Distributed Programming in Ada with Protected Objects

by

Pascal Ledru

ISBN: 1-58112-034-6

DISSERTATION.COM

1998

Copyright © 1998 Pascal Ledru
All rights reserved.

ISBN: 1-58112-034-6

Dissertation.com

1998

www.dissertation.com/library/1120346a.htm

DISTRIBUTED PROGRAMMING IN ADA WITH PROTECTED OBJECTS

by

PASCAL LEDRU

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in
The Department of Computer Science
of
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

1995

ABSTRACT
School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science College/Dept. Science / Computer Science

Name of Candidate Pascal Ledru

Title Distributed Programming in Ada With Protected Objects

As distributed applications become more sophisticated, their implementation becomes more and more difficult. It is therefore important to study how to facilitate the implementation of efficient distributed applications. This thesis reviews the different classes of distributed languages and presents a new approach to develop efficient distributed programs using the Ada language. This approach is compared in detail with existing distributed programming languages, existing approaches to distribute Ada programs, and the Distributed Annex of the new revision of the Ada language.

Abstract Approval: Committee Chair S. G. Shiva 10/26/95
(Date)

Department Chair [signature] 10/30/95

Graduate Dean [signature] 11/6/95

THESIS APPROVAL FORM

Submitted by Pascal Ledru in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Science.

Accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee:

S. G. Shiva 10/26/95 Committee Chair
(Date)

10/26/95

10/26/95

____

____

____

Department Chair

College Dean

Graduate Dean

ACKNOWLEDGMENTS

First, I would like to express my gratitude to Dr. Sajjan G. Shiva for his support and guidance as the thesis progressed. He has been very helpful with comments and suggestions. Secondly, I would like to thank the other members of my committee, Dr. Terence J. Reed and Dr. Hui Wang, for their comments.

Last, but not least, I would like to thank my wife Sungmi for all her help and encouragement in the preparation of this thesis, which seemed at times as though it would never end.

TABLE OF CONTENTS

Page

List of Figures..................... vii

List of Tables..................... viii

Chapter Page

1. INTRODUCTION..................... 1

2. DISTRIBUTED PROGRAMMING LANGUAGES..................... 3

2.1 IMPORTANCE OF LANGUAGE SUPPORT FOR PROGRAMMING DISTRIBUTED SYSTEMS..... 4

2.1.1 Parallelism..................... 4

2.1.2 Interprocess Communication and Synchronization..................... 6

2.2 LANGUAGES FOR PROGRAMMING DISTRIBUTED SYSTEMS..................... 10

2.2.1 Languages using a message passing mechanism..................... 10

2.2.1.1 Message Passing - CSP..................... 11

2.2.1.2 Rendezvous - Ada 83..................... 11

2.2.1.3 Remote Procedure Call - Distributed Processes..................... 12

2.2.1.4 Multiple communication primitives - SR..................... 12

2.2.1.5 Object oriented languages - Emerald..................... 14

2.2.2 Languages using a shared data mechanism..................... 15

2.2.2.1 Distributed Data Structures - Linda..................... 15

2.2.3 Languages using Distributed Shared Data Objects..................... 16

2.2.3.1 Orca..................... 16

2.2.3.2 Distributed Eiffel..................... 21

2.2.3.3 Ada 95..................... 22

2.2.3.4 Concurrent C/C++..................... 24

2.3 SUMMARY..................... 25

3. ADA AS A LANGUAGE FOR PROGRAMMING DISTRIBUTED SYSTEMS....... 27

3.1 THE ADA 95 MODEL 29

3.2 THE DISTRIBUTED PROTECTED OBJECT MODEL........ 32

4. DISTRIBUTED PROTECTED OBJECTS 35

4.1 TRANSFORMATION 36

4.2 RESTRICTIONS AND ASSUMPTIONS 36

4.3 TASKS........ 38

4.4 PROTECTED OBJECTS........ 44

4.4.1 Protected Functions 49

4.4.2 Protected Procedures 50

4.4.3 Entries 54

4.4.4 Requeue Statement 60

4.5 COMMUNICATION SUBSYSTEM........ 63

4.6 EXAMPLE - THE TRAVELING SALESPERSON PROBLEM 64

5. CONCLUSION 67

APPENDIX - THE TRAVELING SALESPERSON PROBLEM........ 69

REFERENCES........ 75

LIST OF FIGURES

Figure Page

2.1 Classification of languages for distributed programming...10

4.1 Remote Rendezvous 44

4.2 Remote Protected Operation Call 45

LIST OF TABLES

Table Page

4.1 TSP Performance Measurements 66

Chapter 1

Introduction

A distributed system is defined as a collection of autonomous computers linked by a network, with software designed to maintain an integrated computing facility. Distributed systems can be programmed using three different methods. The first builds applications directly on top of the hardware. Although this method is highly efficient, it is extremely hardware dependent, difficult to implement, and non-portable. The second uses an existing sequential language plus a collection of operating system primitives. This method is also non-portable since it is operating system dependent. The third uses a programming language containing all the facilities for expressing distributed programs. This method is highly portable and also makes it easy to program by providing a higher level of abstraction.

When distributed systems first appeared, they were programmed using the first and the second methods. As distributed applications became more and more sophisticated, these methods were no longer satisfactory. In order to solve this problem, distributed programming languages have emerged. Distributed languages can be divided into two categories: languages with *logically distributed address space* and languages with *logically shared address space*. The latter has the advantage of hiding complexity of communication over the former.

Ada is a language which supports communication of sequential processes [Hoare, 1978], and therefore should be able to support distributed systems programming. Ada has

a tasking mechanism, and according to the Ada Reference Manual [Ada 95 Reference Manual], it is possible for tasks to be distributed within an Ada program. Parallel tasks may be implemented on *multicomputers*, on multiprocessors, or with interleaved execution on a single physical processor. Extensive studies have been undertaken to distribute Ada programs, and several authors pointed out that the Ada 83 Reference Manual was incomplete regarding the distributed issues [Volz et al., 1987] [Volz et al., 1989]. Due to the incompleteness of Ada 83, Ada 95 contains an annex based on the *Remote Procedure Call* paradigm. It describes a model for execution of programs on distributed systems.

Another enhancement of Ada 95 is a new feature, the *protected object,* which is intended for synchronized access to data on shared memory systems. Even though the protected object is intended for a shared memory system, the future of the computer technology lies in the distributed environment. With this in mind, this thesis examines the possible implementation of *distributed protected objects*. There are existing languages which support concept similar to the protected object over a logically shared address space. But these languages (e.g., Occam and Linda) suffer from the following limitations: they are experimental and often architecture dependent compared to Ada. Therefore, for this thesis, Ada was the choice for implementing distributed protected objects.

The thesis is organized as follows. Chapter two reviews characteristics of several distributed programming languages and shows the advantages of the *logically shared address space* paradigm over the *logically distributed address space* paradigm. Chapter three first reviews how Ada 83 implements distributed programs and points out limitations of the language. Secondly, Annex E, Distributed Systems of Ada 95, is closely reviewed. The chapter ends with the introduction of a new model: distributed protected objects. Chapter four presents the implementation of the model. Chapter 5 concludes the thesis.

Chapter 2

Distributed Programming Languages

As stated previously, distributed applications can be built directly on top of the hardware, on top of an operating system, or in a special language for distributed programming. The first method provides total control over all primitives provided by the hardware, such as interfaces to communication devices. Although this method allows efficient utilization of the available resources, it has the severe disadvantage of being hardware dependent, and hence is not practical for developing large software systems.

The second method uses an existing sequential language plus a collection of operating system primitives accessed through library routines. Applications developed using this method can be made hardware independent, but they will always be operating system dependent.

The third method employs a programming language containing all the necessary constructs for expressing distributed programs. This method shields the application programmer from both the operating system and the hardware. As a second major advantage, such a programming language can ease the programming task by presenting a higher level, more abstract model of a distributed system.

Many languages for distributed programming have already been developed and implemented. A closer look at these languages is the main subject of this chapter. But before examining some of these languages which represent different classes in detail, this chapter reviews how parallelism and communication and synchronization are supported in distributed systems, programmed with or without language support.

2.1 Importance of Language Support for Programming Distributed Systems

A distributed application can be characterized by parallelism, and interprocess communication and synchronization. Distributed applications differ from their sequential counterparts in such a way that the former contain sections of code which run in parallel. Among different sections of the code exchange of information may be necessary, and thus requires communication and synchronization between the sections. Due to the fact that parallel portions are independent of each other, failure of one should not prevent the rest from continuing their execution. The following sections examine parallelism, and interprocess communication and synchronization in detail and show why it is important to provide language support.

2.1.1 Parallelism

In most distributed operating systems, the unit of parallelism is a process. The Unix *fork* system call creates a process with an execution environment copied from the parent process which executed the system call. The *exec* call transforms the calling process into the one executing the code of a named program. The *rexec* and *rsh* calls extend the exec call to remote calls. All these system calls can provide a user with parallelism, but it is the user's responsibility to distribute work among different processes using these system calls. On the other hand, there are languages which have parallelism built in. For example, in Occam [Inmos, 1984], there is a construct called *statements*. These statements can be executed either sequentially, as in

```
SEQ
   S1
   S2
```

or in parallel, as in

```
PAR
   S1
   S2
```

In a pure functional language, functions behave like mathematical functions: they compute a result that depends only on their input data. Such functions do not have *side effects*. If functions do not have any side effect, the order in which they are executed makes no difference. For example, in the expression

h(f(3,4), g(8))

it is possible to evaluate f and g in parallel [Hudak, 1986].

Logic programs can be read *declaratively* as well as *procedurally*. In the code below, two *clauses* for the predicate A are given:

(1) A :- B,C,D.

(2) A :- E,F.

The declarative reading of the clauses is "if B,C and D are true, then A is true"(Clause (1)) and "if E and F are true, then A is true"(Clause (2)). Procedurally, the clauses can be interpreted as "to prove theorem A, you either have to prove subtheorems B, C, and D, or you have to prove subtheorems E and F." From the procedural readings, it becomes clear that there are two opportunities for parallelism:

(1) *OR-parallelism* - The two clauses for A can be worked on in parallel until one of them succeeds or both fail.

(2) *AND-parallelism* - For each of the two clauses, the subtheorems can be worked on in parallel until they all succeed or any one of them fails.

As seen in the previous examples, built-in parallelism of a distributed language can determine which parts can be executed in parallel. And there are also languages which support concepts similar to processes provided by an operating system, as a part of

language definition. For example, Ada supports *tasks.* The main advantage of having such constructs is portability.

2.1.2 Interprocess Communication and Synchronization

In most distributed operating systems, distributed processes communicate by *message passing*, and operating systems typically support two forms of message passing mechanisms: Blocking or non-blocking primitives, and reliable or non-reliable primitives. Even though some operating systems provide higher level of abstractions such as *Remote Procedure Call* (RPC) or broadcasting, these are operating system specific. In hybrid environment, it is not easy to distribute work across different operating systems. For instance, an operating system supporting RPC (Amoeba) cannot communicate with another supporting only message passing primitives (Unix) unless much work is done to simulate RPC. Programs written on top of a specific system are therefore difficult to port to other systems. [Bal, 1990a] gives a specific example of the difficulties encountered in porting an application developed for the Crystal multicomputer which uses asynchronous message passing to Amoeba which uses (synchronous) RPC. Another area in which distributed operating systems are not adequate is *strong type checking*. Distributed operating systems only support exchange of byte streams; they do not check whether the sender and the receiver are interpreting the parameters consistently. In addition, problems arise while trying to send a complex data structure such as a tree. Usually, the programmer has to write a subprogram to convert the data structure to a sequence of bytes.

The problems described above can be minimized if interprocess communication is directly supported by languages. Languages provide interprocess communication by *message passing* and *shared data*. The message passing allows two entities to communicate with each other by exchanging the data explicitly. The shared data can be

conceived as a mailbox. A process deposits data and the other retrieves, and thus interprocess communication occurs implicitly.

In message passing, there are two issues of importance. The first issue is how the sender and the receiver address each other. In the simplest case, both the sender and the receiver explicitly name each other. Requiring the receiver to specify the sender of the message is rather inflexible, precluding the possibility of receiving messages from any other client (for example, a file server). A solution is to allow the receiver to accept messages sent by any process. This form is called asymmetric direct naming. Another solution is indirect naming which uses an intermediate object, like a port rather than process names. Ports are more flexible than direct naming since neither the sender nor the receiver needs to know the identity of the other. The second issue concerns synchronous versus asynchronous message passing. Both use *send* and *receive* primitives but differ depending on when the sender continues. With synchronous message passing, the sender waits until the message has been delivered at the receiving process; i.e., the receiver has finished executing the *receive* statement and has stored the message in its local variable. With asynchronous message passing, the sender continues immediately after issuing the *send* statement. Synchronous message passing is simpler to use, because when the sender continues it knows the message has been processed. But synchronous message passing is more restrictive since the sender blocks until the receiver receives the message. Some languages support both synchronous and asynchronous communication (e.g., SR). Other languages (e.g., CSP, Occam, Ada 83) provide only synchronous message passing.

The send and the receive operations used above transfer information in one direction, from the sender to the receiver. In many cases, the receiver wants to return a reply to the sender. With the send and the receive primitives, the programmer needs two messages, one for each direction. An alternative is to use a single message passing construct that transfers data in both directions. Two-way interactions occur in the *client-server* model.

The client sends a request and the server returns a reply. Two widely used message passing mechanisms based on this principle are the *rendezvous* and the *remote procedure call.*

The rendezvous mechanism is based on two-way message passing and explicit message receipt. With the rendezvous, the receiver (server) contains operations that senders (clients) can request to execute. Such procedures are called *entries* in Ada. The declaration of an entry is similar to that of a procedure. If a client issues a request to execute an entry, it is said to do an entry call. The client now blocks until the entry call has been processed by the server and it has received the result of the call.

Like the rendezvous, the Remote Procedure Call is based on two-way message passing. The idea behind RPC is to allow communication between the client and the server through conventional procedure calls, even if they are on different machines. The Remote Procedure Call mechanism is described in detail by [Birrel and Nelson, 1984].

Parallel programming languages on *tightly coupled* (shared memory) systems use not only mechanisms such as the rendezvous or other forms of message passing, but also shared variables. Variables are defined as *shared* if they can be accessed or modified by several threads. In order to ensure integrity of the shared variables, the access must be explicitly synchronized. The synchronization on these shared variables is usually ensured by low-level primitives such as *semaphores, monitors,* or *critical sections* which are easy to misuse and can lead to programs which are difficult to maintain. Shared data is defined as an abstract data type which encapsulates both variables and operations on these variables. Operations defined on these shared data enforce synchronization implicitly, and thus is hidden from the users.

The shared data concept also can be extended to parallel programming languages on *loosely coupled* (distributed memory) systems. At first glance, it may seem unnatural to

use shared data for communication among distributed systems, as such systems do not have physically shared memory. Nevertheless, the shared data paradigm provides several advantages over message passing. A message passing mechanism is usually used to communicate between two processes. On the other hand, shared data can be used to communicate among any number of processes. A process which updates shared data does not need to know the location of the other processes using the shared data [Bal and Tanenbaum, 1988]. Finally, an assignment to a shared data has immediate effect; i.e., it guarantees *order-preserving*; such order-preserving may be harder to obtain if several processes communicate using a message passing mechanism [Bal and Tanenbaum, 1988].

The shared data model can also be viewed as an abstraction layer over the *distributed shared memory* (DSM). The DSM provides a virtual address space shared among processes on loosely coupled processors [Nitzberg and Lo, 1991]. In DSM, replication is usually used in order to improve performance, and the unit of replication is a physical page. Of course, like the shared variables described above, since multiple processors can access the DSM, data synchronization has to be explicitly provided. Conversely, the shared data model is integrated in a programming language, and it is superior to the DSM model in terms of programmability and readability. Better performance can be achieved [Levelt et al., 1992] since only the shared data are replicated instead of physical pages [Bal et al., 1992b].

Several languages support the shared data model using different mechanisms. These languages are *Shared Logical Variables* in Parlog, *Tuple Space* in Linda, *Shared Data Objects* in Orca, *Protected Objects* in Ada 95, *Capsules* in Concurrent C/C++, *Classes* in Distributed Eiffel, and *Resources* in Pascal-FC [Burns and Davies, 1992]. In the following sections, Linda, Orca, Ada 95, Concurrent C/C++, and Distributed Eiffel are discussed in detail; Ada 95 is also discussed in detail in the following chapters. Some implementation details of the Shared data objects model of Orca are also presented.

2.2 Languages for Programming Distributed Systems

The languages reviewed in this thesis are categorized into three: languages using a message passing mechanism, languages using a shared data mechanism, and languages using both mechanisms. Each category of languages is further partitioned into a number of classes. The classification is illustrated in Figure 2.1.

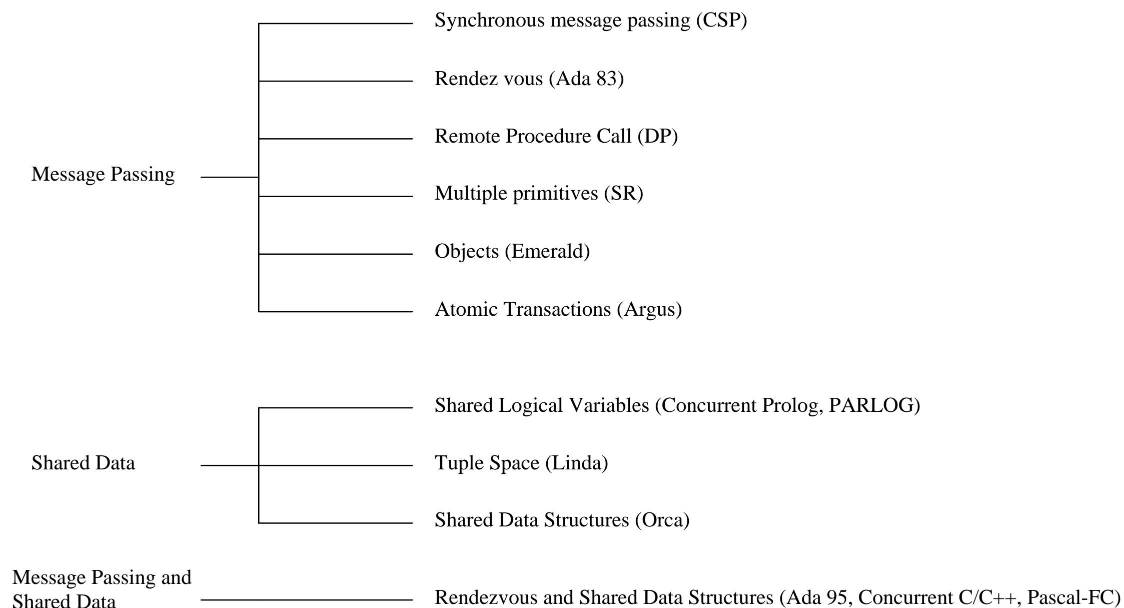

Figure 2.1. Classification of languages for distributed programming

For each language, the following sections describe how parallelism, communication, and synchronization are expressed in the language, and how parallel units are mapped onto processors if the language addresses this issue. For some languages, some implementation details are also given.

2.2.1 Languages using a message passing mechanism

This section presents several languages supporting message passing, rendezvous, RPC and multiple communication primitives.

2.2.1.1 Message Passing - CSP

The Communicating Sequential Processes (CSP) [Hoare, 1978] model consists of a fixed number of sequential processes that communicate only through synchronous message passing.

Parallelism: CSP provides a simple parallel command to create a fixed number of parallel processes. CSP processes take no parameters and cannot be mapped onto specific processors. An array of similar processes can be created, but their number must be a compile time constant.

Communication and Synchronization: CSP processes may not communicate through global variables. All interprocess communication is done using synchronous receive and send. The sending process specifies the name of the destination process and provides a value to be sent. The receiving process specifies the name of the sending process and provides a variable to which the received value is assigned. A process executing either a send or a receive is blocked until its partner has executed the complementary statement.

2.2.1.2 Rendezvous - Ada 83

Ada 83 supports applications that can use multiple processes and run on *multicomputers*, on multiprocessors, or with interleaved execution on a single physical processor [Ada Programming Language].

Parallelism: Ada provides the task as the unit of parallelism. Tasks are identical to a process, and they can be created dynamically.

Communication and Synchronization: Ada tasks communicate by using the rendezvous mechanism and by global variables. Tasks synchronize with each other during a

rendezvous; then they continue their normal execution independently of each other. A task can call an entry of another task by using an *entry call statement*, similar to a procedure call. An entry call specifies the name of the task containing the entry as well as the entry name itself. Ada uses a *select statement* for expressing nondeterminism. Ada's select statement is used for three different purposes: to select an entry call nondeterministically from a set of requests, to call an entry conditionally (i.e., only if the called task is ready to accept it immediately), and to set a timeout on an entry call.

2.2.1.3 Remote Procedure Call - Distributed Processes [Brinch Hansen, 1978]

Parallelism: In Distributed Processes (DP), the number of processes is fixed at compile time. The intention is to dedicate one processor to execute each process. Each process, however, can contain several threads of control. A process definition contains an initial statement which may be empty if this is the first thread; a thread may continue forever, or it may finish executing at some point. But in either case the process itself continues to exist; DP processes never terminate. Additional threads are initiated by calls from other processes. Arrays of processes may be declared. A process can determine its array index using the built-in function*this*.

Communication and Synchronization: DP processes communicate by calling the other's common procedures. Such a call has the form: call P.f(exprs, vars) where P is the name of the called process and f is the name of a procedure declared by P. The expressions such as exprs are input parameters; the return values such as vars of the call are assigned to the output variables. The calling process and all its threads are blocked during the call. A new thread of control is created within the called process P.

2.2.1.4 Multiple communication primitives - SR

Synchronization Resources (SR) [Andrews et al., 1988] is based on Modula, Pascal, and DP, and provides several models of interprocess communication.

Parallelism: An SR program consists of one or more *resources*. A resource is a module which runs on one physical node (either a single processor or a shared memory multiprocessor). Resources are dynamically created and optionally assigned to run on a specific machine. An identifier for the resource instance is returned by the *create* command. A resource can contain several processes, and these may share data. Synchronization among these processes is supported by the use of semaphores. Communication with processes in other resources is restricted to *operations*. A resource may contain an initialization and a termination process. These are created and run implicitly. A resource terminates when it is killed by the *destroy* command. A program terminates when all its processes terminate or block.

Communication and Synchronization: An SR operation definition looks like a procedure definition. Its implementation can either look like a procedure or an entry point. When implemented as a procedure, the operation is serviced by an implicitly created process. When implemented as an entry point, it is serviced by an already running process in a rendezvous. The two types of implementation are transparent to the invoker of the operation. On the invoker's side, an operation may be called asynchronously using a *send* or synchronously using a *call.* Several calls can be grouped in a parallel call-statement, which terminates when all calls have been completed. The operation and its resource instance must be named explicitly in the invocation. This is done using the identifier for the resource returned by the *create* command. By combining the two modes of servicing operations and the two modes of invoking them, four types of interprocess communications can be expressed.

SR uses a construct similar to the Ada *select* statement to deal with nondeterminism. The SR's *guarded command*, or *alternative,* has the following form:

```
entry_point(params) and bool-expr by expr -> statements
```

Within an operation, a guarded command may contain an entry point, a Boolean expression, and a priority expression. An alternative is enabled if there is a pending invocation of the guarded command and the Boolean expression associated with it evaluates to true. The expression in the *by* part (`by expr -> statements`) is used for priorization when there are several pending invocations of the same guarded command. If all Boolean expressions are false, the process suspends.

2.2.1.5 Object oriented languages - Emerald

Emerald [Black et al., 1987] is an object-based programming language for the implementation of distributed applications. Emerald considers all entities to be objects. For the programmer, both a file accessible by many processes and a Boolean variable local to a single process are objects. Objects are either passive or active. Emerald is categorized as *object based* since it does not support inheritance. Abstract types are used to define the interface to an object.

Parallelism: Parallelism is based on simultaneous execution of active objects. The language supports process migration by moving objects from one processor to another. Such a move may be initiated either by the compiler using compile time analysis or by the programmer using language primitives.

Communication and Synchronization: An object consists of four parts: a name, a representation, a set of operations, and an optional process. The name uniquely identifies the object within the distributed system. The representation contains the data of the object. Objects communicate by invoking each other's operations. There can be multiple active invocations within one object. The optional process runs in parallel to all these invocations. The invocations and the data shared among them can be encapsulated in a monitor construct. The internal process can enter the monitor by calling an operation of its own object. Emerald provides the same semantics for local and remote invocations.

2.2.2 Languages using a shared data mechanism

As opposed to the message passing mechanism, data sharing has several advantages in interprocess communication. This section reviews some languages based on a shared data mechanism.

2.2.2.1 Distributed Data Structures - Linda

Linda [Ahuja et al., 1986] [Carriero and Gerlernter, 1989] consists of a set of primitives which can be integrated into another language, such as C or Ada, and provides parallelism to the existing language. Linda limits itself to four constructions and the notion of *Tuple Space*. The Tuple Space is the core component of Linda, and this data structure may be distributed. Distributed activities can use Linda for communication and synchronization. The Tuple Space stores a collection of tuples, and processes produce and consume tuples. A tuple is an ordered set of heterogeneous elements which are called *fields*. A consumer waits until a producer drops a matching tuple in the Tuple Space, and this is the way it expresses synchronization and communication.

(1) Parallelism. Linda provides a simple primitive called *eval* to create a sequential process. Linda does not have a notation for mapping processes to processors.

(2) Communication and Synchronization. *Out* drops a tuple in the Tuple Space. *Rd* reads a tuple without removing it from the Tuple Space. *In* reads a tuple and removes it from the Tuple Space. For example:

```
out("tuple1",1,false)
in("tuple1", int i, bool b)
rd("tuple1", int i, bool b)
```

All these operations are atomic; i.e., they are not interrupted until completion.