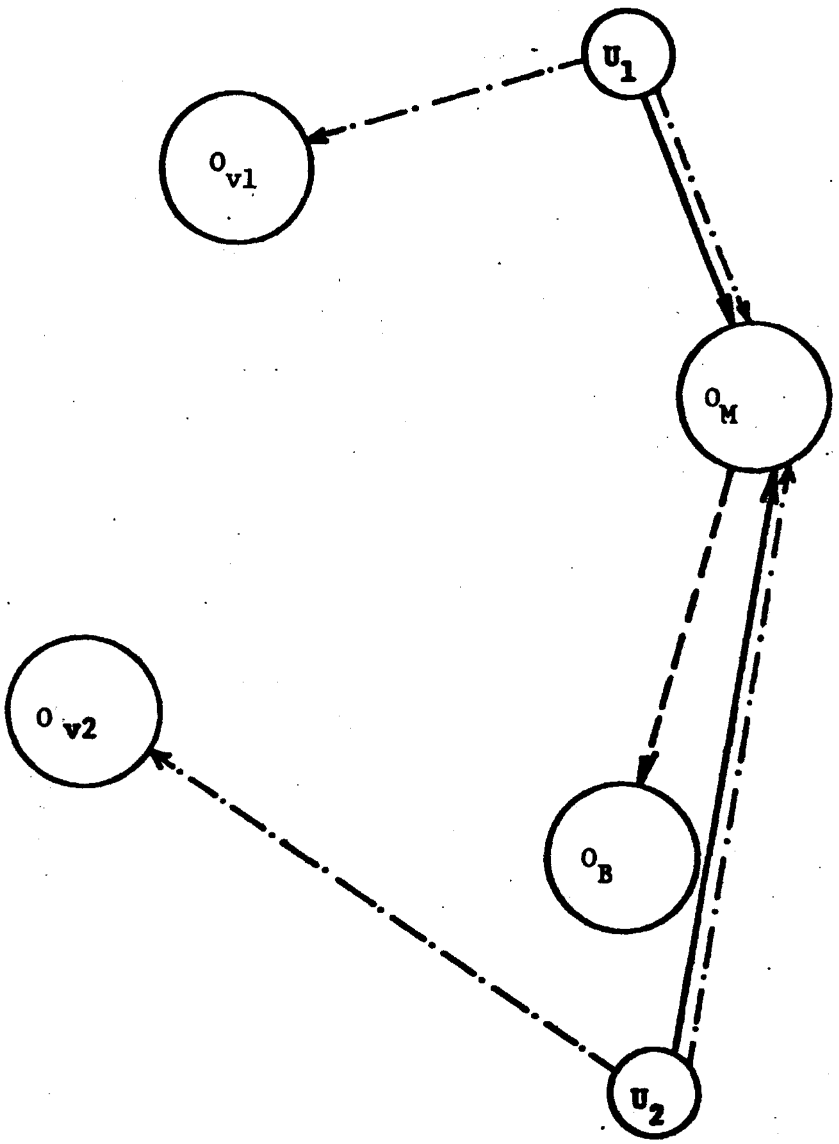
In the systems under consideration, the most important type of object is a data object. Maintaining multiple copies of data objects that need to be frequently updated represents a special problem. It is important to distinguish between failures and bottlenecks since the right solutions to the problem of mutual consistency are significantly different. In the first case, only one copy needs to be actively used, that is, an object has a master copy and one or more backup copies. The changes made to the master copy must be propagated to the backup copies, immediately if every state of the object must be recoverable, or periodically upon special command if in a case of a failure it is sufficient to back out ot some consistent state, not necessarily the last consistent state. In the second case, all copies must be available for active use. It is often assumed that all copies must always be the same, but this requirement may defeat the very purpose for which the multiple copies were introduced: reduction of delays. The delay caused by synchronization of updates with other updates and accesses of multiple copies may exceed the delay that would result if only one copy were maintained. However, it is not always necessary to have the most current version of an object; the information obtainable from an older version may be entirely satisfactory. Thus it seems much more realistic to allow for multiple versions of an object; the local copy may not always be the most current version, but the most current version is known and a local copy of it can be obtained upon request.

The system level ought to support, in a selective way, the kind of redundancy required to cope with failures. The other case is more complex and more application dependent. The fact that there exist several versions of an object may need to be visible not just to the application programmer, but to the application user. Thus, the solution should be left to the application level; the system, however, ought to provide mechanisms to make the solution possible. A mechanism for maintaining multiple versions of objects in such a way that a consistent version of a set of objects can always be obtained was developed by Reed [REED78]. In addition, Reed's mechanism solves the problem of updates and backout in a distributed system in a most natural way.

However, to prevent loss of information, the most current version ought to have at least one backup copy. The scheme that combines multiple versions and backup copies is sketched in Figure 2.

6.2 Reliable Communication Subsystem

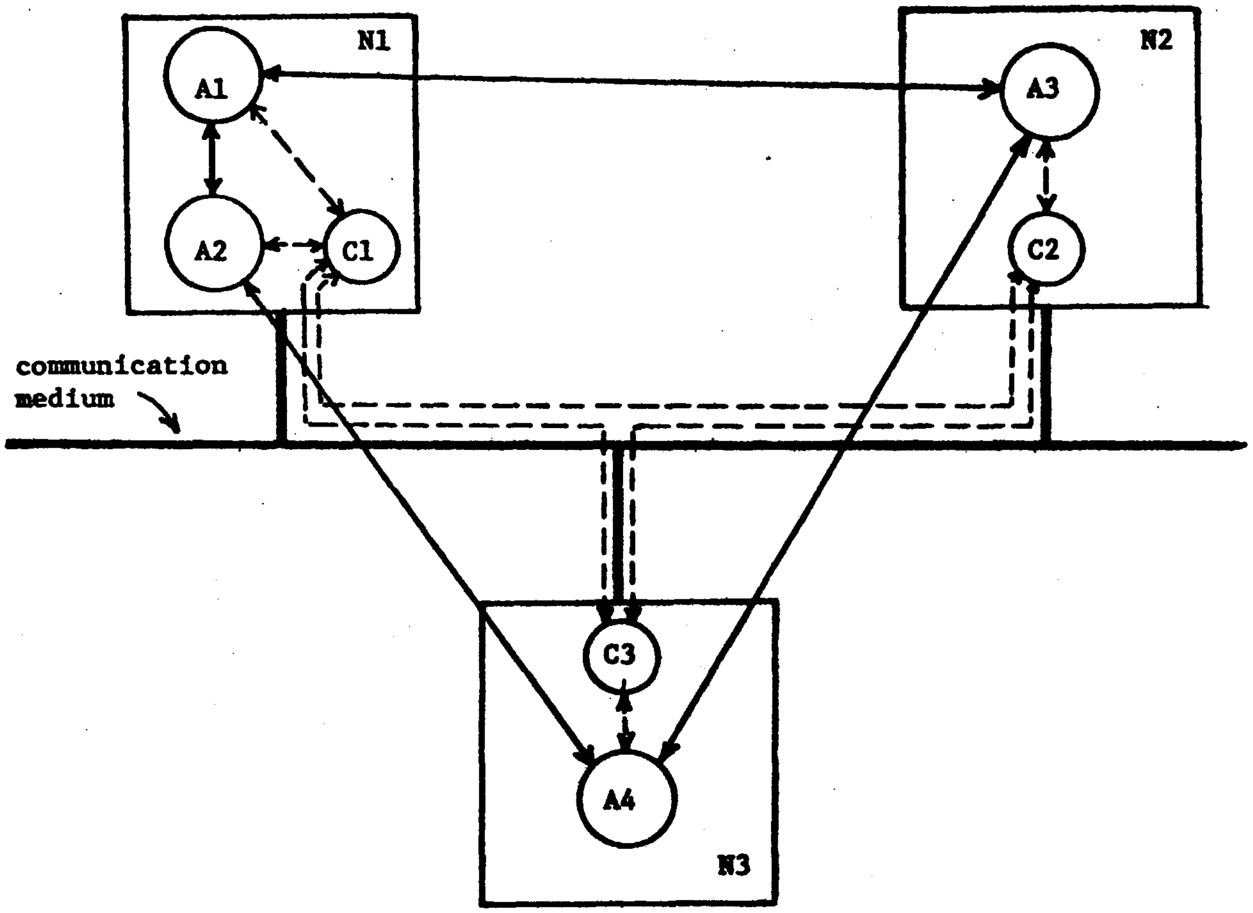
The communication subsystem is the part of the system level that delivers messages between physical nodes. This subsystem consists of the actual physical network of communication lines and of communication processes that control the delivery of messages. Figure 3 shows relationship between the abstract network and the communication subsystem. The application processes exchange messages that, logically, contain values of high level (abstract) objects meaningful at that level. The values of these objects have to be translated (encoded) into a string of bits for delivery to another node and decoded into the proper abstract objects at the receiving node. At the system level, messages, now in the form of bits, may have to be partitioned into packets. The messages are checksummed, so that transmission errors can be detected. It is difficult to correct transmission errors at the receiving node, since transmission errors are bursty (affect not just a single bit, but several bits). Checksum facilitates detection of errors, where the number of detectable simultaneous errors is determined by the size of the checksum field. Correction is performed through retransmission. In general, once a message has been translated into a string of bits, the protocols used by the communication processes should take care of the correct transmission, that is, either a correct message is delivered, or nothing is delivered. However, the primary responsibility for checking that a message has been acted on, that is, ensuring that a process that sent a message will not wait indefinitely, and



O_M master copy of the object (current version)
O_B backup copy of the object (current version)
O_{vi} object version i (current or older version)
U_k users

$\longrightarrow$ update requests
$\dashrightarrow$ propagation of changes
$-\cdot-\cdot\rightarrow$ read requests

Figure 2: Multiple version scheme with backup.



Ai	application processes
Ci	communication processes
Ni	nodes

<——> possible communications on the application level
<- - -> flow of information in the network

Figure 3: The abstract network: communications on and between the application and the communication subsystem.

also that the message contains values acceptable from the application standpoint,* must rest with the application.

One of the most difficult problems in this type of distributed system is that unless an explicit reply (or an acknowledgement) is received, it is impossible to determine with certainty whether a message sent to a process at a different physical node has been received and if the receiving process has acted on it. The only defense against possibly waiting indefinitely for a response is to use a timeout mechanism. The sender of a message can specify a time interval after which it gives up waiting for the response; the timeout mechanism will alert the sender when such a time interval has elapsed. The possible reactions of the sender to a timeout event can be divided into two categories: the sender decides to give up the attempt to communicate with the particular process, or, the sender decides to resubmit the request. Because of the uncertainty discussed above, it is possible that the first request will eventually be processed. Thus, in the first case, the request may be processed in spite of the sender's decision not to continue and may conflict with the subsequent actions taken by the sender after the timeout. In the second case, the same request may be processed twice, possibly leading again to an inconsistency. Thus, in situations where an inconsistency may arise form such internode requests, it is necessary to use special (often complex) protocols [LAMP76, GRAY78, REED78, MONT78, TAKA78]. The question that arises at this point is if such protocols ought to be a part of the communication subsystem. We believe that many such situations represent special cases that

* A message may contain a higher-level error: either a message has not been constructed properly by the application process (wrong command or wrong data) or the translation from abstract data to the bit representation has not been done correctly.

are more appropriately handled at the application level. However, our language should provide constructs for detection of duplicate messages at the application level.

The reliability of the communication subsystem could be increased through the use of recoverable queues. That is, in addition to dealing with the communication errors that result in a loss or garbling of messages sent across a physical communication link, the communication subsystem can guarantee that messages that have been presented to it by the application processes and _queued_ for delivery (that is, messages accepted by the communication subsystem) will not be lost if the node fails. This degree of reliability may be important if translation from an abstract data object to the corresponding bit representation is a costly operation, or if the input to such a translation step is not automatically repeated (e.g. message typed by a user). This argument can be extended to the requirement that the communication subsystem should _guarantee delivery_ of all messages it has accepted from the application processes. That means that in addition to providing recoverable queues for messages that have not been sent yet, the communication subsystem must continue trying to send the queued messages until it eventually succeeds. At the receiving node, the messages have to be stored again in recoverable queues until they are picked up by the target application process. Unless the target node is disabled permanently, it is possible to ensure that each request either will be processed or it can be determined that the request could not be processed and why. Of course, the sending application process may have to wait for a long time for the outcome of its request, so it may still be desirable to use a timeout on the application level.

The reliability mechanisms do represent potentially large overhead, and their use should not be imposed on all communications. The basic communication scheme should be simple, fast, and inexpensive. Rather than insisting on the guaranteed delivery scheme, we will investigate whether it is possible to vary the degree of reliability provided by the system by letting the application programmer choose from several different protocols, where such protocols would be implemented as extensions (abstractions) built from the basic protocol.

7. PROTECTION ISSUES

In the class of distributed systems considered in our project, a likely case is that a particular node is utilized by one user or at most by a set of cooperating and mutually trusting users. In this case, intra-node protection mechanisms are not required to have power sufficient to protect against subversion and malice. This is in strong contrast to a system such as Multics [SALT74], and many other time-shared and multiprogrammed systems that were designed to operate properly with a set of mutually hostile users. What is required within a single node is a mechanism that protects against error and forgetfulness. Inter-node protection, on the other hand must be able to deal with the potentially hostile environment: 1) individual nodes are autonomous, that is, it is not possible to assume that they will behave as desired by other nodes, and 2) the communication lines between nodes in general cannot be physically secured.

We assume that a capability mechanism will be the basic mechanism used for intra-node protection. By capability we mean an unforgeable identifier for an object that identifies the type of the object* and that must be presented as part of addressing an object. By constraining a procedure to execute with a limited collection of capabilities, it is easy to guarantee that the procedure will not do arbitrary damage to stored information.

In the context of our model, the efficiency of capabilities in comparison to an alternative mechanism such as access control lists becomes very important. Since we assume a world with a large number of small objects, it is clearly impossible to imagine that every object comes complete with an access control list; the overhead of the access control might be substantially larger than the object itself. Capabilities, on the other hand, need be no more than slightly enlarged addresses. We thus propose that the intra-node protection mechanism is based on capabilities, with some sort of capability cataloging mechanism playing the role now associated with the traditional file system.

The inter-node protection is more a matter of policy than of mechanism. We believe that protection between nodes should be based on an access control list mechanism rather than a capability mechanism. This claim is not based on difficulty of implementation; either mechanisn can be imagined. Rather, it is based on our perception of the high level needs of distributed applications. A fundamental way to characterize the difference between capabilities and access control lists is that capabilities do not provide any easy answer to

* "Capability" if often used to mean more than an unforgeable identifier: a capability may also include a specification of the access rights, that is, a specification of which of the operations defined for the type of the object in question are acutally allowed on that specific object. Alternatively, access control could also be achieved by making the object appear to be of the type that imposes the desired restrictions. The desirability of including the access rights in the capability and the feasibility of the other approach (especially in connection with providing different "views" of data bases) will be investigated.

the question "Who are all the people who can get to this object?", while access control lists make it very difficult to ask the question "What are all the objects that I can get to?". If one considers real world protection problems, including those drawn from domains other than the computer domain, the more workable model of protection generally turns out to be that based on access control lists. While capabilities are often used in the real world, the most obvious example being keys, the drawbacks are well known. Keys are subject to unauthorized duplication, loss, theft, etc. More relevantly, capabilities (or keys) do not provide a means to support accountability.

7.1 Protection In The Abstract Network

The intra-node protection problem, while less severe than the problem that results from fully suspicious cooperation, is still not trivial. The programmer must be provided a means of partitioning his computations, so that certain objects are accessible only in certain computations. This mechanism will allow him to debug new versions of software without running the risk of destroying existing objects; however, such a mechanism is also desirable after the debugging stage, to limit the effect of undetected (unreported) failures that may arise from residual bugs or hardware failures.*

The abstract network presented in Section 5 assumes such a partitioning: the local address spaces of individual guardians define protection domains that are assumed to be mutually exclusive. Thus, the guardian (abstract node) is an important logical unit of protection. Within a guardian, it is sufficient to provide only the simplified capabilities discussed earlier. Between guardians only messages can be exchanged; there is no physical sharing

* This subject is also included in the discussion of reliability issues. Indeed, there is a strong overlap between reliability and protection, both in their definitions, and in the mechanisms used.

of data. A guardian can scrutinize all incoming messages, and, using access control lists and authentication, can validate the request. However, not every guardian need use this form of protecton. Just as the implementation of a procedure can make use of procedures, so may the implementation of a guardian be defined in terms of other guardians. Thus, some guardians are more autonomous than others; by design, some guardians will receive messages only from "friendly" guardians. Therefore, the system cannot automatically enforce protection constraints either at physical node boundaries or at abstract node boundaries.

An important consideration in maintaining autonomy of the individual nodes in a distributed system is the control over installation of the guardians. We are assuming that guardians are always created at the same (physical) node as the creating guardian (process).* Remote guardians can be created only be requesting some already existing guardian at the remote node to do the creation. This approach implies that each node must come into existence with (at least) one guardian, the <u>primal guardian</u>. The primal guardian is tailored by the owner of the node, prior to adding the node to the network, by specifying who has the right to install guardians at that node as part of a particular application.

7.2 Protection Agents

Let us look now at the inter-node protection problem from a slightly different viewpoint. It will be a rare case where a request occurring between nodes consists of nothing more than the reading or writing of a single primitive object. In most cases, we can expect the request to be composed of an aggregate of reads and writes on various objects, which the requesting node

* The same is true for objects.

views as atomic. This is generally referred to as an atomic transaction. The thing that must be protected from outside is the right to execute this atomic transaction. It is possible that the isolated reads and writes required as part of this transaction are not legitimate for the outside user except as a part of this or some other transaction. A typical example is a situation where the user is allowed to see some statistics but not the individual items in the actual set from which the statistics are divided.

To express such higher level protection constraints concisely, the language used to describe the inter-node requrests must have primitives that will it easy to confirm whether or not a proposed transaction falls within the bounds of the outstanding protection constraints. Alternatively, it is possible to visualize inter-node messages that contain an arbitrary algorithm expressed in terms of a sequence of operations on a set of objects; such an algorithm would have to be examined and confirmed at the receiving node before execution to ensure that it conforms to the higher level security constraints. However, the construction of a verification algorithm that ensures that an arbitrary program conforms to one of the number of high level protection constraints would be a challenge to the most optimistic of the program verification researchers; thus this approach must be rejected.

We postulate the idea, common in data management systems, that different users of a data base have different views of the data base, often called different data models. From the outside, the data model appears to describe physically stored information and the acceptable operations on it. However, internally, the data model may have little correspondence to the information actually stored. Rather, it may be realized as algorithms that derive the modeled data from the shared information. Thus, we see users first being divided into large groups, based on which data model they use, and then within those groups being further divided according to which operations they can perform on the data model provided. For example, some users may be able to read certain records, others to read and write them. Each data model implies the existence of an algorithm to translate between that data model and the stored information. It is these algorithms that must be provided in advance, one set for each data model. The programming system must provide facilities for creating such data models, mapping them into the actual stored information, and synchronizing read and write operations originating from different data models.

Higher level protection constraints can be expressed through the use of abstract types.* Users of an abstract object are permitted to manipulate the object only through the operations defined by the object's type; in reality, an arbitrary computation may be performed on a possibly large number of objects that constitute the representation of the given abstract object. In addition, individual users may be restricted to only a subset of the operations defined for an abstract type. However, the concept of the data model calls for yet something more. The traditional view of data models permits a low-level information entity to be shared by different users through a variety of data models. To support this view via abstract types, it must be possible to manipulate a single low-level object as part of a number of different abstract data objects, depending on the rights of the different users.** The idea of data models is that different users have different views

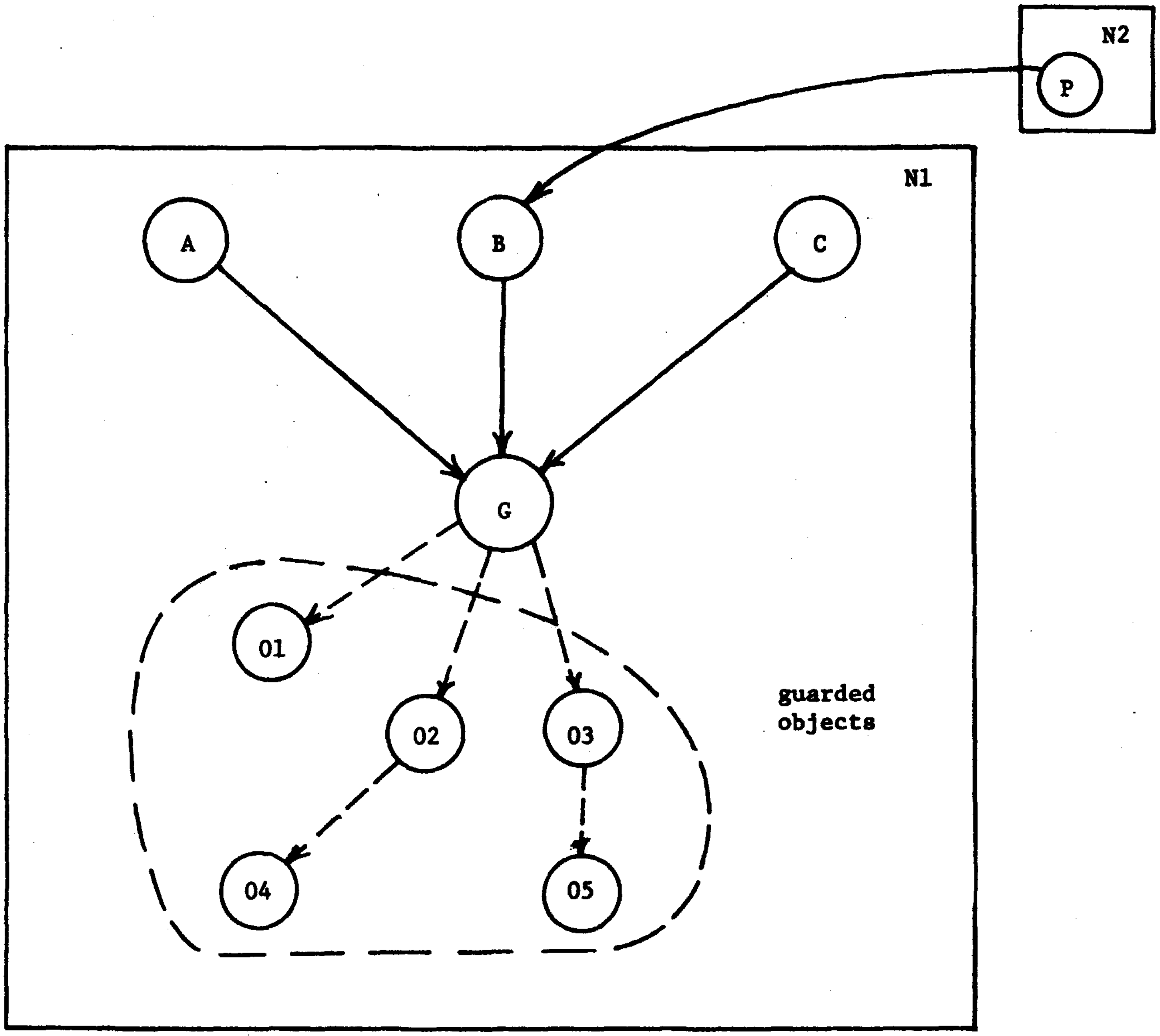
* Several research projects on the use of abstract types in implementation of data base management systems are under way [SMIT77, SCHM77].

** This can be a very difficult problem. In the context of relational data bases, for anything but very simple views, "propagation of updates from views to base relations becomes complicated, dangerous, and sometimes impossible" [CHAM75].

of the world, but fundamentally, they do turn out to be views of the same world. Thus, in some sense, they must ultimately rest on the same physical data.

The inter-node protection can be enforced as follows. Any outside user (process) perceives the information in a particular (abstract) node as a number of objects that he can manipulate independently, and a set of permissible operations on those objects. These externally visible objects are arranged in such a way that there are no explicit protection constraints that tie one object to another. A message arriving at a node to manipulate one of these objects must be processed by some active entity (e.g. a process) that confirms that the requestor of the action has the right to perform it, and then implements the operation at the node by invoking operations on other objects, not directly accessible from outside. We will refer to this entity as a protection agent. The protection agent could be the same entity as the guardian. The guardian, however, presents only a single view of the guarded objects. The protection requirements described above lead to a model outlined in Figure 4. In this model, the high level protection function is separated from the low level protection and synchronization function of the guardian. The protection agents represent different views of the guarded objects. The guardian controls the actual physical access to the guarded objects; it imposes synchronization constraints on requests passed to it from the protection agents.

While there are an infinite number of protection checks that the protection agent may wish to perform in a particular case, there are three checks that can normally be expected to occur. First, the protection agent will wish to confirm that the originator of the message has the right to invoke the protection agent at all. Second, the protection agent may wish to



A, B, C: protection agents
G: guardian

Figure 4: Inter-node protection mechanism: the remote process P (node N2) can reach the objects guarded by G (node N1) only through the protection agent B.

confirm that the particular object or objects involved in the operation requested are indeed accessible to the requestor, and third, that the particular operation to be performed on the object is permitted for the requestor. For example, a data base manager may wish to confirm that the requestor can invoke it, that the particular record being manipulated is accessible to the requestor, and whether or not updates or just reads are permissible for that requestor. In any particular case, one or more of these steps may be omitted. Thus, for example, provided that a user has a right to invoke a protection agent at all, he may have the right to manipulate any object normally made accessible through that agent. He may also be permitted to perform any of the operations defined on the objects. In this case, only the first of the three tests need be performed.

We should now pause and consider how this representation of protection meshes with the conclusion drawn earlier that inter-node protection should be expressed in terms of access control lists. Clearly, the use of access control lists implies that the protection agent must be able to determine reliably the originator of every message. Using the terminology developed for characterization of protection mechanisms in a centralized system, we will assume that every message, at its origin, has associated with it a principal identifier, which identifies the entity to be held accountable for the request in the message. Some technique such as encryption will be used to ensure the believability of the principal id by the recipient of the message, if the message has originated from a different physical node than the recipient [KENT76, NEED78]. Using the principal id, the first protection check described above is easy to implement. We can associate with every protection agent an access control list, and insist that the principal identifier associated with the message be on that list before the protection agent can be invoked at all. The second test, that of ensuring that this principal is allowed to manipulate the particular objects in question, can be handled in a variety of ways. One obvious technique is to associate with each entry in the access control list, a list of all the objects that the particular principal is allowed to use. The protection agent can then refer to the list to determine the access privileges of the requestor. If the third type of protection check is required, it can be implemented as part of this same list, by associating with the entry for each object a notation describing the operations that this principal is permitted to perform on that object.

8. LANGUAGE CONSTRUCTS

The desirability of various kinds of language constructs has been pointed out in several places in the preceeding discussions. We see the language design to be the main vehicle of our project. The language has to provide an effective interface for the application programmer, while at the same time it is necessary to take into consideration all the problems arising from the distributed processing environment. This section presents some examples of the planned language constructs.

Guardians will be a major type of module available for program construction. A guardian is a program whose purpose is to protect a resource. For example, a guardian might guard an entire data base, a partition of that data base, a physical resource, e.g. an I/O device, or an abstraction of such a device. The guardian may be thought of as a local collection of processes and data; the processes within the same guardian may share the data, but they communicate with processes in other guardians only by message passing. As was said earlier, a guardian may be implemented in terms of other guardians. The user of a guardian is aware of its behavior: the kinds of messages that can be addressed to (processes in) it, and the kinds of repsonses it makes to these messages. The fact that the guardian is implemented in terms of lower level guardians is invisible to the user.

An important issue in designing a language for a distributed system is the primitives for sending and receiving messages. The basic scenario in the abstract network is one process sending a message to another process requesting some action; later there should be another message, flowing in the other direction, indicating the result of the action. It must be possible to express in the language that a particular message is a reply to another message. In addition, it is necessary to address the problem that the reply may never arrive, or that the request message cannot be sent. Possible approaches differ in how long (for what event) the sending process must wait before it can proceed. Closely related to this degree of waiting is how the language will handle failures.

8.1 The Send Command That Waits For A Reply

In this approach, the sending process is forced to wait until the response comes back from the receiver, or some timeout or failure results. A possible syntax might be:

```
Send C(args) To A Timeout Time:
     R1(formals) Do S1;
     R2(formals) Do S2;
      ...
     failure (formals) Do Sfailure;
     timeout Do Stimeout;
     End;
```

Here A is a process and C(args) is the message, consisting of a command, C, and some arguments. The remainder of the construct lists the various possible responses, together with the appropriate action to be taken by the sending process. R1, R2, etc., are responses for A; some might be normal, and some abnormal. "Failure" covers various failures that are detected either by the system or by the receiving process A. The arguments of "failure" specify the type of failure. Some examples of a detected failure are:

a) the message as specified cannot be constructed,

b) the specified process (A) no longer exists,

c) the target node is inaccessible,

d) congestion (the target node of the target process (A) does not have enough buffer space),

e) the message cannot be decoded (the abstract objects contained in a message cannot be reconstructed).

Which of these failures are visible at the application level depends on the design of the system level. As discussed in Section 6, the system level might be designed in such a way that message delivery is guaranteed. This would eliminate the need to cope with the failures of the type c and d at the application level.

The timeout action is taken if "time" is exceeded. If the system is designed for guaranteed delivery, the timeout action that terminates the send command should release the buffers in which the system keeps the message for delivery to the target application process. It should be understood that this timeout is for the pair of messages to be exchanged between processes in the abstract network; a different timeout value is used in the underlying system to govern retransmission of packets.*

A different kind of "send" command is needed in the receiving process, since the receiving process must be able to respond to the command without waiting for the original sender process to respond back. To receive messages, A might use a construct:

```
Command case
          ...
          C(formals) Do ... reply R(args);...;
          ...
          End;
```

Here, A is waiting for one of a number of messages; if several are available, one is selected in a fair way. The message is then decoded, the contained data assigned to the formals, and the statements associated with the selected message are executed. The above form of Reply Command sends a message back to the process that sent the message; the identity of the process to reply to is automatically extracted from the received message. Another form of reply:

```
          Reply R(args) To B
```

that explicitly names the process to reply to will probably also be needed; this would permit a third process to be the replier to the original sender.

The approach sketched above has the obvious advantage of pairing sends and receives. It also has some obvious disadvantages. For one thing, there

* The "timeout" parameter presents a rather difficult programming problem -- it requires the programmer to be aware of the physical timing of the communication subsystem and the time needed by the recipient to process the request. However, unless the guaranteed delivery approach is adopted, this timeout is essential to cope with failures. Although, as argued in Section 6, some form of timeout is also desirable in the guaranteed delivery scenario, possibly a more intelligent way of specifying a timeout condition could be devised.

are two send commands. More important, however, is the loss of parallelism. If the sending process has other tasks to do while its request is being processed, it must either not do them, thus reducing efficiency, or it must spawn another process to do these tasks. Thus a language supporting this approach must provide rich facilities for parallelism.*

8.2 Separation Of Send Command And Reply Processing

The second possible approach is the opposite of the waiting approach: the sending process does not wait at all but continues running, performing actions on local objects, or possibly sending more messages. Only when it needs a response that is not ready, it will have to wait. The language now has to provide additional constructs that allow the programmer to distinguish which response goes with which request and to specify that the process wishes to wait for the reply to a specific message, rather than a reply to any message that may have been previously sent.

There are various ways in which these problems might be solved. For example, send commands might be labelled:

```
S1: Send C1(args) To A1;
S2: Send C2(args) To A2;
        ...
    Get S1 Replies:...;
```

In this approach, each Send Command has a continuation (as in Actors [HEWI76]) that can be named to identify the responses of interest in Replies. Following Replies Would be the list of alternative responses, as shown in the preceding section, to the message sent by statement S1. Note that the errors arising in

* Note: this is not the only reason for which such facilities for parallelism might be needed. See the discussions of guardians.

turning Ci(args) into a message would be exceptions (abnormal terminations) of the Send Command; failures such as (c), (d) and (e) described earlier would have to be reported outside of the Send Command (in Replies).

Another possible approach is to use ports:

Send C(args) To A Reply-to P

where P is a port that can be named by more than one Send Command. Ports offer flexibility in expressing different patterns of requests and replies, both between a single pair of processes and in cases where a process communicates with several other processes.

This approach permits parallelism and is more flexible, especially in connection with ports. However, the linguistic mechanisms needed to enable the programmer to do the matching introduce extra complexity; how much flexibility is gained and how much complexity is added requires further study. Note that this approach does not eliminate the need for supporting timeout, but now the timeout is specified at the point where the process must wait for the reply.

8.3 The In-Between Approach

The third approach is to make the sender wait for some indication about the progress in the delivery of the message instead of waiting for the reply from the target process. For example, in Hoare's language [HOAR77], the sender waits until the replier receives the message.

The first question to ask is: does this approach offer the programmer any advantages over the other two approaches? Since sends and replies are not explicitly paired, from this point of view in the in-between approach offers similar advantages and disadvantages as the no-wait approach described in Section 8.2. What is gained over the no-wait approach is that certain failures, for example (c) and (d), or possibly even (e) can be treated as exceptions of the send command. More importantly, the completion of the send command indicates that a meaningful message (to some extent) has been received. However, the completion of the send does not guarantee that the message will be processed (e.g. the receiver's node might fail immediately after the message is received). It should be noted that there is a substantial loss of parallelism over the no-wait approach.

The in-between approach is often advocated on implementation grounds, as a means to prevent flooding of the receiver. Flooding means that messages are delivered faster than the receiver can process them. Since the buffer space of the receiver is always limited, either some control must be provided to stop the flow of messages or some messages must be discarded. In a system with shared memory, a very efficient implementation is possible, namely, each process has one send buffer, and the message is held there until the receiver wants it. In a system without shared memory such as our distributed system, this approach is clearly impractical, since extra messages would be needed to inform the communicating parties that a message is ready (sender to receiver) and that it can be transferred (receiver to sender).

To conclude, the in-between approach does not seem to offer any real advantages. The waiting approach, while simpler, is more limiting from the point of concurrency and flexibility; thus we believe that we should provide the type of constructs outlined in Section 8.2.

9. SUMMARY AND FUTURE WORK

This report presented a basic conceptual model of distributed computation in an environment where the individual nodes of the underlying network are autonomous and may be spread over a large geographical area. Several important language and system issues were discussed, including reliability, protection, and the language constructs for sending and receiving messages in the abstract network. We tried to justify our decisions regarding each individual topic; of course, these decisions may be modified as we gain more experience with the use of distributed systems.

It should be realized that many of the issues that are being attacked in our project go beyond the area of distributed systems. The integration of a programming language and an operating system is one of these issues. The problem of such a "total environment" has not yet been solved satisfactorily even for a single processor system; in the case of distributed systems, the design task is even more difficult. However, while for a centralized system the total environment is a desirable feature, we believe that for a distributed system it is an essential property.

The work continues on several levels. To get a better understanding of what it means to do distributed computing, we must implement some distributed applications. This means that we need a language that will facilitate such a task. The CLU language [LISK77] is being extended to incorporate the guardian construct and corresponding message passing primitives. Extended CLU will serve as an experimental base for both the language work and the design of the reliable system support. Through this experiment, we hope to determine the effectiveness of different types of language constructs and their implementability on conventional hardware. This experience will be used to improve the language, and possibly to guide a design of a more hospitable machine architecture.

On the system level, research concentrates on high level protocols and mechanisms needed to achieve reliable operation. The feasibility of using some of the recently developed approaches to providing atomic operations [REED78; MONT78; TAKA78] in real systems is being investigated. Other issues include naming and copying objects in the abstract network [SOLL79], the guaranteed delivery support, and the use of backup copies.

Ultimately, the success of our programming system will depend on its usefulness in constructing distributed applications. In addition to a design of distributed implementation of several small scale sample problems, we envision our system to be used in implementation of a substantial distributed application; one of the primary candidates in an "office-automation" of our laboratory. This last project is dependent on two other projects within LCS: development of a local area network and development of advanced nodes for a distributed system. These two projects will provide the hardware base necessary for experimentation with distributed processing.

REFERENCES:

BRIN75 Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, (1975).

CHAM75 Chamberlin, D.D., et al., "Views, Authorization, and Locking in a Relational Data Base System," *Proc. AFIPS NCC*, 1975, pp. 425-430.

CLAR78 Clark, D.D., et al., "An Introduction to Local Area Networks," *Proc. Of IEEE*, Vol. 66, No. 11, November 1978, pp. 1497-1517.

DOLI77 D'Oliveira, C.R., "An Analysis of Computer Decentralization," M.I.T. Laboratory for Computer Science, Technical Memo No. 90, (October 1977).

DENI75 Dennis, J.B., "First Version of a Data Flow Procedure Language," M.I.T. Laboratory for Computer Science, Technical Memo No. 61, No. 11, (November 1976), pp. 624-633.

FELD77 Feldman, J.A., "A Programming Methodology for Distributed Computing," University of Rochester, Department of Computer Science, Technical Report No. 9, (1977).

GRAY78 Gray, J.N., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, *Lecture Notes on Computer Science*, Vol. 60, Springer-Verlag, 1978, pp. 393-481.

HEWI76 Hewitt, C., "Viewing Control STructures as Patterns of Passing Messages," M.I.T. Artificial Intelligence Laboratory, A.I. Memo 410, (December 1976). Accepted for publication in A.I. Journal.

HEWI77 Hewitt, C., et al., "Parallelism and Synchronizaiton in Actor Systems," *Record of 1977 Conference on Principles of Programming Languages*, Los Angeles, California, January 1977, pp. 267-280.

HOAR77 Hoare, C.A.R., "Communicating Sequential Processes," Oxford, University Computing Laboratory, Programming Research Group, (1977), DRAFT.

KENT76 Kent, S.T., "Encryption-Based Protection Protocols for Interactive User-Computer Communication, M.I.T., Laboratory for Computer Science, Technical Report No. 162, June 1976.

LAMP76 Lampson, B., Sturgis, H., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, (1976), (to appear in *Comm. of the ACM*).

LAUE78 Lauer, Needham, R., "On the Duality of Operating System Structures," *Proc. Second International Symposium on Operating Systems*, IRIA, October 1978.

LISK77 Liskov, B., et al., "Abstraction Mechanisms in CLU," Comm. of the ACM, Vol. 20, No. 8, (August 1977), pp. 564-576.

MILL77 Millstein, R.E., "The National Software Works: A Distributed Processing System," Proc. of ACM Conference, (October 1977), pp. 44-52.

MONT78 Montgomery, W., "Robust Concurrency Control for a Distributed Information System," M.I.T. Department of Electrical Engineering and Computer Science, PhD Thesis, December 1978.

NEED78 Needham, R.M., Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," Comm. of ACM, Vol. 21, No. 12, December 1978, pp. 993-999.

REED78 Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," M.I.T. Department of Electrical Engineering and Computer Science, PhD Thesis, September 1978.

ROBE70 Roberts, L.G., Wessler, B.D., "Computer Network Development to Achieve Resource Sharing," Proc. AFIPS SJCC, (1970).

ROTH77 Rothnie, J.B., et al., "The Redundant Update Methodology of SDD-1: A System For Distributed Databases," Computer Corporation of America, Report CCA-77-02, (February 1977).

SALT74 Saltzer, J.H., "Protection and the Control of Information Sharing in Multics," Comm. of the ACM, Vol. 17, No. 7, (July 1974), pp. 388-402.

SCHM77 Schmidt, J.W., "Type Concepts for Database Definition: An Investigation Based on Extensions to Pascal," Institut fur Informatik, Universitat Hamburg, June 1977.

SMIT77 Smith, J.M., Smith, D.C.P., "Database Abstractions: Aggregation," Comm. of the ACM, Vol. 20, No. 6, June 1977, pp. 405-413.

SOLL79 Sollins, K.R., "Copying in A Distributed System," M.I.T. Depratment of Electrical Engineering and Computer Science, MS Thesis, 1979 (in preparation).

STEA76 Stearns, R.E., et al., "Concurrency Control For Database Systems," Extended Abstract, IEEE Symposium on Foundations of Computer Science, October 1976, pp. 19-32.

SVOB78A Svobodova, L., "Distributed Computer System in a Bank: Notes on the First National City Bank," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments NO. 155, January 23, 1978.

SVOB78B Svobodova, L., "Distributed Computing in the Bank of America," M.I.T. Laboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 157, (February 1978).

TAKA78 Takagi, A., "Concurrent and Reliable Updates of Distributed Databases," M.I.T. LAboratory for Computer Science, Computer Systems Research Division, Request for Comments No. 167, Novemberr 22, 1978.

THOM76 Thomas, R.H., "A Solution to the Update Problem for Multiple Copy Data Bases which Use Distributed Control," Bolt Beranek & Newman, Inc., Report No. 3340, July 1976.

WIRT77 Wirth, N., "Modula: A Language for Modular Multiprogramming," Software Practice and Experience, Vol. 7, No. 1, January 1977.

WULF76 Wulf, W.A., et al., "An Introduction to the Construction and Verification of Alphard Programs, IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 253-265.