LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

On Linguistic Support for Distributed Programs

CSG Memo 201-1
October 1980

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 N00014-75-C-0661, and in part by the National Science Foundation under grant MCS79-23769.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

ON LINGUISTIC SUPPORT FOR DISTRIBUTED PROGRAMS

Barbara Liskov
MIT Laboratory for Computer Science
Cambridge, Ma. 02139

Abstract

Technological advances have made it possible to construct systems from collections of computers connected by a network. At present, however, there is little support for the construction and execution of software to run on such a system. This paper presents an overview of a research project whose goal is to provide the needed support. The major focus of our current work and this paper is support for the construction of robust software that survives node, network and media failures.

1. Introduction

Technological advances have made it possible to construct systems from collections of computers that communicate over a network. These advances call for the organization of software as *distributed programs*, whose modules reside and execute at several communicating yet geographically distinct locations. The goal of our research is to provide an integrated programming language and system to support the construction and execution of distributed programs. This paper describes the current status of our work.

Our approach is to extend an existing sequential language with primitives that support distributed programs. Our base language is CLU [1, 2]. CLU has been chosen for two reasons: it supports the construction of well-structured programs through its abstraction mechanisms, especially data abstractions, and it is an object-oriented language, in which programs are thought of as operating on long-lived objects, such as data bases and files -- a view well-suited to the applications of interest.

We have concentrated on applications that are concerned with manipulation of on-line data, e.g., airline reservation systems and banking systems, because we believe such applications are well-suited to a distributed implementation. Our intent is to design primitives that are as high level as possible yet are application-independent (within the chosen domain).

In the applications of interest, a major concern is to provide robust behavior in the face of failures of nodes, network and storage media. Accordingly, we have been studying linguistic support

for robust software. This is a difficult problem and one that has been largely ignored in linguistic work, including current proposals for distributed programming constructs, e.g., [3, 4, 5, 6]. One exception is the work at Newcastle upon Tyne [7, 8], but that approach is concerned with recovery from user errors rather than failures. An *error* occurs when a software module does not meet its specification, while a *failure* is an expected, if not very desirable, outcome. Clearly it is possible to plan in advance how to deal with expected failure outcomes, but much more difficult (if not impossible) to cope with errors. We do not address the problem of error recovery.

In addition to our concern with reliability, we have a number of goals that are derived from the factors that make a distributed organization attractive in the applications of interest. Our approach must support extensibility in a natural way, so that, for example, the addition of new computers to the network can be handled gracefully. The programmer must be free to distribute processing and data in a way that satisfies availability and efficiency requirements. Also, the programmer must be able to protect sensitive data from misuse.

The remainder of this paper discusses various aspects of our work. In Section 2 we discuss some underlying assumptions, and then briefly describe guardians, the basic modular unit that we provide for constructing distributed programs. Section 3 discusses some issues in message communication.

Section 4 is the main part of the paper. It discusses problems that arise in trying to write robust programs, and cutlines the approach we are taking to solve these problems. We identify the remote procedure call as a useful communication primitive, and define its semantic properties with respect to reliability. We also define the notions of permanent data, atomic actions, and atomic objects.

Finally, Section 5 discusses some of the design issues that we are working on at present.

2. Basic Model

In this section we begin by discussing some assumptions that underlie our approach. We then go on to describe the modular unit we provide for organizing distributed programs.

2.1 Assumptions

The distributed programs of interest run on *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 different sizes and provide different capabilities, and be connected to different external devices.

The nodes communicate with each other only through the network; there is no (other) shared memory. We make no assumptions about the network. For example, it may be longhaul like the ARPANET [10] or shorthaul [11], or some combination with gateways in between; these distinctions are invisible at the programmer level.

We assume that each node has an owner (a person or an organization) with the authority to determine what that node does. For example, the owner may control what programs are allowed to 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 assumption of autonomy supports some of the goals mentioned in the Introduction. A consequence of this assumption is that the programmer, and not the system, must control where programs and data reside. Therefore, the language cannot hide completely the location of programs and data from the programmer. Furthermore, the system may not breach the autonomy of a node by moving processing to it for its own purposes. Our support for autonomy distinguishes our approach from multi-processor organizations such as CM* [12], and from high level appproaches such as the Actor system [13] where the mapping of a program to physical locations is entirely under system control.

2.2 Modularity

To support distributed programs, a modular unit is needed that can model the tasks and subtasks being performed in a natural way, and that can be realized efficiently. Toward that end we provide a construct called a *guardian*.

The purpose of a guardian is to provide controlled access to a resource or set of resources, e.g., by synchronizing concurrent accesses, and by checking access requests to determine if they should be allowed. A guardian contains *processes* and *data objects*. A process is the execution of a sequential program. The processes do the actual work of the guardian. They manipulate the data objects, and can communicate with one another via shared objects.

Many guardians may cooperate to provide a *subsystem*: an application or system service such as a distributed data base or a message system (see [9] for an example). However, processes in different guardians can communicate only by sending *messages*. Messages 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 that object. (A token is a sealed capability [14] 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 the system can perform storage management locally for each guardian.

Although a subsystem may make use of guardians at many nodes, each individual guardian exists entirely at a single node of the underlying distributed system: its objects are stored on the memory devices of this node and its processes run on the processors of the node. Since the sizes of the physical nodes may vary, different nodes will support different numbers of guardians. During the course of a computation, the population of guardians will vary; new guardians will be created, and existing guardians may go away. A guardian may be created at a node only by (a process in) a guardian at that node. Each node comes into existence with a *primal* guardian, which can, if the owner of the node wishes, 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. Guardians may move from one node to another, but in a similarly restricted way.

One way of thinking about a guardian is as an abstraction of a physical node: a guardian supports one or more processes (abstract processors) sharing private memory, and communicates with other guardians (abstract nodes) only by sending messages. A programmer can conceive of a distributed program as a set of abstract nodes, each of which performs a meaningful task for its application. 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 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.

3. Communication

We have found it useful to distinguish two issues within communication. The first concerns the form of messages, and addresses such problems as type checking of message communication and the kinds of values that can be sent in messages. The second concerns important semantic properties of communication, including such questions as order of arrival of messages, and, most importantly, the reliability of communication. In this section we discuss the first issue; discussion of

the second issue is deferred to Section 4.2.

We believe that message communication should share certain desirable properties with the conventional procedure call, namely the ability to do compile-time type checking of message communication, and the ability for programs to communicate in application-oriented terms (i.e., in terms of the abstract objects of interest in the application program). The properties that the communication primitive should provide are as follows:

1. User programs need not deal with the underlying form of messages. For example, users should not need to translate data into bit strings suitable for transmission, or to break up messages into packets.

2. All messages received by user programs are intact and in good condition. For example, if messages are broken into packets, then the system only delivers a message if all packets arrive at the receiving node, and are properly reassembled. Furthermore, if the bits in a message have been scrambled, the message either is not delivered, or is reconstructed before delivery; clearly some redundant information is required here.

3. Messages received by a module are the kind that module expects. Support for this property requires type checking, which may be performed either at compile time or run time. Performing such type checking is analogous to type checking procedure calls.

4. Modules are not restricted to communicating only in terms of a pre-defined set of types, e.g., the built-in ones. Instead, modules can communicate in terms of values of interest to the application. In particular, if the application is defined using abstract data types, then values of these types can be communicated in messages.

The above four properties are needed to give message communication the same status as procedure call. The properties are supported to some extent in all languages that provide linguistic primitives for message communication [3, 4, 5, 6]. For example, ADA [5] requires all four properties.

Type checking at compile time requires a declarative mechanism similar to a header of a procedure definition. The information in such declarations can be used by the compiler to check that the form of a message satisfies the type constraints imposed by the intended recipient. For example, suppose that messages are sent to ports. Then each port must have a declared type that completely determines the set of messages it can receive and the responses to those messages. Furthermore, a guardian definition must contain declarations of the port types that can be used to communicate with that guardian. This information is sufficient to permit compile-time type checking of 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 guardians. CLU already is based on such a library [2].

For example, a message might be sent by executing

```
send C(a1, ..., an) to p
```

Here *p* is a port attached to some guardian *G*, *C* is the name of a request or operation that *G* is being asked to perform, and *a1*, ..., *an* are the actual arguments of operation *C*. The effect of executing such a **send** statement is that a message containing the name *C* and the values of *a1*, ..., *an* is constructed and sent to *p*. To acquire an incoming message, *G* might execute a receive statement:

```
receive on p
  when C (f1: T1, ..., fn: Tn): S
  ...
  end
```

When a message containing the request named *C* is received, the associated statement S is executed, with the formals *f1*, ..., *fn* initialized to contain the values extracted from the message. Properties (2) and (3) ensure that requests named *C*, with arguments of types *Ti*, can be sent to *p* and received from *p*.

All arguments of the **send** are transmitted by value.[1] Therefore, when one of these arguments is of abstract type, it will be necessary to somehow "copy" the value of that argument to the receiving module. We have studied the problem of how such a copy can be made under the assumption that the sending and receiving guardians may use different implementations of the abstract type. A description of the method can be found in [15, 16].

4. Reliability

In the applications of interest, important information is entrusted to the system. It is crucial that such information not be lost if various failures occur. In this section, we explore the issues that arise in trying to write programs that are robust -- survive node, network and storage media failures without loss of essential information. We also outline our approach to support for robust software.

Our assumptions about node, network and media failures are the same as those discussed in

---

1. Some kind of call-by-reference for **send** arguments, in which a reference to the argument is sent to the receiving module, is not permitted in our model, since guardians do not share address spaces. We believe call-by-reference is not very useful in a distributed program in any case.

[17]. Roughly, we divide failure into two categories, expected (but undesirable) events, and unexpected events, and we assume that the probability of an unexpected event occurring can be made as small as desired. Some combination of system and user code will mask the occurrence of expected failures; unexpected failures will be ignored. For example, it is expected that recognizably corrupted packets will sometimes be delivered by the network; it is unexpected that packets that look good are actually corrupted.

4.1. Permanence of Effect

In a distributed program, modules at different nodes may interact to achieve some common end. In our model, since guardians have no common memory, this interaction is achieved by message passing. One guardian sends a message to another requesting it to perform some action. If, subsequently, the requesting guardian is notified (by some other message) of the effect of the action, it must be able to rely on this information. Partly, the concern is one of program correctness: both requester and server must agree about the meaning of the message exchange. However, there is an additional issue here concerning reliability: it is important that the reported effect not be undone by subsequent (node, media, or possibly network) failures. We refer to the desired property as *permanence of effect*. Permanence is needed in both centralized and distributed programs, but the need seems to be particularly acute in distributed programs, because different parts of such programs can fail independently.

Our approach to permanence of effect is to provide guardians with a means of ensuring that their data survives crashes. Each guardian definition can declare a set of permanent variables. The guardian's permanent state consists of these variables, and all the data reachable[2] from them. The permanent state is stored in primary memory, and the data in it may be modified in the ordinary way. However, a backup copy of this information resides in non-volatile memory,[3] and the guardian has the ability to control when this copy is changed. For purposes of discussion, we will assume the existence of a **save** primitive. To change the copy, the guardian executes the **save** primitive. When this primitive is called, it stores an image of the permanent state in non-volatile storage. The store is done *atomically*: either the entire permanent state is saved, or the effect is as if the save had not been started. Thus a crash in the middle of a save does not leave the backup copy of the guardian's

---

2. The notion of reachability arises because there may be pointers.
3. This non-volatile memory could be located at another physical node.

permanent state inconsistent.[4]

Note that we are not talking here about a virtual memory scheme, where the system is moving pages between the secondary and primary memory. Such a scheme cannot ensure that the copy on secondary memory is in a consistent state at the time of a crash. Instead, a primitive like **save** is essential.

The system guarantees that guardians themselves are permanent. This means that after a node crash, the system will cause all guardians running at that node before the crash to continue their existence. Each guardian will restart with its permanent state having the value it possessed at the last completed save before the crash.

To use permanent storage properly, a guardian definition must have two code sections. The initialization section runs whenever a new guardian is created; its purpose is to initialize the permanent state to some consistent, initial value. Only after initialization is finished is the guardian in a fit condition for surviving crashes; at this point, the guardian becomes permanent.

The second section runs when initialization is complete, and also whenever the guardian is restarted after a crash. Its function is first to initialize whatever volatile state the guardian uses (for example, the guardian may keep an inverted index to a data base for fast retrieval), and then to do its actual work, e.g., continue what it was doing before the crash, and respond to incoming request messages.

An example of a simple guardian definition is given in Figure 1 to illustrate these concepts. This guardian provides an unbounded buffer of items. The buffer is stored in an array, which constitutes the permanent state. The volatile state is simply a count of the current number of elements; this information is redundant with information in the array.

The syntax used here is not intended to be real, but is introduced just for the example. The keywords **init** and **start** mark the two code sections discussed above. The communication primitive in use pairs requests and responses; when the **reply** statement is executed, the system sends the reply message to the process that made the request.

Although very simple, this example illustrates a common property of guardians, namely that in response to requests guardians perform mappings from a consistent permanent state to a new, consistent, permanent state. The example also illustrates that the mechanism used in the simple way shown is not powerful enough to prevent obvious problems. For example, a node crash between the

---

4. The save primitive stores permanent state in atomic stable storage as defined in [17].

Fig. 1. A guardian that provides a buffer.

```
buffer = guardian

    buf = array [item]
    pport = port [put (item) replies (ok)]
    gport = port [get ( ) replies (ok(item))]

  permanent
    b: buf
    p: pport
    g: gport

  init ( ) replies ok (pport, gport)
    b := buf$new ( )              % create a new empty buffer
    p := pport$create ( )         % create ports for receiving put
    g := gport$create ( )         % and get messages
    save
    reply ok (p, g)               % make p and g available
    end                           % to creating process

  start
    count: int := buf$size (b)
    while true do
      if count = 0 then
        receive on p
          when put (i: item): % add item to end of buffer
                              buf$addh (b, i)
                              save
                              count := count + 1
                              reply ok
        end
      end
      receive on p, g
        when put (i: item): buf$addh (b, i)
                            save
                            count := count + 1
                            reply ok
        when get ( ):       % remove and return 1st item from buf
                            i: item := buf$reml (b)
                            save
                            count := count - 1
                            reply ok (i)
      end
    end

end buffer
```

time the code handling *get* performs **save** and the time it replies will cause the item to be lost. Such problems will be discussed in the next section.

It is worth noting that the permanence mechanism is tied to guardians and not to processes. The system creates a single process inside the guardian to run the **init** or **start** code; however, this process can fork others, so that many processes can be running concurrently inside a guardian. (These processes must synchronize with each other as needed.) Nevertheless, when one of these processes executes **save**, what is saved is the permanent guardian state, and not the state of the process. Saving just guardian state is a convenient way of giving the programmer close control over the amount of permanent state. One possible negative result of this decision is that sometimes information about processes must be encoded in the permanent state, so the processes can be restarted (by the **start** code) after a crash. In the examples we have studied so far, it has been natural and easy to save information about processes; the programmer need not save the equivalent of a process checkpoint, but instead records tasks that the guardian is working on.

A few remarks are in order about the "non-volatile" storage used to store permanent data. We believe that the reliability of this storage may vary from node to node. The information could be stored in such a way that it survives node failures but not media failures, or failures of the node and a single media device, or failures of two media devices, etc. Whatever the storage method, reliability will never be 100%. To obtain reliability high enough for the needs of an application, nodes with the desired reliability properties could be purchased. Alternatively, critical data could be duplicated by storing it as the permanent state of guardians at other nodes. For example, in a data base system, a duplicate copy of the log might be kept at another guardian.

4.2. Inter-Guardian Consistency

As was mentioned above, the permanence mechanism does not solve all problems. The basic difficulty is that the mechanism allows a single guardian to make transitions from one permanent, consistent state to another, while what is needed is a mechanism that allows groups of guardians to make such transitions. For example, the user of a buffer guardian requests an item, and the buffer guardian provides one. The user then uses this item, and finally makes a change in its permanent storage to record the result. Only at this point should the item be truly removed from buffer guardian's permanent storage, because only at this point are the two guardians in a mutually consistent state: the item has truly been consumed.

We have been studying the problem of support for inter-guardian consistency. There are a number of difficulties that arise, some of which are discussed below. In the following *B* is a buffer

guardian and *U* is its user.

1. Suppose *B*'s node crashes before *U* gets a reply. Does the system hide this from *U*, or must *U* send the request again (after a suitable time has elapsed)?

2. If either *U* or the system sends the request again, there is a possibility that *B* already acted on the previous request. Must *B* recognize duplicate requests, or does the system hide this by never presenting *B* with a duplicate?

3. If *B* crashes after a **save** but before a reply to the *get* message, what prevents the item from being lost? Must *B* prevent this, or does the system prevent it?

4. If *U*'s node crashes after the request is sent, what ensures that after the crash, *U* picks up from where it left off in interacting with *B*, assuming *U* is still interested in the request?

5. In fact, *U* may not be interested in the request if the answer takes too long in coming, either because of a crash (at either node), or just because the person *U* was working for got tired of waiting. In this case, how is work performed for *U* by *B* undone, i.e., the effect of the request(s) removed from *B*'s permanent storage?

Note that in all these cases, the questions concern whether the system or the programmer handles the problems. Furthermore, all the questions are phrased in terms of communication between *U* and *B*. So what is at issue here is the exact semantics of message communication, and the relationship of message communication to permanent storage.

We return now to the issue, raised in Section 3, of the reliability of message communication. We believe one viable approach to this issue is to provide a communication primitive with the properties discussed in Section 3, but with no additional reliability properties over what the underlying network provides. The primitive might pair requests with responses, or it might not. In either case, the primitive would *not* hide the unreliability of the network or the nodes. It would not hide the fact that messages may be duplicated or arrive out of order. Most importantly, it would not guarantee message delivery.

Such a level is reasonable because it is inexpensive. However, it provides no help to the programmer of either *U* or *B* in solving the problems discussed above.

We have studied how various applications might cope with the above problems while using such a primitive in conjunction with the permanence mechanism of Section 4.1. For example, *U* might periodically re-send the request message, while *B* would check for duplicates. Sometimes duplicates

are not a problem; this happens for requests that are naturally *idempotent* (many executions are equivalent to one) [17]. For non-idempotent requests, such as *put* and *get* for the buffer, the request must have as an extra argument a unique identifier that can be used to recognize duplicates. *B* can use the unique identifier to remember its previous response to the request, and send this response again if the request comes in again. The information about requests and responses must be stored in permanent storage; however, saving it before the reply, and then saving the change to the array after the reply, solves problem (3). To avoid having to remember old requests forever, *U* and *B* may have to resort to a protocol that allows *U* to inform *B* that old requests can be forgotten.

So far, it appears that the programmer only encounters awkwardness, but no real difficulty, in solving the above problems (although we might be a little concerned that there are too many **saves** going on). However, this appearance is deceptive, because we have not yet solved problems (4) and (5), and they are the difficult ones. Furthermore, we have been looking at a limited kind of interaction, involving just two guardians, a *client* requesting service, and a *server* providing service. In general, we must expect nesting to occur, e.g., the client may actually be performing a service for some higher level client. *U* might be such an intermediate server/client.

Consider the case where *U*'s node crashes after *B* has performed a request but before *U* has recorded the reply in its permanent storage. When, after the crash, *U* re-receives the message that caused it to send the request to *B*, it must be able to re-send the request to *B* with sufficient information so that *B* can identify that request as a duplicate. Another possibility is that *U*'s client may not re-send the request to *U* or may ask *U* to abandon the request (problem 5 above). In this case, *U* (or some other guardian) must send a message to *B* requesting it to undo the previous work.

The analysis above is actually oversimplified, because we have not considered synchronization requirements. For example, often a read operation is not really idempotent, since usually it is important whether the read is done before or after a write operation on the same data. Also, if *U* adds an item to *B*'s buffer, this item should not be available to any other of *B*'s clients until *U* (or *U*'s client) is really finished.

With the addition of some sort of synchronization method, programmers can solve all the above problems. However, to do so requires substantial work, including both bookkeeping and an agreement about protocols between all cooperating guardians. Roughly, the guardians must carry out a two-phase commit protocol, complete with intentions lists [17] or undo/redo logs [18]. To perform such a protocol correctly requires careful analysis, and shortcuts usually result in errors, while to perform it efficiently is a difficult systems problem requiring substantial ingenuity. Therefore, it seems appropriate to attempt to provide primitives that make this work part of the language

implementation, and hide it from the programmer.

We intend to support a *remote procedure call* (RPC) primitive that, together with some other primitives to be described in the next section, provides the desired semantics. RPC pairs requests and replies, and re-sends messages as needed. More importantly, however, RPC provides *at-most-once* semantics: If the caller receives a reply, the system guarantees that the call was acted on exactly once, without any programmer having to worry about providing this. For example, a programmer need not worry about recognizing duplicates, because the system won't deliver any. Furthermore, if the call was not completed, either because the caller lost interest, or because it was not possible to do the requested work, the RPC is automatically undone, and guaranteed to have no effect. The RPC will also have synchronization properties, as described in the next section.

It is worth noting that at-most-once semantics are just what we would like for an ordinary procedure call in the presence of node crashes, permanent data, and impatient users. Furthermore, all the problems discussed above arise in a centralized system. In the past, however, the semantics of procedure call have not taken these factors into account.

RPC is a very high level primitive, certainly much higher level than the primitive discussed above, which did little to hide the unreliability of the underlying network. For some time we were hopeful that there might be an intermediate level primitive that would solve many of the user's problems, and would not be as expensive as RPC. However, in the course of our research we have come to believe that there is no such primitive.

As a simple example, consider a primitive that doesn't guarantee message delivery, but does guarantee that messages arrive in order, and that duplicate messages created by the network are not delivered to the user. Such a primitive is easy and fairly cheap to implement. However, it is not very helpful to the user, since the user must be concerned with detecting duplicates at a higher level. There will be duplicates generated by the user in attempting to guarantee delivery. For example, *U* was generating such duplicates in the scenario above. And, there will be duplicates created by the user after a crash, when a previous request is re-tried. Since the user must worry about duplicates anyway, the logic of his program is not simplified by the work the system is doing. It is true that the order preserving property could be used to control how long old requests must be remembered; however, the unique identifiers mentioned above must still be put in the messages and can also be used for this purpose. So although in this case it was not expensive to implement the communication primitive, the work being done is wasted effort.

In fact, intermediate level primitives often seem to be both of little help to the user, and expensive to implement. Therefore, we believe that only the two extremes are reasonable choices.

4.3. Atomic Actions

The previous section argued that in addition to the permanence mechanism, RPC with at most once semantics was needed to help programmers provide inter-guardian consistency. The approach we are investigating at present assumes as a basis a guardian structure as proposed in Section 4.1. Thus, a guardian still has a permanent state, as well as a volatile state. However, the save primitive has been subsumed into the RPC.

The at-most-once semantics that we want from our RPC is similar to what is required of an atomic transaction [18] on a data base. An atomic transaction either completes entirely or has no effect. There are really two separate but mutually dependent atomicity properties: *failure atomicity* is the all or nothing property mentioned above, while *indivisibility* ensures that intermediate states of a transaction are not visible to other transactions, including those running concurrently. Indivisibility requires a synchronization mechanism.

Our method of supporting RPC is based on atomic transactions. We will provide both atomicity properties. In addition, we will allow for nested transactions, since this is needed for generality.

In response to incoming RPC's, guardians perform *actions*. Actions are like ordinary procedures, except that they terminate by committing or aborting. If an action *commits*, changes it made to the guardian's state continue in effect, but if it *aborts*, it should have no effect on its guardian's state. In this case, any changes it made will be undone.

To define actions more precisely, we must define exactly which changes made by an action need to be undone if the action aborts. Since the state that really matters in a guardian is the permanent state, we could say this constitutes the affected state. However, we reject this approach for two reasons: first, volatile state is useful too, and second, we would like to define the effect of an action more narrowly, i.e., we would like to be able to define actions that only affect a subset of the total guardian state. A narrower definition will let us write more efficient guardians, since actions that affect disjoint subsets of the state could run in parallel. For example, actions on different records of a data base could proceed concurrently.

Therefore, we introduce a new concept, the *atomic object*. Atomic objects have both recovery and synchronization properties; we will discuss the synchronization properties a little later in this section. Actions operate on both atomic and non-atomic objects, but the affected state for an action consists only of the atomic objects it uses. As was mentioned above, if an action aborts, it should have no effect. The system provides for this by keeping a backup version for each atomic object modified by the action; then, if the action aborts, the system undoes its effect by restoring the

backup version.

An action is performed by executing an *action call*. Both local actions (in the calling guardian) and remote actions (in some other guardian) may be called. In either case, the system automatically provides at-most-once semantics. Thus, remote action calls are the remote procedure calls mentioned above.

Action calls come in two varieties: *top level calls*, and *nested calls*. A nested call is an action being performed on behalf of an action at a higher level. When such an action commits, there is still the possibility that some action higher in the call chain may abort. If this happens, the result of the nested call is not wanted, and its effect is undone automatically by the system. Therefore, when a nested call commits, the system still retains the backup versions of all affected atomic objects.

A top level call typically corresponds to doing something for a customer, e.g., withdrawing money from a bank account, so it seems appropriate to make a real change to the system state at such a time. Thus, when the top level call commits, the effects of that action really happen. This includes the effects of all nested calls performed on behalf of that top level action. At this point, there is no longer any possibility of abort, so backup versions of modified atomic objects can be discarded.

The synchronization properties of an atomic object ensure that changes made to it by one action will not be visible to other actions running concurrently. For greater concurrency, an atomic object can be read concurrently by many actions, but an action that writes the object will lock out all other actions. An action must lock an atomic object in order to use it. Actions do not unlock atomic objects explicitly, however; unlocking is done automatically by the system as actions commit and abort.

The synchronization properties of actions are needed to ensure that the concurrent execution of actions is equivalent to some serial order [19]. Since actions can be nested, the synchronization rules must be defined accordingly (see [20] for a set of appropriate synchronization rules). The main issue is that when a nested action completes, the changes it made should be visible to its caller, but not to other actions, since it has not yet really happened.[5] When a top level action completes, then the new values of all atomic objects written by it or by any nested actions it caused to be called do become visible.

When a top level action commits, this also has an effect on permanent storage. Some subset

---

5. Greater concurrency can be provided by allowing the changes to be used, but then aborting the action that made the changes will cause the actions that used the results to abort too. See [21] for a discussion.

of the atomic objects modified on behalf of that action will be in some guardian's permanent state. The new versions of these objects must be saved (by the system) in permanent storage as part of completing the top level action. Since the modified atomic objects may be distributed among many different guardians, some form of 2-phase commit protocol will be needed [22].

As was mentioned earlier, our concept of an action is based on the idea of an atomic transaction that arose from work in data bases. One level transactions (no nesting) have been implemented in centralized data base systems [22], a distributed data base system [23], and in distributed file systems [24, 25, 26]. Multi-level transactions have not yet been implemented, although there is a growing recognition of the need for them. Proposals for multi-level actions can be found in [20, 27].

In addition to support for nesting, an important property of our work is that we permit users to define the data of interest (via the permanent state and atomic objects). In the systems mentioned above, the data is predefined (it is the data base or the files). In addition, we propose to integrate these notions into a programming language, which has not been done before.

There is some similarity between our approach and recovery blocks [7, 8], but also many differences. One important difference is that recovery blocks were not really intended for crashes, but for recovery from program errors, so there is no analog of guardian restart, or of a subset of guardian state that survives crashes.

5. Discussion

The previous sections discussed problems that arise in constructing distributed programs, and sketched some primitives that would aid the programmer in solving the problems. These proposals must be considered tentative, since we have not yet completed analyzing the proposed primitives and their interactions.

For example, in integrating actions and atomic objects into the language, the following issues must be addressed:

1. How can the programmer provide enough concurrency to achieve reasonable efficiency? For any given guardian, the main issue here is to provide concurrency between actions performed on behalf of higher level actions that have not yet committed. For example, in the buffer guardian, it may be important that a put action *p1* not delay another put action *p2* until the top level action that caused *p1* commits. This kind of concurrency is often much more important than actual concurrency within a guardian, e.g., whether the two put actions can actually run in parallel inside a buffer guardian.

2. Sometimes programmers do not want the call/return discipline inherent in the remote procedure call. For example, a call may be addressed to one guardian, and the reply come from another. Or, a sequence of calls may be made before a reply is desired. Can these patterns be accommodated?

3. The exact relationship of actions to the saving of permanent data needs more study. If saves are done early, e.g., as each action commits, then unnecessary work will have been done if an abort happens later. If saves are done only when the top level action commits, that commit may not be possible because of a crash at a node where a nested action was performed. Perhaps programmer control is desirable here.

4. Are other kinds of synchronization needed in addition to what is provided by atomic objects?

In spite of the need for further analysis, we are hopeful that our general approach is valid. We are certain that the difficulties described in Section 4.2 are real and must be overcome in implementing any application where reliability is essential. Primitives that solve these problems will be useful in those applications. We think they may also be useful in other applications, for example, a mail and message system, where people are willing to settle for less reliable behavior today. The main questions are: Can we identify the right primitives, which are simple to use and interact well with other features? And, can we implement the primitives efficiently enough that people are willing to use them? These are the questions our current work is attempting to answer.

Acknowledgements

The author gratefully acknowledges the efforts of the many people who contributed to the work discussed in this paper. Most important are the contributions of the members of the author's research group, including T. Bloom, M. Herlihy, P. Johnson, E. Moss, R. Scheifler, G. Stark and W. Weihl.

References

1. Liskov, B. H., Snyder, A., Atkinson, R. R., and Schaffert, J. C. Abstraction mechanisms in CLU. **Communications of the ACM** 20, 8 (August 1977), 564-576.

2. Liskov, B. H., Moss, J. E., Schaffert, J. C., Scheifler, R. W., and Snyder, A. **CLU Reference Manual.** Technical Report MIT/LCS/TR-225, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1979.

3. Brinch Hansen, P. Distributed processes: A concurrent programming concept. **Communications of the ACM** 21, 11 (November 1978), 934-941.

4. Hoare, C. A. R. Communicating sequential processes. **Communications of the ACM**, 21, 8 (August 1978), 666-677.

5. Preliminary ADA reference manual. **SIGPLAN Notices** 14, 6 (June 1979).

6. Feldman, J. A. High level programming for distributed computing. **Communications of the ACM**, 22, 6 (June 1979), 353-368.

7. Randell, B. System structure for software fault tolerance. **IEEE Transactions on Software Engineering**, 1, 2 (June 1975), 220-232.

8. Shrivastava, S. K. and Banatre, J. P. Reliable resource allocation between unreliable processes. **IEEE Transactions on Software Engineering**, 4, 3 (May 1978), 230-240.

9. Liskov, B. H. Primitives for distributed computing. **Proceedings of the Seventh Symposium on Operating Systems Principles**, December 1979, 33-42.

10. Roberts, L. G. and Wessler, B. D. Computer network development to achieve resource sharing. **Proceedings of the AFIPS 1970 Spring Joint Computer Conference**, May 1970, 543-549.

11. Clark, D. D., Pogram, K., and Reed, D. P. An introduction to local area networks. **Proceedings of the IEEE**, 66, 11 (November 1978), 1497-1517.

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

13. Hewitt, C. Viewing control structures as patterns of passing messages. **Artificial Intelligence**, 8, 3 (June 1977), 323-364.

14. Redell, D. D. **Naming and Protection in Extendible Operating Systems.** Technical Report MIT/LCS/TR-140, MIT, Laboratory for Computer Science, Cambridge, Ma., November 1974.

15. Herlihy, M. **Transmitting Abstract Values in Messages.** Technical Report MIT/LCS/TR-234, MIT, Laboratory for Computer Science, Cambridge, Ma., May 1980.

16. Herlihy, M. and Liskov, B. **Communicating Abstract Values in Messages.** Computation Structures Group Memo 200, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1980, submitted for publication.

17. Lampson, B. and Sturgis, H. **Crash Recovery in a Distributed Data Storage System.** Xerox PARC, Palo Alto, CA, April 1979, unpublished.

18. Gray, J. Notes on data base operating systems. **Operating Systems, An Advanced Course,** American Elsevier, 1978.

19. Eswaren, K., Gray, J., Lorie, R., and Traiger, I. The notion of consistency and predicate locks in a database system. **Communications of the ACM,** 19, 11 (November 1976), 624-633.

20. Moss, J. E. B. **Nested Transactions: An Approach to Reliable Distributed Computing.** Technical Report MIT/LCS/TR-260, MIT, Laboratory for Computer Science, Cambridge, Ma., forthcoming.

21. Takagi, A. **Concurrent and Reliable Updates of Distributed Databases.** Technical Memorandum MIT/LCS/TM-144, MIT, Laboratory for Computer Science, Cambridge, Ma., November 1979.

22. Gray, J. et al. **The Recovery Manager of a Data Management System.** Research Report RJ 2623, IBM Research Laboratory, San Jose, Ca., August 1979.

23. Rothnie, J., et al. Introduction to a system for distributed databases (SDD-1). **ACM Trans. on Database Systems,** 5, 1 (March 1980), 1-17.

24. Israel, J., Mitchell, J., and Sturgis, H. Separating data from function in a distributed file system. **Proceedings of the Second International Symposium on Operating Systems,** France, October 1978.

25. Paxton, W. H. A client-based transaction system to maintain data integrity. **Proceedings of the Seventh Symposium on Operating Systems Principles,** December 1979, 18-23.

26. Swinehart, D., McDaniel, G., and Boggs, D. WFS: A simple shared file system for a distributed environment. **Proceedings of the Seventh Symposium on Operating Systems Principles,** December 1979, 9-17.

27. Reed, D. P. **Naming and Synchronization in a Decentralized Computer System.** Technical Report MIT/LCS/TR-205, MIT, Laboratory for Computer Science, Cambridge, Ma., October 1978.