M.I.T. LABORATORY FOR COMPUTER SCIENCE
Computer Systems Research Division

December 27, 1979
Request for Comments No. 180

Programs for Distributed Computing: An Interim Report of an Experiment

by Irene Greif

This paper is a report of progress and plans as of August 1979 of an experiment with implementations of applications programs. The focus is on relevance to language design especially in support of communications protocols, conversational continuity and sharing of data via circulation of forms. A more detailed report of recent work on the calendar application is forthcoming.

1. Introduction

Computer systems that reflect the structure of the organization that they serve are likely to consist of networks of cooperating but autonomous nodes -- personal computers, departmental computers, branch computers. For some functions the nodes will choose to cooperate closely. However, each will have the capacity to protect itself, e.g. by refusing to serve certain requests or even by disconnecting itself from the system. Programs for individual nodes will have to be robust, able to tolerate network failure and capable of taking responsibility for resuming conversations after local failures.

This paper is a report on our experience with design of a language for distributed systems application programs. Since there are few examples of such programs, experimentation will play an essential role in the design. To this end we have been implementing applications programs, trying to focus on a variety of language issues. We report here on the approaches we are taking to studying communication protocols, facilities for maintaining conversations, and the sharing of information via message passing.

Section 2 is a description of the application area. In section 3 we describe the language we have been using and its relation to two other languages for distributed computing, and then discuss some language issues which can be addressed by programming in this language. Section 4 contains a brief report on the completed implementations.

2. The Application

This section contains a description of the calendar application domain and discussions of a few system specification alternatives.

2.1 Calendars

A distributed calendar system is fairly typical of the sorts of subsystems that will make up the fully automated office and is also a good source of interesting communication patterns. We will implement both personal calendars and semi-public calendars such as the schedule for some resource. We assume that personal calendars exist on individuals' personal computers, that they may not always be available, and that there is no shared data base of information about people's schedules. It is in this last assumption that we differ from other calendar systems, even those for distributed environments such as that of [Gifford, 1979]. The implication of this assumption is that all communication must be done via message passing.

Calendars will have fairly simple user interfaces and will offer a variety of services from the simple recording of one's personal appointments to the arranging of meetings that involve coordination among many people and resources. An example of the potential complexity of the communication is found in our laboratory when a conference room must be reserved for a seminar. The seminar must be scheduled at a time convenient for the host and speaker, the room must be available for the length of the talk plus some additional time for refreshments. The coffee pot must be reserved and arrangements made for it to be set up at the appropriate time. Conflicts with other seminars in other rooms must be prevented, announcements of a variety of sorts must be sent out, blackboards must be cleaned, and so on.

2.2 Forms

There are a wide range of applications which can be viewed in terms of the model of an office that operates by the circulation of forms. The form serves both as a communication medium and as a storage medium. To provide this view, the user interface should consist of procedures for filling in forms.

There are several ways in which forms can be worked into the design of a calendar system. The most straightforward use of forms is to closely mimic their use in offices. The forms are passive data structures which are circulated. In the calendar application they might be reservation of cancellation request forms. The calendar would then be a processor of forms.

Forms may have associated procedures which are used for guidance in filling out the form. Such a procedure could, for example, implicitly fill in additional fields based on an answer to one question. Also, some routing and additional communication could be initiated as the form is filled in. For example,the calendar itself can be thought of as a form. Then when one writes "meeting with John Doe" in the "August 13 at 4pm" field, some communication with John Doe's calendar should be initiated.

A personal calendar can be designed as a user interface for filling in a variety of forms, such as forms for requesting the scheduling of meetings, cancellation of appointments, etc. Alternatively, the user might fill in his own "active" form that contacts his personal calendar and possibly other calendars according to how it is filled in. We will implement both kinds of calendar system, since the implementations of each will use different communication protocols and data structures.

2.3 Reliability

For a variety of reasons, a message may not reach its destination. Therefore, when an expected response is not received after a reasonable delay, one might try to retransmit. In the initial design we have placed much of the responsibility for retransmission on the person who uses the calendar. The programs do have to provide reliable back-up storage (referred to as stable storage), so that the calendar can file copies of its outstanding requests in stable storage. The calendar provides an operation which allows the user to check the status of a request. If insufficient progress has been made, the user can either retransmit or abort. The calendar display helps by flagging appointments which are unconfirmed. This design will be adequate if the underlying system is relatively reliable. In a later version, intended for use on hardware that is known to be unreliable, the specification will include stronger reliability requirements. In that version we will learn more about language support for communication protocols (*cf.* section 3.4).

2.4 The Data Base

Each calendar maintains its own private data base which includes appointments, reminders, and outstanding requests. Completed transactions are recorded in the data base in stable storage.

3. Language Issues

We describe first the programming language which we are using and then go on to introduce the notion of a form as a programming aid. The subsequent sections pose some questions about language issues.

3.1 The Experimental Language

The programming language is a fairly simple extension of the sequential language CLU. We have adopted several of the early design decisions of the Extended CLU project [Liskov, 1979a], but have diverged somewhat from the current proposals of that group.

An application will be implemented as a group of *guardians* each of which is a single process with its own address space. Guardians have *ports*, one of which is created when the guardian itself is created. There are primitives for sending and receiving *messages*. Messages are sent to ports and *timeouts* are associated with receive statements. Ports have type determined by the kinds of messages they can receive. A message has a header, which is an identifier (e.g. *request* or *confirmation*) and is otherwise similar to a record in that it consists of labelled components of fixed type. Messages can contain objects of any type, including user defined types.

Our language is also similar to PLITS [Feldman, 1979] which is an Algol-like language with send and receive statements. We have used CLU as a base language for its data abstraction facilities, but otherwise agree with the PLITS philosophy which implies that an experimental language should be a simple one, subject to modification after it has been used.

3.2 Forms as Messages

Specifying that a calendar have a user interface for filling in forms does not necessarily imply that forms will actually be transmitted in the implementation. Messages may consist of values extracted from forms, thereby reducing the amount of information transmitted. However, this apparent savings in size of messages may be at the cost of program complexity. We have found that it is often more difficult to properly interpret these shorter messages.

A simple example arises in the case of a single "secretary" program which is communicating with a resource calendar to make several reservations. Each request can include a reply port, to which the answers **OK** or **FULL** will be sent. Then by maintaining a record of the correspondence between ports and requests, the secretary can interpret a reply according to the port on which it is received. If instead of **OK** or **FULL** the resource calendar returned a copy of the request marked **CONFIRMED** or **UNAVAILABLE** the secretary could receive all messages on one port and refer to the contents of the reply itself for details of the request. In longer conversations and other communication patterns, the savings in program complexity increases.

Implementations in which all messages contain forms have proved to be simpler in many respects than alternative programs that did not observe this discipline.

3.3 Conversations

Problem: Can one guardian maintain several conversations at once? How can understandable code for such a guardian be written?

A guardian can participate in more than one conversation at once, but keeping the conversations separate may prove difficult. Mechanisms for identifying a conversation include the use of multiple ports on which to receive messages (so that there is one port per conversation) and *transaction keys* [Feldman, 1979] that can be included in each message. To continue a conversation, there must be a record of the state of the conversation. This state can be represented by the values of local variables plus the program counter of a sequential program, hence suggestions such as multiple processes within a guardian [Liskov, 1979a]. Alternatively, since the state of a conversation can be stored in a form, conversations can be conducted in terms of forms. When the messages contain sufficient information, conversations can be programmed clearly without modifying the simple single process guardian language construct.

3.4 Communication Protocols

Problem: Can we identify patterns of send and receive statements which appear frequently and should therefore be supported by language constructs?

An example of a communication protocol is a program loop in which a message is retransmitted until an acknowledgement is received. A language for distributed systems might provide a "guaranteed acknowledged send" primitive to replace repeated occurrences of this loop.

We have found several simple protocols in our programs, usually matched pairs of sends and receives. Complex protocols such as the remote invocation discussed in [Liskov, 1979b] have not yet arisen. As we discussed in section 2.3, we have been passing much of the responsibility for reliability to the user. When we implement a calendar program that is expected to run on

a very unreliable network, additional protocols will certainly appear.

To summarize, we expect to continue to find more complex protocols, but in initial implementations have been comfortable programming the simple protocols explicitly.

3.5 Type Declarations

Problem: Will strong typing of messages and/or ports aid or interfere with good program design?

The most notable problem has been the potential for very long recursive definitions of types. The difficulty arises from the fact that messages can contain ports, and the type of a port is the list of the types of all its messages. The problem is similar to full declaration of the types of procedures which take procedure arguments.

It is easy to come up with scenarios of long recursive type definitions. Basically, the type declaration becomes an outline of all possible communication patterns. Writing this outline may in fact be a useful exercise for a programmer, but it is likely that shorter alternatives will be needed. One feature that would help is a single port type which subsumes all the different typed ports. Then a message type could be declared simply to include a port.

When using forms, the number of distinct port types is not so great and declarations become simpler. Within one application, messages usually contain the same form, and are distinguished only by their headers. Thus a few syntactic aids for listing the headers and declaring the type of the form would support full type declarations for communication in terms of forms.

3.6 Permanent Storage

Problem: How will the programmer indicate which values must be stored in stable storage so that execution can be resumed after a crash?

In an application in which forms are used, when a form is "filed" it is assumed to be stored safely. Currently we find that the application level notion of filing a form can double for the programmer as a tool for recording state. Filing copies of outstanding requests also serves as a way to save the states of conversations which must be resumed after a crash.

3.7 Synchronization for Sharing Data

Problem: How can a large data base be shared if guardians do not share data?

If guardians can fork internal processes which share data, then a data base could be implemented within a guardian. Concurrent reads would be performed by parallel processes. Synchronization would be enforced by a monitor or a similar synchronization primitive.

Unfortunately, the introduction of parallelism within a guardian adds enormous complexity to the language and conflicts with the simple model of communicating autonomous nodes. Instead we have been programming guardians that are managers of data. One way to provide concurrent access the data base is to partition the it into sections that will be managed by other guardians. All requests for data base operations are received at one port in the managing guardian and then forwarded to the appropriate guardian.

The final decision on language support for shared data may rest on one's taste in programming style. Our experiment will contribute to the resolution of this question by providing a body of programs in which data sharing is managed by guardians.

3.8 Forms Reconsidered

Problem: How should forms be supported in the language?

We are just beginning to explore the uses of forms in programming. We anticipate the need for syntactic aids for resuming conversations, so that the programmer can be relieved of having to extract all the necessary values from the form to update local variables. Forms for the calendar application are not very large, but in applications with larger forms optimization would be desirable. Perhaps only changes to forms or names of forms should be sent in messages.

We are currently using the data abstraction facilities of CLU to define forms. When we implement calendars which are "active" forms, we will have to reconsider the question of whether a form is a data object or a guardian. It is not yet clear how active forms can be circulated in messages. Also, we must clarify the differences between the kinds of forms that arise in application design and the forms used as message structures in our programming language.

To summarize, we consider language support for forms to be a major research topic to be tackled after we have more experience with the use of forms in programming distributed applications.

4. Current Implementations and Remarks about Final Paper

Several versions of the calendar application have been implemented already. There is a user interface to a personal calendar which is being used by several people. This calendar has a simple display, and can be used to make and cancel reservations and to add reminders and notes to days. A conference room calendar accepts reservations and cancellations when written on "reservation/cancellation" forms. In one implementation, personal calendars communicate via a mixture of passive forms and other unstructured data. We are currently debugging a second implementation in which the personal calendars communicate entirely in terms of forms. Conversations are much simpler in this version.

5. References

Feldman, J. 1979. High Level Programming for Distributed Computing. *CACM* 22, 6. June 1979. pp. 353-367.

Gifford, D.K. 1979. Violet, an Experimental Decentralized System. To appear in proceedings of *SOSP*. Dec. 1979.

Liskov, B. 1979a. Primitives for Distributed Computing. MIT - Computation Structures Group Memo 175. To appear in proceedings of *SOSP*. Dec. 1979.

Liskov, B. 1979b. Remote Invocation Protocols. MIT-DSG Note 45. July, 1979.