LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Modular Program Construction Using Abstractions

Computation Structures Group Memo 184
September 1979

Barbara Liskov

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Primitives for Distributed Computing[1]

Barbara Liskov
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract -- Distributed programs that run on nodes of a network are now technologically feasible, and are well-suited to the needs of organizations. However, our knowledge about how to construct such programs is limited. This paper discusses primitives that support the construction of distributed programs. Attention is focussed on primitives in two major areas: modularity and communication. The issues underlying the selection of the primitives are discussed, especially the issue of providing robust behavior, and various candidates are analyzed. The primitives will ultimately be provided as part of a programming language that will be used to experiment with construction of distributed programs.

1. Introduction

Formerly, economic considerations favored the sharing of a single computer among many users, and work in operating systems was concerned with the support and control of such sharing. Now, however, there is less of a need to share a single, expensive resource. Advances in hardware technology have led to greatly decreased costs for processors and memory. A possible consequence of the decreased cost is a new way of organizing software, where subparts of a program reside at and are executed at different computers connected by a network. We will refer to such a program as a *distributed program*.

A primary reason why distributed programs are desirable is that the organizations that use the programs are distributed. For example, a business is subdivided into many divisions. Each division has different responsibilities, and carries out internal procedures on internal data. However, the divisions provide services for one another, so some communication among them is needed. The divisions may be physically close, or they may be geographically dispersed.

An ideal program structure to support the above organization consists of an independent program unit for each division (to carry out the internal procedures on the internal data), together with a communication mechanism that permits data and requests for services to be exchanged among the units. Such a structure could be organized on a single centralized computer, but there are a number of advantages associated with a distributed organization, among them

1. Reduced contention. If all units run on a centralized facility they compete with one another for CPU and memory cycles. If each runs on its own computer, this contention can be eliminated. Contention for the communication medium is introduced instead, but in many organizations, communication is infrequent so this contention may not be a problem.

2. Speed of access. Each division can have faster response from its unit, partly because of reduced contention, but also because the unit can be located physically close to the division. This advantage is particularly important if the organization is geographically dispersed.

3. Physical control. Each division has physical control of its unit, and can determine who may use it and what it does [1].

Other important advantages include the potential for better reliability and higher availability, and natural extensibility to accommodate changing computation needs [2].

Thus there are important reasons why a distributed program is a desirable structure. However, experience in building such programs is so limited that we must experiment to learn the proper way of constructing them. This paper describes a set of primitives intended to support construction of distributed programs. The primitives are part of a programming language to be used in carrying out experiments in distributed program construction.

---

1. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-C-0661, and in part by the National Science Foundation under grant MCS 74-21892 A01.

1.1 Assumptions

In our selection of primitives, we are influenced by some assumptions about hardware, and about the way in which that hardware will be used. We assume that distributed programs run on a collection of computers, called *nodes*, that are connected by means of a communications network. Each node consists of one or more processors, and one or more levels of memory. The nodes are heterogeneous, e.g., they may contain different processors, come in many different sizes and provide different capabilities, and be connected to different external devices.

The nodes may communicate only via the network; there is no (other) shared memory. We make no assumptions about the network itself other than that it supports communication between any pair of nodes. For example, the network may be longhaul or shorthaul, or some combination with gateways in between; these details are invisible at the programmer level.

We assume that each node has an owner with considerable authority in determining what that node does (see Advantage 3 above). For example, the owner may control what programs can run on that node. Furthermore, if the node provides a service to programs running on other nodes, that service may be available only at certain times (e.g., when the node is not busy running internal programs) and only to certain users. We refer to such nodes as *autonomous*.

The principal consequence of the assumption of autonomy is that the programmer, not the system, must control where programs and data reside. The system may not breach the autonomy of a node by moving processing to it for purposes of load sharing. This attitude distinguishes our approach from multi-processor organizations such as CM* [3] and from high level approaches such as the Actor system [4], where the mapping of a program to physical locations is entirely under system control. Work in the same general area includes [5] and [6], although autonomy is not explicitly addressed.

1.2 Approach

Our approach is to extend an existing sequential language with primitives to support distributed programs. Our base language is CLU [7, 8]. Although the primitives are mostly independent of the base language, CLU is a good choice for two main reasons. It supports the construction of well-structured programs through its abstraction mechanisms, especially data abstractions; it is reasonable to assume that distributed programs will require such mechanisms to keep their complexity under control. Secondly, CLU is an object-oriented language, in which programs are thought of as operating on long-lived objects, such as data bases and files; this view is well-suited to the applications of interest, e.g., banking systems, airline reservation systems, office automation.

Although the research concerns linguistic primitives, the issues under discussion are systems issues. Indeed, the abstract machine on which the programs will run bears a strong resemblance to that provided by an operating system kernel. There are advantages, however, in orienting the work toward a programming language: a more regular structure to the abstract machine, enforced restrictions on program structure, compile time checking.

The remainder of this paper discusses primitives for distributed computing in two key areas, namely, modularity and communication. Linguistic constructs are proposed, emphasis is on the semantics of the constructs although some syntax is sketched. A major concern within both areas is robustness (providing reliable performance in spite of node and network failures). This concern is addressed within each area, and its interaction with proposed constructs is sketched.

Section 2 discusses modularity, and proposes a modular structure for distributed programs. Section 3 discusses communication; it examines possible message passing primitives, and the issues that arise in choosing among them. The example of an Airline Reservations System is used in both sections to illustrate the constructs. Finally, Section 4 contains a brief review and discussion.

2. Modularity

For distributed programs, a modular unit is needed that

1. Can be used to model the tasks and subtasks being performed in a reasonably natural way.

2. Can be realized efficiently, i.e., gives the programmer a realistic model of the underlying architecture.

A major issue in point (2) is control of direct sharing of data. Data that is shared directly (i.e., many entities know its location in the distributed address space) is a problem for three reasons. It can be a bottleneck because of the contention for its use. It is a storage management problem, since to deallocate data while avoiding dangling references requires detection and invalidation of all references to the deallocated data. Finally, to coordinate data sharing correctly can lead to increased program complexity. The main conclusion that can be drawn from considering these problems is that a linguistic mechanism that encourages the programmer to think about controlling the direct sharing of data is desirable. Note that a synchronization mechanism such as a monitor [9, 10] helps with the synchronization problem but not with the other two, since the monitor itself is a shared datum.

2.1 Guardians

We provide a construct called a *guardian* to support modular distributed programs. A guardian consists of objects and processes. A *process* is the execution of a sequential program. *Objects* contain data; objects are manipulated (accessed and possibly modified) by processes. Examples of objects are integers, arrays, queues, documents (in an office automation system), bank accounts and procedures. Objects are strongly typed: They may be directly manipulated only by operations of their type. The types may be either built-in or user-defined.

A computation consists of one or many guardians. Within each guardian, the actual work is performed by one or many processes. The processes within a single guardian may share objects, and communicate with one another via these shared objects.

Processes in different guardians can communicate only by sending messages (message passing will be discussed in Section 3). Messages will contain the *values* of objects, e.g., "2" or "#176538 $173.72" (the value of a bank account object). An important restriction ensures that the address space of a guardian remains local: it is impossible to place the *address* of an object in a message. It is possible to send a *token* for an object in a message; a token is an external name for the object, which can be returned to the guardian that owns the object to request some manipulation of the object. (A token is a sealed capability [11] that can be unsealed only by the creating guardian.) The system makes no guarantee that the object named by the token continues to exist; only the guardian can provide such a guarantee. Thus a guardian is entirely in charge of its address space, and storage management can be done locally for each guardian.

A guardian exists entirely at a single node of the underlying distributed system: its objects are all stored on the memory devices of this node and its processes run on the processors of the node. During the course of a computation, the population of guardians will vary; new guardians will be created, and existing guardians may self-destruct. The node at which a guardian is created is the node where it will exist for its lifetime. It must have been created by (a process in) a guardian at that node. Each node comes into existence with a *primal* guardian, which can (among other things) create guardians at its node in response to messages arriving from guardians at other nodes. This restriction on creation of new guardians helps preserve the autonomy of the physical nodes.

A guardian is an abstraction of a physical node of the underlying network: it supports one or more processes (abstract processors) sharing private memory, and communicates with other guardians (abstract nodes) only by sending messages. In thinking about a distributed program, a programmer can conceive of it as a set of abstract nodes. Intra-guardian activity is local and inexpensive (since it all takes place at a single physical node); inter-guardian processing is likely to be more costly, but the possibility of this added expense is evident in the program structure. The programmer can control the placement of data and programs (one of the requirements discussed in Section 1) by creating guardians at appropriate nodes. Furthermore, each guardian acts as an autonomous unit, guarding its resource and responding to requests as it sees fit.

2.2 Robustness

A major problem in distributed programs is how to achieve robust execution of atomic operations in spite of failures. (An atomic operation is either entirely completed or not done at all.) This is an area where distributed programs are likely to differ significantly from centralized programs. Not that the need for robustness is new; rather, the issue has been largely ignored in centralized systems, with the exception of some work in data base systems.

One requirement for robustness is *permanence of effect*. Permanence means that the effect caused by a completed atomic operation (e.g., a change in the state of the resource owned by the guardian that performs the operation) will not be lost due to node failures.

To achieve permanence requires a finer grain of backup and recovery than is provided by occasional system dumps and automatic system restart. We believe that permanence must be provided by each guardian for the resource it guards. We expect that backup and recovery will be provided on a per guardian basis: processes in the guardian save recovery data as needed (by, e.g., logging it in storage that will survive a node crash), and the guardian provides a recovery process that is started after a node crash to interpret the recovery data.

2.3 Discussion and Examples

The guardian construct was invented to satisfy the modularity criteria given above. The purpose of a guardian is to provide a service on a resource of a distributed program, but in a safe manner, i.e., it guards the resource by properly coordinating accesses to it, by protecting the resource from unauthorized access, and by providing backup and recovery for the resource in case of node failures. The resources being so guarded may be data, devices or computation.

For example, the flight data for an airline might be guarded by a single guardian that handles reservations for all flights and also provides a number of administrative functions such as deleting or archiving information about flights that have occurred, collecting statistics about flight usage, etc. It responds to requests such as "reserve," "cancel," "list passengers," and so on. For such requests, it checks that the requestor has the right to request the access (perhaps using some sort of access control list mechanism [12]). For example, only a manager can request a passenger list, or a reservation request from some other airline might not be permitted to reserve the last seat on a flight. The guardian guarantees that requests are properly coordinated, for example, performed in an order approximating the externally observable order in which they were requested. It performs the reserve and cancel requests as atomic operations, and logs them so that information will not be lost if the node fails.

Internally, the airline guardian might make use of a guardian for each flight. The top level guardian simply dispatches a request to the appropriate flight guardian, which does the actual work and logs results. A flight guardian might be organized in several different ways, for example:

1. A single process handles requests one at a time (Figure 1a).

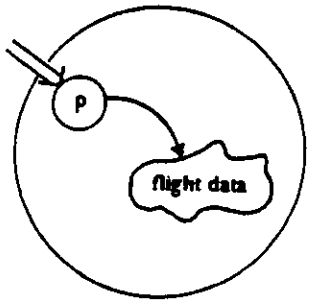
2. Requests for different dates are permitted to proceed in parallel. A single process synchronizes requests; it hands them off to other processes that perform the actual work (Figure 1b) when the flight data of interest are available. Such a structure is similar to that provided by a serializer [13].

3. A single process receives a request and immediately creates a process to handle it (Figure 1c). The forked processes synchronize with each other to ensure that only one process is manipulating the data for a particular date at a time. The processes synchronize using shared data, e.g., a monitor [9] providing operations *start_request(date)* and *end_request(date)*.

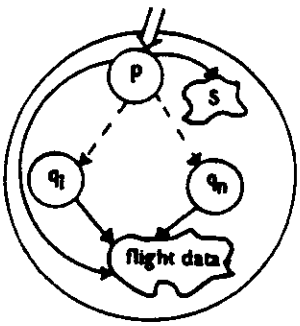
Organizations 2 and 3 can provide concurrent manipulation of the data base, while organization 1 cannot.

The airline data base discussed above had a single top

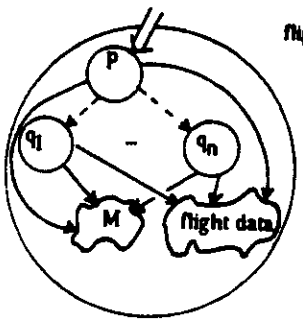
Figure 1. Possible organizations for a flight guardian.



a. One-at-a-time solution: process p handles requests sequentially.



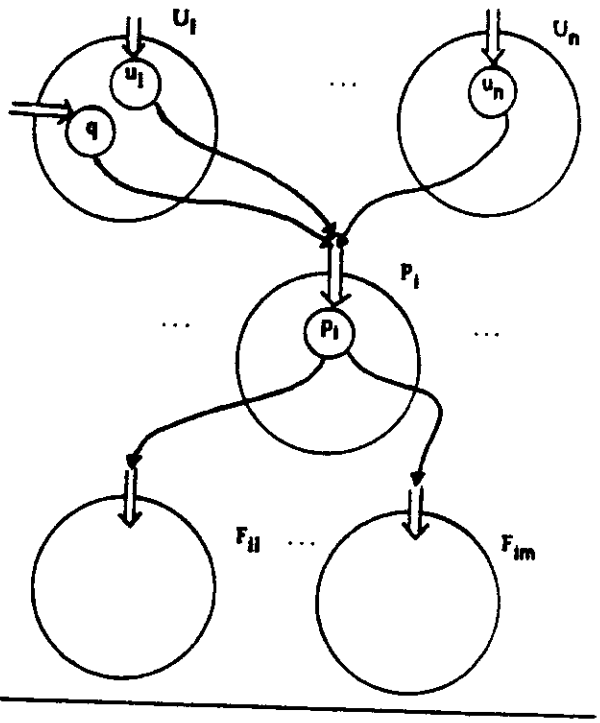
b. Serializer solution: process p uses synchronization data S to determine when requests should be performed. It forks processes q_i to do the actual requests.



c. Solution using a monitor: process p forks a process q_i upon receipt of a request. The processes q_i synchronize with each other using monitor M and perform the requests on the data base.

level guardian. Alternatively the data base might be distributed; for example, it might be divided into partitions for different geographical regions, each residing at a distinct node, and the guardian for a flight assigned to the region containing the flight's destination. Such a structure is shown in Figure 2. Here each node belonging to the airline has one guardian, P_i, for the region in which it resides, and one guardian, U_i, to provide an interface to the airline data base for that node's users (e.g., reservation clerks and administrators). A user makes a request to the U_j at his node; some checking for access rights would be done here, and then one or more requests sent to the appropriate P_i. The P_i would

Figure 2. Distributed airline system example.
There are n front ends (guardians U_1 to U_n) and n regional managers (only one, guardian P_i is shown) that communicate with the guardians of flights in its region (guardians F_{i1}, ..., F_{im}). Process q in U_j is carrying out a transaction for a user. Processes u_i are ready to accept requests from new users.



dispatch these requests to the flight guardians for its region. The P_i and U_j would coordinate as needed by means of some protocol established for that purpose. A possible organization for the U_j might be to fork a process to handle a transaction consisting of many requests; this process would carry out U_j's end of the coordination protocol. This process might, for example, interact with a clerk to make a number of reservations for the same customer.

In the organization shown in Figure 2, each guardian U_j guards the entire airline data base and provides transactions that consist of sequences of requests. Each guardian P_i guards the data for a geographical region, while each flight guardian guards the data for a single flight. Thus, access to the entire distributed data base is provided by a group of guardians, but each guardian in that group guards a discernable resource.

It is appealing to imagine a system structure in which processes do not share any data. Although multi-process guardians are not necessary for computational power, we permit many processes in a guardian for two main reasons: concurrency (e.g., Figures 1b and 1c) and conversational continuity. Concurrency could be obtained by having guardians that guard very small resources (e.g., the information about a single flight and date, or a single record in a data base), but we feel such a structure will often be unnatural. Conversational continuity is illustrated in Figure 2: process q carries on a conversation with the user and the "state" of this conversation (e.g., the identity of the passenger for whom

reservations are being made and the reservations made so far) is captured naturally in the state of process q.

3. Communication

Processes in different guardians can communicate with each other only by sending and receiving messages. This section discusses some issues in message communication. Our goal is to provide a general purpose communication mechanism that can be used in implementing application protocols. This primitive should make efficient use of the underlying hardware. It should also support communication in terms of abstract values meaningful in the application domain.

The following are communication primitives that might be considered.

1. The process sending a message waits only until the message has been composed. We will refer to this as the *no-wait send*.

2. The sending process waits until the message has been received by the target process. We will refer to this as the *synchronization send*. It is the send primitive described by Hoare [14]. The name is chosen because the primitive requires the sender and receiver to synchronize to exchange a message.

3. The sending process waits for a response from the receiving process that the command has been carried out. We will refer to this as the *remote invocation send* because of its similarity to invocation. Such a primitive has been described by Brinch Hansen [5].

In discussing the possible primitives, it is useful to have some examples of what users might like to accomplish. Various protocols have been discussed in the literature. For example, protocols have been described for distributed simultaneous updates [15, 16, 17], for recoverable atomic transactions [18, 19, 20], and for establishing secure communication links [21]. When these protocols are examined, we see that often messages are exchanged in pairs: one process sends a message to another to request some action, and later a response message flows in the opposite direction detailing the result. Such exchanges are like remote invocation: the first message is the invocation, while the second is the return. Thus, we might be led to believe that remote invocation is the appropriate choice.

However, not all message exchanges have this form. At least two other patterns can be identified. In the first, several messages are sent from one process to another, but only one response message is expected. In the second, the response comes from a different process than the original recipient of the request message. In both cases, there are request messages that have no corresponding response.

Many existing protocols are concerned with providing robust atomic operations, including permanence of effect (see Section 2.2) and also

1. Ensuring reliable communication between the requesting process and the process that performs the atomic operation.

2. Coping with node failures while the operation is being performed.

The above communication primitives differ primarily in the extent to which they address these two issues. The no-wait send can usually ensure message delivery. The synchronization send can guarantee delivery (if it terminates), but a subsequent node failure will disrupt communication. Presumably, the remote invocation masks both node and network failures. However, the variability in existing approaches to masking node failures is, in our opinion, an argument against selecting remote invocation at this time.

We believe that at present it is best to be conservative and select a primitive that can implement currently known protocols; in particular, it must provide for the patterns described above. Distributed computing is in its infancy; there is not yet, as there is for parallel programs, a set of examples that can be used to test the sufficiency of a proposed primitive. In such a situation, a flexible and general low-level primitive is a better choice than a higher-level one that may preclude desirable solutions.

Our choice is the no-wait send. It is the only primitive of the three that matches the above patterns, since either of the others would require additional messages to be exchanged. Furthermore, it can be used to implement the others, but not vice versa (if extra message passing is to be avoided). For receiving messages, we will provide a receive primitive with timeout. Timeout is necessary because an expected response may not arrive due to software errors or hardware failures.

The semantics of send and receive will be discussed in more detail below. First, however, we discuss other aspects of communication, namely, what messages are like, where they are sent, and how abstract values are communicated.

3.1 Messages

A communication involves the exchange of a *message* between two processes. Although a message could be viewed as an ordinary object that can be manipulated (in accordance with its type), our approach is to treat messages specially. A message is created as part of the execution of a **send** command, analogously to the creation of an activation record in the execution of an invocation.

The similarity between sending messages and invocation is emphasized by the way we structure messages: a message consists of a *command identifier*, and zero or more arguments. For messages sent to request a service, the command identifier corresponds to the name of an operation to be invoked. An example is the message sent to a regional guardian (P_1 in Figure 2) to reserve a seat:

reserve (flight_no, passenger_id, date)

Here *flight_no*, *passenger_id* and *date* are types. An instance of this message type would be created when a send command was executed, e.g.,

send reserve (f, p, d) ...

Such a command would be legal only if *f* were a *flight_no*, *p* were a *passenger_id* and *d* were a *date*. For messages sent to convey the result of a request, the command identifier explains the kind of result obtained. For example, responses to a request to list the passengers on a flight might include

info (passenger_list), no_such_flight

3.2 Ports

The next issue is where messages go. Messages could be sent, for example, to a guardian or to a process. The latter seems incompatible with our view of processes as anonymous providers of activity within a guardian. The former is more desirable, but seems a little too restricted. For example, in Figure 2, processes u_i and q could both get their messages from the same source, but since they handle different kinds of messages, some additional mechanism (e.g., pattern matching as in [14]) would be needed to keep the messages separated.

Our solution, instead, is to send messages to ports [22]. A port is a one-directional gateway into a guardian. There can be many ports on a single guardian; each port belongs to a guardian, and only processes within that guardian can receive messages from it.

Ports are the only entities that have global names. When a guardian is created, it provides one or more ports; the names of these ports are made known to the creating process. The names of ports can also be sent in messages; this implies that messages can be sent to the same port from many different sources. We assume that ports provide some buffer space so that messages may be queued if necessary.

Ports are described by describing messages that can be sent to them. For example, a port to one of the regional guardians P_i shown in Figure 2 might be described as follows.

```
regional_port = port [reserve (flight_no, passenger_id, date)
                        replies (ok, full, wait_list, pre_reserved,
                                 no_such_flight),
                      cancel (flight_no, passenger_id, date)
                        replies (cancelled, not_reserved, no_such_flight),
                      list_passengers (flight_no, date)
                        replies (info (passenger_list), no_such_flight)]
```

Note that here each request message is being paired with the expected response messages. The **replies** part is actually a description of an additional argument of the message: a port that can receive the expected responses. The syntax simply highlights the request-response relationship. To describe a message with no expected responses, the **replies** part is omitted.

The header of a guardian definition lists one or more ports that can be used to communicate with an instance of the guardian, e.g.,

```
regional_manager = guardiandef( ) provides p: regional_port
                   % definition of a sequential program to be run
                   % when an instance of regional_manager is
                   % created. p is a local variable to this
                   % program.
                   end regional_manager
```

The effect of executing

q: regional_port := **create** regional_manager ( )

is to create a new instance of regional_manager, and assign the name of the newly provided port to q. For example, the effect of executing the above **create** statement in process *x* in Figure 3a is shown in Figure 3b. Now *x* can send request messages to *y* via the new port. Recall that the new guardian will be created on the same node as its creator, so *A* must be resident on the appropriate regional node (e.g., *A* might be the primal guardian for that node).

Port types and guardian headers enable compile time type checking of all message passing. Compile time checking is possible even if guardian definitions are compiled separately, provided that compilation is done in the context of a library containing descriptions of guardian headers. (CLU already is based on such a library.)

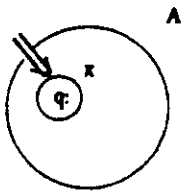
3.3 Sending Abstract Values

In the example, various abstract values, e.g., flight_no, date, passenger_list, are shown as being transmitted in messages. While these values may be of built-in type (e.g., flight_no may be an integer), they might also be of user-defined type. It is desirable that the two cases be treated uniformly as far as the send command itself is concerned.

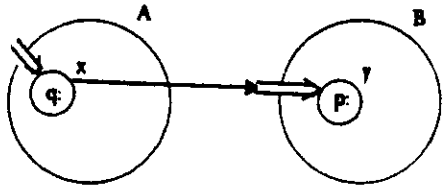
We start with the assumption that the system can build and decompose messages consisting of objects of built-in types. Furthermore, the system is responsible for the low-level protocols involved in actually transmitting a message, e.g., breaking a large message into packets and reassembling the packets, use of redundant information for error detection and correction. For example, the sending of a message

C(a, s)

Figure 3. Creating a guardian.



a. Before creating an instance of *regional_manager*.



b. After creating *B*, an instance of *regional_manager*; *y* is the newly created process in *B*, and *p* is used in *y* to name the newly created port, while *q* is used in *x* for this purpose.

where C is a command identifier, a is an array of integers and s is a string can be entirely handled by the system.

Since objects of abstract (user-defined) type are ultimately represented by built-in types, one possible approach is to have the system build and decompose messages containing abstract objects by transmitting their representations in the same way as above. Often, in fact, this is just what is wanted. However, there are a number of reasons why such a method is not always the proper one:

1. The system cannot automatically determine the boundary of an object. For example, consider an abstract object that is a graph composed of nodes. When a node is being sent in a message, should just the node be sent, or should the subgraph headed by the node be sent?

2. It is desirable to permit different representations of types on different nodes. The system can accomplish this for built-in types, but not for user-defined types.

3. An object may contain information that is guardian dependent, e.g., an index into a private table of the guardian. Such information should not be transmitted in a message since it would not be meaningful to any other guardian, but the system cannot distinguish this class of information from other information.

4. For some types it may be desirable to forbid sending the abstract values in messages.

For these reasons, the programmer must be permitted control over the transmission of abstract values. The approach we are taking is the following. Every transmissible abstract type (those whose abstract values may be transmitted in messages) has an associated *external rep*, which is the representation to be used in messages. Each implementation of a transmissible type must provide two operations, *encode* and *decode*. *Encode* performs a mapping from the internal representation of the implementation to the external rep, while *decode* maps the external rep into the internal representation. *Encode* and *decode* do not construct messages; they merely build and decompose in-computer objects suitable for sending in messages. The actual construction of the message from the external rep objects is done automatically by the system.

A simple example is complex numbers, where on one node the representation might be real/imaginary coordinates, while on another polar coordinates might be used; the external rep might be the real/imaginary coordinates. As a second example, consider an associative memory abstract type, which provides lookup of items in an associative memory on the basis of a key. Operations for this type include

add_item (am, item, key)	adds a key/item pair to associative memory
get_item (am, key) returns (item)	retrieves the item associated with a key

Suppose that on node A the representation makes use of a hash table, while on node B the representation uses a tree. A possible external rep might be a sequence of items with associated keys. Then *encode* on node A would build a sequence of key-item pairs from the hash table representation, and *decode* on node B would construct a tree representation from such a sequence.

Within a distributed system, the meaning of a type must be fixed and invariant over all the nodes, even though the flexibility exists to have different implementations at different nodes. The single external rep is part of this fixed meaning. For example, the bounds on legal integer values must be defined system-wide. If 24 bit integers were the system standard, then all nodes must support them, and the external rep would probably contain just 24 bits. However, a byte-oriented machine might use 3 bytes to represent an integer while a 16-bit word machine could use two words. In the latter case, only 24 bits of the 32 available could be used for the integer, and results of integer arithmetic must be checked to ensure they are within bounds. Otherwise it might be impossible to send an integer value in a message because it was too big.

3.4 Sending and Receiving Messages

In this section we sketch a syntax and semantics for send and receive primitives. Both syntax and semantics are tentative and incomplete, and not all issues are addressed.

The form of the send is (approximately):

send C(args) **to** <port> [**replyto** <port>]

The **replyto** part is optional and is used to convey where to send a response if one is required (provided the recipient doesn't already know where to send the response). As was mentioned earlier, the reply port is really an extra argument of the message, but it is singled out in the syntax to clarify the intent of the **send**. Note that the reply port could belong to a different guardian than the sending process.

A possible semantics of **send** is as follows:

1. The arguments are encoded from left to right using the appropriate encode operation for each arg_i (the one belonging to the type of arg_i). One possibility is that some encode invocation may raise an exception; in this case the **send** command terminates and raises that exception.

2. The message is actually constructed (made into a string of bits with appropriate format).

3. The message is sent (after being broken into packets if necessary). This step may be combined with step 2.

The process executing the **send** can continue execution as soon as it can be guaranteed that future actions of that process cannot affect the values transmitted (e.g., after step 2). The system will attempt to deliver the message to the receiving node intact and in good condition; the delivery is not guaranteed, but will happen with high probability.

When the message is entirely and correctly received at the receiving node (i.e., all packets have arrived, and the bits of the message are not in error, as is indicated by the error detection bits), it is forwarded to the target port. If there is no room for the message, or if the port or guardian doesn't exist, the message is thrown away. When a discarded message has a **replyto** port, a failure message is sent by the system to that port (e.g., failure ("target port doesn't exist")). No guarantee about arrival order is made, i.e., even two messages x and y sent by a single process to the same port are not guaranteed to arrive in the same order they were sent. If the order is important, processes must coordinate to achieve it.

The form of the receive statement is

```
receive on <port list>
  when CI (formal arglist) [replyto <formal portarg>] : SI
  ...
  when CN (formal arglist) [replyto <formal portarg>] : SN
  when Failure (s: string) : Sfailure
  when timeout <expr> : Stimeout
  end
```

The meaning of this statement is as follows: If messages have already arrived at ports in the port list, one of these messages is removed. (A way of giving ports priority will be provided.) The line containing the command identifier of this message is selected (such a line must exist; this can be checked at compile time). The objects in the message are decoded left to right (using the appropriate decode operation) and assigned to the formals in the formal arglist, and the reply port, if any, is assigned to the formal portarg. Then the associated statement is executed. If no messages are waiting, the receiving process waits for one to arrive, or times out, whichever happens first.

The message "*failure* (string)" is automatically and implicitly associated with each port type. Failure messages are mostly generated by the system and convey such information as transmission problems, or non-existence of the target port or guardian.

3.5 Example

We now present a sketch of the airline reservation system shown in Figure 2, to illustrate the use of send and receive and some of the reasoning involved in providing robust atomic operations. Figure 4 provides a sketch of the regional manager guardian. It simply looks up the guardian of the requested flight using a map, and forwards the request; the response will go directly from the flight guardian to the original requesting process, bypassing the regional manager. We assume that the flight guardian logs the results of reserve and cancel operations, as discussed in Section 2.2. (In Figures 4 and 5, the notation T$op is used to refer to operation "op" of type T.)

Figure 5 shows the process that handles a transaction with a clerk. Recall that the user interface guardians U_i create a new process to handle a transaction consisting of a set of reservations and cancellations for a single customer. This process accepts requests one at a time. It does each reserve request and reports the result to the clerk. Cancel requests are not done immediately, however, but are processed at the time the transaction finishes. To finish the transaction the clerk indicates "done". The process keeps a transaction history; if the clerk wishes the transaction can be partially or totally undone. Cancels are saved until the end of the transaction to permit the customer a late change of mind. An unwanted reservation can be undone by a cancel, but the reverse is not true since the seat may have been taken in the meantime.

Figure 4. The regional flight manager guardian.
The data abstraction map provides a mapping from its first to its second parameter; it is used here to find the guardian of the desired flight. The actual work is done by the flight guardian; note that it replies directly to the original request.

```
regional_port = port [reserve (flight_no, passenger_id, date)
                      replies (ok, _), _]
flight_port = port [reserve (passenger_id, date) replies (ok, _), _]

regional_manager = guardiandef ( ) provides p: regional_port
  fmap = map [flight_no, flight_port]
  flights: fmap

  while true do
    receive on p
      when reserve (fl: flight_no, pa: passenger_id, d: date)
        replyto q: port [ok, full, pre_reserved, waitlist, noflight]:
          fp: flight_port := fmap$get(flights, fl)
            except when no_entry: send no_flight to q
                                  continue % to next iteration
                   end %except
          send reserve (pa, d) to fp replyto q
      ...
    end %receive
  end %loop
end regional_manager
```

A failure of the regional node will cause the **timeout** line of the receive statement in Figure 5 to be selected; the expression e would cause a delay long enough to permit the request to complete under reasonable circumstances. If the time out occurs, *nothing* is known about the true state of affairs: the request may never be done, or it might already be done. (This uncertainty always occurs after a timeout.) In the example, the information is conveyed to the clerk. One possibility is that he will ask to retry. Although a retry may result in a reserve or cancel request being made more than once, no problems result since they are *idempotent* (many performances are equivalent to one [18]).

Now suppose the node that is running the transaction process fails. Since there are alternative methods of finishing a transaction (e.g., the clerk can make a phone call), when the node comes back up it is possible that the old transactions are obsolete, and should not be continued. We have chosen, therefore, to forget transactions rather than to try and finish them after a crash.

When the node crashes in the middle of a transaction, the clerk knows the result of each request except the one being worked on. However, this one can simply be redone (since it is idempotent). To finish the transaction, the clerk starts a new transaction, either waiting until the node comes up or using

Figure 5. The transaction process.
Procedure do_trans is forked each time a new transaction starts up. The data abstraction transhistory is used to keep track of the history of the transaction. Argument c is the port provided by the guardian that manages the display used by the reservations clerk.

```
transport = port [reserve (flight_no, string, date), ... ]
termport = port [ok, illegal, full, ... ]
directory = map [string, flight_port]
replyport = port [ok, full, pre_reserved, no_such_flight, wait_list]

dotrans = proc (p: transport, c: termport, dir: directory,
                pa: passenger_id)
  t: transhistory := transhistory$create( )
  while true do
   receive on p
    when reserve (fl: flight_no, dest: string, d: date):
     x: flight_port := dir$get (dest)
      except when no_entry: send illegal to c
                            continue % wait for next request
                end % except
     s: replyport := new port
     send reserve (fl, pa, d) to x replyto s
     receive on s
      when ok: transhistory$add (t, "reserve", fl, d)
            send ok to c
      ...
      when timeout e: send failure ("can't communicate") to c
      end % receive
    ...
    when done: % do all cancels
               return % this terminates the process
    end % receive
   end % loop
 end dotrans
```

alternative means. This new transaction begins with the request being worked on when the node failed.

4. Discussion

The purpose of this paper has been to discuss issues that arise in the design of primitives supporting distributed programs. Two main areas were identified, modularity and communication. To support modular program construction, a novel construct called a guardian was proposed. A guardian completely controls access to the resource it guards and provides permanence of effect for that resource. It is the abstract analog of a physical node: One or more processes inside a guardian may share data belonging to the guardian, but interguardian communication is only via message passing.

In the area of communication, we discussed how well various message passing primitives support existing protocols. A major concern was support for robust and reliable programs that can recover from node and network failures; the most important difference between the primitives was the extent to which they masked failures. We concluded that it was too early to choose a primitive that masked node failures, since there is not yet agreement on how to do this. We chose the no-wait send, since it provided the best support for experimentation. We then discussed various aspects of message passing, including sending of abstract values, and compile time type checking.

Of course, program correctness will be a major concern in distributed systems as it is for centralized systems. We believe that modularity is the main issue here, and that a program structure like guardians, where shared data is strictly controlled, is what is needed. As far as message communication is concerned, permanence of effect is crucial for using information about the result obtained by a message exchange as a basis for future actions.

In this paper, we have concentrated on goals and issues that influence selection of primitives. Not all needed primitives have been discussed; for example, a serious omission is the mechanism for doing recovery. Although we have tried to justify decisions, some are undoubtedly wrong. Furthermore, the paper discusses current positions on issues under study; these positions are likely to change as our understanding increases.

What is chiefly needed at present is more experience with distributed programs. Our plan is to gain experience by writing a number of distributed programs. CLU extended with the primitives will be implemented to run on a network of computers, and will serve as a basis for experiments. We expect our current work will make it easier to carry out the experiments and to evaluate the results.

Acknowledgements

The research being described has been done in collaboration with several others, chiefly Dave Clark and Liba Svobodova. The author gratefully acknowledges their efforts, and also the efforts of those who provided criticisms of an earlier version of this paper.

REFERENCES

[1] Saltzer, J. H. Research problems of decentralized systems with largely autonomous nodes. **Operating Systems Review 12**, 1 (January 1978), 43-52).

[2] Scherr, A. L. Distributed data processing. **IBM Systems Journal 17**, 4 (1978), 324-343.

[3] Fuller, S. H., et al. **A Collection of Papers on CM*: A Multi-microprocessor Computer System.** Department of Computer Science, Carnegie Mellon University, February 1977.

[4] Hewitt, C. Viewing control structures as patterns of passing messages. **Artificial Intelligence 8**, 1977, 323-364.

[5] Brinch Hansen, Per. Distributed processes: a concurrent programming concept. **Comm. of the ACM 21**, 11 (November 1978), 934-941.

[6] Feldman, J. A. **A Programming Methodology for Distributed Computing.** Technical Report 9, Department of Computer Science, University of Rochester, Rochester, N. Y., 1977.

[7] Liskov, B., Snyder, A., Atkinson, R., and Schaffert C. Abstraction mechanisms in CLU. **Comm. of the ACM 20**, 8 (August 1977), 564-576.

[8] Liskov, B., Moss, E., Schaffert, C., Scheifler, R., and Snyder, A. **CLU Reference Manual.** Computation Structures Group Memo 161, Laboratory for Computer Science, M.I.T., Cambridge, Mass., July 1978.

[9] Hoare, C. A. R. Monitors: an operating system structuring concept. **Comm. of the ACM 17**, 10 (October 1974), 549-557.

[10] Brinch Hansen, P. The programming language Concurrent Pascal. **IEEE Trans. on Software Engineering 1**, 2 (June 1975), 199-207.

[11] Redell, D. D. **Naming and Protection in Extendible Operating Systems.** Technical Report LCS/TR-140, Laboratory for Computer Science, M.I.T., Cambridge, Mass., November 1974.

[12] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. **Proc. of the IEEE 63**, 9 (September 1975), 1278-1308.

[13] Hewitt, C., and Atkinson, R. Specification and proof techniques for serializers. **IEEE Trans. on Software Engineering SE-5**, 1 (January 1979), 10-23.

[14] Hoare, C. A. R. Communicating sequential processes. **Comm. of the ACM 21**, 8 (August 1978), 666-677.

[15] Thomas, R. H. **A Solution to the Update Problem for Multiple Copy Data Bases Which Uses Distributed Control.** BBN Report 3340, Bolt Beranek and Newman, Inc., Cambridge, Mass., July 1976.

[16] Alsberg, P. A., and Day, J. D. A principle for resilient sharing of distributed resources. **Proc. of the Second International Conference on Software Engineering**, 1976, 562-570.

[17] Rothnie, J. B., Bernstein, P. A., Goodman, N., and Papadimitriou, C. A. **The Redundant Update Methodology of SDD-1: A System for Distributed Databases.** Technical Report. Computer Corporation of America, Cambridge, Mass., February 1977.

[18] Lampson, B., and Sturgis, H. **Crash Recovery in a Distributed Data Storage System.** Xerox Research Center, Palo Alto, Ca., 1976.

[19] Gray, J. N. Notes on data base operating systems. **Operating Systems: An Advanced Course, Lecture Notes in Computer Science 60**, Springer-Verlag, 1978, 393-481.

[20] Reed, D. P. **Naming and Synchronization in a Decentralized Computer System.** Technical Report TR-205, Laboratory for Computer Science, M.I.T., Cambridge, Ma., October 1978.

[21] Needham, R. M., and Schroeder, M. D. Using encryption for authentication in large networks of computers. **Comm. of the ACM 21**, 12 (December 1978), 993-999.

[22] Balzer, R. M. PORTS – a method for dynamic interprogram communication and job control. **Proc. of the AFIPS Conference 39** (1971).