A Shared Object Environment for Developing Distributed Applications

Zair Abdelouahab
Department of Electrical Engineering,
Federal University of Maranhão
São Luís 65080-040, Brazil
zair@dee.ufma.br

Francisco José da Silva e Silva
Department of Electrical Engineering
Federal University of Maranhão
São Luís 65080-040, Brazil
fssilva@ufma.br

Mohamed Mohsen Gammoudi
Department of Computer Scince,
University of Science of Tunis
Tunis, Tunisia
Mohamed.Gammoudi@fst.rnu.tn

Abstract

This paper presents an environment support for developing distributed applications (ESDDA) in the form of a library. It supports programming through shared objects. The environment provides a collection of predefined classes through which users can instantiate objects that can be accessed from any machine transparently in a distributed network environment. The classes refer to common data structures such as queues, stacks and bags. They provide an interface for creating, using and manipulating object instances. ESDDA offers as well a mechanism in which programmer can create objects (not shared) that can be accessed from remote clients.

1. Introduction

The development of cheap processors, workstations, personal microcomputers, powerful and fast networks led to change from centralised model of computing to the distributed one.[1] The programming model evolved as well in that applications are written in the form of separate client/server components.

The object oriented technology has been pointed to be a promising one to control the complexity generated by distributed systems. The concepts of object orientation such as modularity, access to data through the interface makes the use of objects appropriate to model distributed systems. This approach has been enforced and well accepted with the development of several distributed environments that use the concept of objects (e.g. CORBA of OMG[14] and OLE of Microsoft. Another aspect that should be taken into consideration is that object oriented applications are programmed in terms of communicating objects which is a natural form to program distributed systems [11,13].

The objective of this paper is to present a programming environment based on shared objects to facilitate programming distributed applications. This work is organised as follows. Section 2 describes existing distributed programming environments, in particular environments based on objects and SVM (Shared Virtual Memory). Section 3 presents the model of computation of ESDDA environment. Section 4 describes the implementation of ESDDA. Finally, section 5 presents some remarks and conclusions found during the development of this work.

2. Background

2.1 Emerald

Emerald is a language developed for supporting distributed applications, based on the object oriented paradigm. The language has been implemented on various platforms and in particular, on a network of distributed workstations. An application in Emerald is composed as a collection of cooperating objects which could exist on nodes of the network. The primary characteristics of Emerald are:

- Localisation of objects is transparent during a remote invocation. The caller object and the callee object may move nodes during the call.
- Emerald is a strongly typed language.
- Support of migration of objects
- Transparent replication of immutable objects

**Proceedings of the Workshop on Computer Science and Information Technologies CSIT'99
Moscow, Russia, 1999**

An object in Emerald has a localisation which specifies the node it which it resides actually. It is the role of the run time system to determine that localisation during the call (remote invocation) and perform the operation. However, an application may decide according to its necessity which configuration of objects is best on the network with respect to: proximity to resources, load balancing, interaction between specific objects ..etc. For this purpose Emerald offers the following primitives that are related to localisation of objects:

- Locate: to determine where an object resides
- Fix: to maintain an object on a node
- Unfix: to free and object in order to migrate to other nodes
- Move: to move an object to other specified nodes.

2.2. Distributed Cm

Distributed Cm [7] extends the sequential Cm [16] with mechanisms to allow distributed computing such as creation of distributed objects, transparent communication between objects, exception handling, communication ports and concurrency inside objects. A remote object is defined as an object that has its own context of execution, separate from the one of its creator, and from those of its clients. A remote object may execute on any node of the machine and may be different from the node of its creator. A remote object is invoked in the same form as other objects. When created, its information are kept with its creator. A remote object can be shared by other objects when it is known to them. One way of making it known to others is that at creation, a symbolic name is associated to the object and registered with the server of names of the ESDDA environment. Clients may access the symbolic name to access the remote reference.

Distributed Cm permits another model of communication using ports. It provides a predefined class of ports implemented within the run time system. In order to achieve communication between objects, ports are associated to them and should be connected.

2.3. SVM Shared Virtual Memory

SVM systems are introduced by Kai Li [8] and extend the concept of virtual memory found in operating systems to systems that have distributed memories. SVM systems use similar techniques but with a difference in that the whole addressing space is virtual and paginations occur between a processor and disc and between processors. Thus, the memory of each node is used to store the pages that can be used by the local processor and is seen as a big local cache. SVM systems use a write coherence protocol to a memory to make sure that the value returned by a read operation is the last value of a write operation. There may be various copies of a page for reading but only one for read and write. When a process attempts to write a page on a node which does not have the copy of read-write, a fault page occurs and the process is suspended. The fault page is handled by a routine which informs all nodes to invalidate their copies and bring the read-write to the local node. Nevertheless, this mechanism has similar semantics of conventional virtual memory systems, SVM systems may be inefficient. For example, two nodes that wish to write continually the same page, the phenomena of thrashing may occur []. To resolve this problem, Munin system [3] permits to the programmer to specify different mechanisms of coherence to the stored data.

2.4 Linda

Linda[10,1] is a DSM system in which its computational model is based on a shared space called Tuple Space TS). A process wishing to communicate with another process creates an entity called a tuple, then deposited in the tuple space. The receptionist process should remove or copy the tuple within the tuple space. The tuple space is logically shared between all processes and is implemented as distributed components between processors.

The tuple space consists of a collection of tuples of passive data and tuples of active processes. Tuples of data are registered and identified with a name associated to them. Tuples of processes are routines which execute and when they terminate they become data tuples. Linda offers three basic primitives to manipulate the data tuples: in, out, rd. Out deposits one tuple inside the tuple space; in removes one tuple from the tuple space whereas rd copies a tuple from TS. A process which executes in or rd blocks if there is no tuple in TS corresponding to the requested one. There exist a non blocking operations of in and rd that are inp and rdp respectively. These operations returns values that are true if there are compatible tuples and values of false otherwise. If returned values are true, then the effect of inp and rdp are the same as in and rd respectively. Linda permits as well a creation of processes inside the tuple space. These processes execute in an independent form of their creators. When the active tuples terminate execution, they become data tuples.

In Linda, there is no mechanism for changing a tuple in the tuple space. A tuple has to be removed from TS by a process, update it, then put back inside TS. This mechanism avoids the coherence problems found in SVM but contention may be generated in case several processes try to update the same tuple.

The primary problem of Linda is the difficulty in implementing the tuple space in an efficient manner. This is because processes do not know where to search for a tuple which is needed. Several implementation strategies have been attempted and some specific ones are successful. However, in several situations, Linda demonstrated a poor performance. Linda suffers as well from the fact that it has an unpredictable semantic of performance. Finally, the tuple space is inflexible since the construction of complex data structures is difficult.

2.5. Shared Objects

In Shared Objects (SO), processes communicate through abstract data types called objects. Programs are written in a sequential language like C and linked to a library of predefined abstract data types such as tree, queue, stack, grid, list and bag. These abstract data types are instantiated and manipulated at run time through an interface which they provide.

SO provides a single and common address space which is shared by all processes in the system. At each address, an object can be created. Objects are instantiated and destroyed using primitives of Create and Destroy, respectively. To instantiate an object, a programmer should supply its type (which data structure to be used) and the address within the common address space. For example, an object of type queue is to be created at address 102 in the common address space. Each process can create any object at the addresses of the common space. Once an object is created, it can be accessed any process to modify it or destroy it.

Processes interact with objects through message passing. Each object has an interface in which processes may invoke its methods. For example, an object of type queue has methods for inserting and removing elements, whereas an object of type stack provides methods for pushing and popping elements.

SO provides some meta functions. Meta functions are operations that exist independently of the interface of a particular object. They are used for creating, destroying and controlling objects. The function Create is used to instantiate an object at a given address; destroy removes an object residing at a given address; AddressWait allows waiting until a specified address is allocated or liberated; EventWait allows a process to block until an event occurs.

The use of operations for removing elements in the blocking form are util. However, these operations do not resolve the case where a process wants to read in a blocking form from several objects. For this purpose, SO provides operations for reading with events where processes may wait in the future. With this mechanism, a process may emit various operations of reading from different objects associated to the same event.

3. Computational Model

3.1. Introduction

The ESDDA environment presents to its users two functions that give support to the development of distributed applications.

The environment provides a collection of classes through which users can instantiate objects that can be accessed by any node present in the machine in a transparent form, giving illusion that the objects reside locally. These classes refer to data structures that are common in programming such as queues, stacks, and bags. Objects of these classes are shared between distributed components of the application.

The second function refers to method calls of a remote object by providing transparency of localisation. The environment offers a mechanism through which users can create objects that can be accessed by remote clients. With this mechanism, an object server can be created through the provided library which controls the communication between the client and the object that contains the method to be invoked. On the other side, if a user wants to achieve an invocation of a method that belongs to remote object, he should first instantiate an object of type proxy from the library, which will represent the remote object. A request for executing a method is made to the proxy object which communicates with the server object resident in the remote machine.

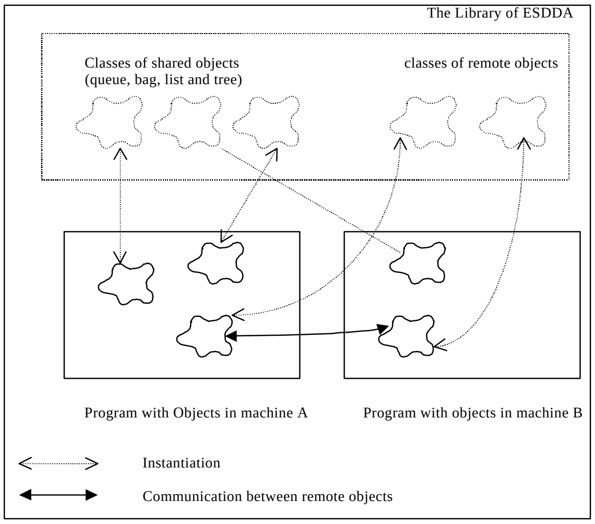
The Library of ESDDA
Classes of shared objects (queue, bag, list and tree)
classes of remote objects
Program with Objects in machine A
Program with objects in machine B
Instantiation
Communication between remote objects

Figure 1 : Computational Model

Figure 1 presents a general view of the computational model. The environment offers a library containing a set of classes through which users may instantiate objects within their programs. Some of the classes are related to the first function of the system (i.e. classes of data structures). Other classes refer to access to remote objects with the client/server model.

3.2 Basic Components of ESDDA

Figure 2 describes the principal components of the environment and their interactions.

The library consists of a set of pre-defined classes in C++, through which users can instantiate objects that represent common data structures such as queue, priority queue, list, stack, tree and bags. Objects that are created from these classes are called shared objects (so) and can be accessed transparently by any process from any machine.

Other component illustrated in figure 2 are part of the RUN Time System (RTS). Each machine that uses the environment runs a kernel. The kernel is responsible for storing the data and controlling accesses to shared objects created locally. In addition the kernel is responsible for handling waiting for events which will be discussed in future sections. The other component of figure 1 is the Global References Repository. The repository is responsible for maintaining a data base containing information such as type, subtype, and localisation of every object defined in the environment. The types indicates if the object is shared or remote. subtype is used for shared objects to indicate their types such as queue, stack, and list.

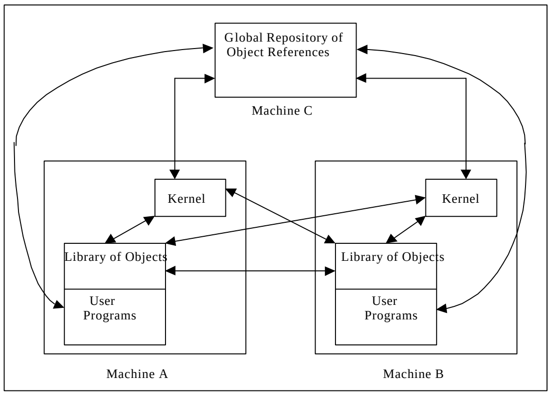

Figure 2 : Componentes do Ambiente

The most important component is the library. The library is defined in C++ and its diagram is illustrated in figure 3 using the notation of Booch [5].

The class *SharedObjectsClass* (of shared objects) is an abstract class and contains all data and methods which are common to every type of shared objects (e.g. queue, list, bag, and stack). In particular, it contains the name of the shared object, the maximum number of elements which can be stored and the maximum length of its elements. This class contains as well methods for creating and destroying objects. We can notice from the diagram that every type of shared objects inherits components of the class *SharedObjectsClass*. The types of objects that are supported by the environment are initially: queue, stack, binary tree and bags. We can remark that some classes are specialisation of other classes. This is the case of priority queue which is a subclass of the class queue. From this initial diagram, it can be extended to support other new data structures that can be shared (e.g. n-ary trees, grids).

Another class of the diagram that manipulates shared objects is the class *EventClass*. In some cases, a user process may need to wait for an element to be inserted in one object belonging to a group. For example, a user program may need to wait at the same time for an element to be inserted in a queue or in a stack. This environment provides a definition of event to achieve that purpose in the class *EventClass*.

Programming with Shared Objects

Programming with shared object in the proposed environment is very simple. The first step consists of instantiating objects. To create a new object, the user needs to instantiate it from one class related to the library of data structures (e.g. queue, stack, list, bag). The user is required to supply the global name of the object, the maximum elements of that particular objects.

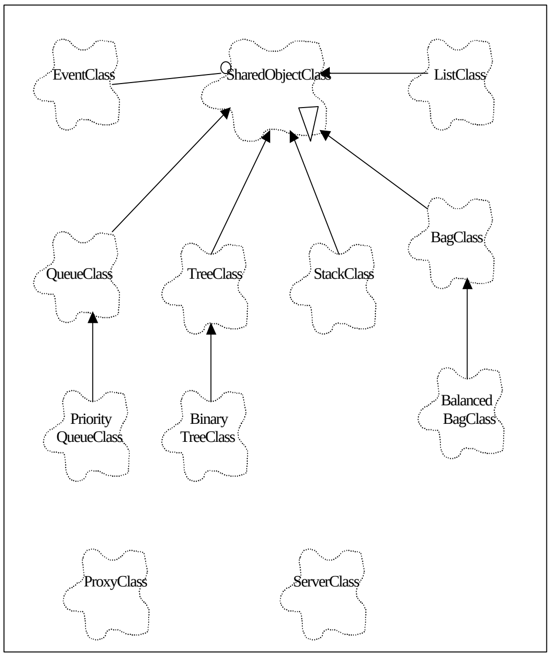

Figure 3 : Class Diagram of ESDDA

The following code shows the syntax for creating a shared object:

```
QueueClass my_queue (“shared_queue”, 1000, 1000);
my_queue.CreateSharedObject ();
```

The code above instantiate an object of type Queue called *shared_queue*. This queue has a global name “*shared_queue*” which can be used by other processes to access the object. The code states that the object *shared_queue* may have a maximum of 1000 elements and these elements may have a maximum of 1000 bytes.

Once the shared object is created, it can be accessed transparently by other objects or processes transparently by knowing just its global name. To allow programs or processes to access a shared object, the user has to instantiate a local object from the same class (i.e. *QueueClass*) and associate it with the global name of the shared object. Any reference to the local name is an access to the shared object. Note that the user do not need to supply parameters such as the maximum number of elements and the maximum number of bytes. The fragment of code for achieving this is shown below:

```
Code = QueueClass my_queue (“shared_queue”);
```

When the above code is executed, the following values may be returned:

0 1 indicates that the operation is complete with success
1 -1 indicates that the object “*shared_queue*” does not exist
2 -2 indicates that the object called “*shared_queue*” exists but of type different from queue.

If the return code of the above function is –1, thus the object is not created yet. The user can wait until the object *shared_queue* is created. The environment provides a function waitexist for this purpose which is part of the abstract class *SharedObjectsClass*.

```
my_queue.waitexist();
```

The environment blocks the process executing the above function until the global object is created by another process or object.

Manipulating Shared Objects

For each type of shared objects, the environment provides methods through which these are manipulated. For example, the environment offers methods for inserting and removing elements to/from an object queue and methods for pushing and popping elements to/from an object stack.

Removing elements from shared object can be done in a blocking or non-blocking form. With the non blocking form, if a shared object is empty (no elements), a code (integer value) is returned to the user process indicating to him that there are no elements and the process execution can continue immediately. In case of a blocking removal from an empty shared object, it causes that particular process to block until another process inserts an element.

Another aspect related to the manipulation of objects is that the environment does not control the type of elements inserted/removed. Each element inserted/removed is just a collection of bytes. It is the responsibility of the programmer to do the appropriate casting at the time of insertion and removal. At insertion time, the programmer should supply a pointer to the element and it length in bytes.

Let us illustrate how to manipulate an object of type queue:

```
my_queue.insert(ptr_data, length);
```

This code causes an insertion of an element whose pointer is ptr_data and length length inside the object *my_queue* (i.e. shared object *shared_queue*).

```
my_queue.remove(ptr_data, &length);
```

This code causes a non blocking removal of an element whose pointer is ptr_data and length length from the object *my_queue* (i.e. shared object *shared_queue*).

```
my_queue.removeB(ptr_data, &length);
```

This code causes a blocking removal of an element whose pointer is ptr_data and length length from the object *my_queue* (i.e. shared object *shared_queue*).

Creating Events

Some problems may require the programmer that its programs necessitate waiting for an element from more than one shared object. For example, a program may request elements from more than one queue object at the same time. The blocking remove operation as provided cannot handle this situation since the blocking is done only at one queue object. One solution for the programmer is to employ a busy waiting with a remove operation from each queue. However, this solution leads to inefficiency. To resolve this situation, the environment provides to users events through which it is possible to wait on various shared objects (e.g. queues).

To use events, users need to manipulate the class *EventClass*. In particular, to instantiate an event, a user should supply the name of the event. Then, users needs to inform the environment what objects are involved with the event. This is done through methods supplied within the classes of those objects. For example, if one of those objects is of type queue, the member method *removeE()* of the class *QueueClass* should be used. The parameters that should be supplied as part of the remove operation are: the location where the element should be copied, the address of the variable which should hold the length of the element, and the name of the event associated with the object. Below is illustrated an example using events:

```
/* Creation of two objects queues */
QueueClass queue1 (“QueueOne”, 1000, 100);
queue1.CreateSharedObject ();
QueueClass queue2 (“QueueTwo”, 1000, 100);
queue2.CreateSharedObject ();

/* Creation of events */
int length;
EventClass ex_event (“Example_Event”);

/* removing with events */
queue1.removeE (ptr_var, &length, ex_event);
queue2.removeE (ptr_var, &length, ex_event);
```

Once the event is created and it is associated to a set of shared objects, the users needs to call the method *EventWait* which is a member of the class *EventClass*. The execution of this method causes the process to wait until the occurrence of the event. In the example given above, the event that is waited for is the removal of an element from *QueueOne* or *QueueTwo*.

```
/* Waiting for the event */
char *objectname;
ex_event.WaitEvent (&objectname);
```

The function *EventWait* returns as a parameter the shared object from which the element is removed. The environment returns the element and its length to the locations supplied within the operation *removeE* corresponding to that particular objects.

3.4 About Remote Objects

Programming with remote object requires two steps. Initially, the programmer needs to handle the form in which methods of a remote object are offered. The second step is concerned with the form in which remote objects are invoked (i.e. invocation of methods of the remote object).

To provide objects that can be accessed remotely by other distributed objects, the programmer should use the class *ServerClass* that is part of the environment. An object of this class will control the communication between the program and its clients that requests remotely the execution of methods that are member of objects created inside the program.

Once a remote object is created, it can be accessed by any clients which knows its global name. To access a remote object, the program should instantiate a proxy object of this object. The instantiation is done from a class *ProxyClass* offered within the library. The user should supply the global name of the object to access as part of the instantiation. The class *ProxyClass* has a method call() used by the program to invoke methods of the remote object.

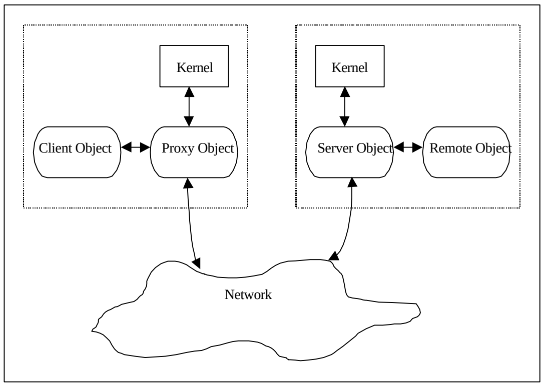

Figure 4 : Invocation of Remote Objects

The proxy object is responsible for sending requests of invocation of remote methods to the remote object and receiving results from the remote object, conform to figure 4. The details related to programming with remote objects are omitted in this paper, but can be found in [17,18].

3.5. Comparison with Other Environments

Distributed Cm

The approach taken Distributed Cm is different from the one taken by our environment in that the first is an extension of a language whereas ours is a library. Since Distributed Cm is an extended language, thus it is natural that it provides more flexible and high level constructs. However, libraries are more portable. Programming with our environment using shared objects is simple since it provides nearly the same semantic of local objects.

ESDDA offers a set of predefined objects that encapsulates data structures that are shared in a distributed environment. It has an advantage in that shared objects reside in the kernel which can manipulate them and execute various procedures that can optimise the performance when accessing them. The kernel may migrate objects transparently from one node to another.

With Emerald

Emerald is a language that is defined from scratch to support distributed applications. Providing a new language leads to more flexibility in offering various functions of a system. However, this option does not take advantage of application codes already written with popular languages.

Both our environment and Emerald provide transparency in accessing objects. Moreover, Emerald permits as well explicit indication of localisation of objects. It provides primitives for fixing and moving objects. Emerald does not provide support for predefined objects as is in our case. As mentioned above in comparing with Distributed Cm, our environment can implement predefined objects efficiently to achieve a good performance.

Emerald supports both inter and intra object concurrency. It uses monitors for achieving synchronisation. Our environment does not support intra object concurrency. Emerald has a strong relation between the language, compiler and the system. A positive aspect is that the full system is implemented efficiently to support programming with distributed objects. A negative aspect is its portability to other architecture.

With SO

The concept of shared object in our work came from SO [10]. This work is for supporting parallel and distributed application in general. Some original aspects of SO have been modified to accommodate our necessities. The main modification refers to the programming paradigm. SO offers an interface for programming in the conventional and procedural paradigm whereas our environment uses the object orientation. This has lead to the modification of other components. For example, our system does not provide a single address space as in SO.

Both environments SO and our provides predefined shared objects. However, this can limit functions of the system to program general distributed applications. Our environment provides another mechanism which is programming with distributed objects (i.e. remote objects) inspired from [12, 17].

4. Implementation

A prototype environment has been implemented on a network of distributed workstations, SPARC running UNIX Solaris of Sun Microsystems. The language has been implemented with Concert/C [2], a version of C that supports distributed programming, developed by IBM. Concert/C provides various mechanisms such as RPC, asynchronous message passing, process management, and EBF (external Binding Facilities). Figure 2 illustrates components of the implemented environment.

4.1. Environment Components

The Global repository of references keeps a database in which all object references are recorded. For each object, the database keeps its name, type (which identifies if the object is shared or remote), subtype (that indicates the data structure type of a shared object), and a reference (bind) to the kernel (machine) responsible for that particular object. In addition to these functions, the repository is also responsible for controlling object creation events. This is the case where a process may block until an object is created by another process. It is the responsibility of the repository to inform processes waiting for an object to be created when the latter is created by another process.

Th kernel is a component which is responsible for storing data objects and controlling every access to a shared object. There is one active kernel on every machine of the running environment. When a user program runs on a machine A and requests a creation of a shared object, the request is sent to the local kernel which creates the object and registers it within the global repository. This makes the created object accessible to every machine of the environment. Every request to the shared object (e.g. insertion, and removal) are handled by the kernel. Another responsibility of the kernel refers to the blocking remove of elements from shared objects. When a user process makes a blocking remove request to an object which is empty, the kernel registers the request and when another process inserts an element to that particular object, the kernels wakes up the process and returns to it the element.

The last component of the environment is the library. The library is linked to user programs and is responsible for routing local requests to appropriate components. In case of shared objects, the requests are directed to kernels that maintain them. In case of remote objects, the library communicates to the server objects. When the library receives results from the server objects, these are directed to appropriate user programs.

5. Conclusion

The environment has presented to the programmer an object oriented interface through which distributed applications may create, use and manipulate objects. The objects are made available to every process of the same application. Objects may interact in a distributed environment where communication and localisation are handled transparently by the environment, thus makes programming easier. The environment offers a collection of predefined classes through which users may instantiate objects which encapsulate common data structures. These objects are shared and their use is simple a does not require any special programming than the one with local objects.

There as certain components of this environment that are implemented in a centralised fashion, for example, the repository of objects. A better solution is to employ a decentralised strategy. Other future improvements of this environment include the distribution of objects. In this version, an object reside entirely on one node of the network. In case several processes want to access this objects, several requests should be directed to that particular node. This may generate contention. To avoid this problem, elements of each object could be distributed around the network.

Acknowledgement

We would like to thanks SUDENE and FAPEMA for their Financial support.

References

1. G. S. Almasi e A. Gottlieb: Highly Parallel Computing, The Benjamin/Cummings, 1994.
2. J. S. Auerbach et al: Concert/C Tutorial and User's Guide: An Introduction to a Language for Distributed C Programming, IBM, 1995.
3. J. K. Bennett et al: Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence, Proceedings of ACM Conference on Principles and Practice of Parallel Programming, 1990
4. A. Black et al: Object Structure in the Emerald System, ACM SIGPLAN OOPSLA'86, 1986
5. Grady Booch: Object Oriented Design with Applications, The Benjamin/Cummings, 1994
6. G. Coulouris, J. Dollimore e T. Kindberg: Distributed Systems Concepts and Design, Addison Wesley, 1994
7. Celso Gonçalves Júnior: Objetos Distribuídos, Unicamp DCC, 1994
8. Kai Li: Shared Virtual Memory on Loosely Coupled Multiprocessors, Yale University, 1996
9. Jeff Kramer: Distributed Software Engineering, IEEE, 1994
10. D. P. Mallon: A Shared Memory Model for Parallel Distributed Memory Machines, PhD. Thesis University of Leeds, 1992
11. M. Mülhäuser, W. Gerteis e L. Heuser: DOCASE: a Methodic Approach to Distributed Programming, Communications of the ACM, Sept. 1993.
12. H. Moons e P. Verbaeten: Object Invocation in the COMET Open Distributed System: the Dialogue Model, IEEE, 1992
13. J. R. Nicol, C. T. Wilkes e F. A. Manola: Object Orientation in Heterogeneous Distributed Computing Systems, IEEE, 1993
14. CORBA: Architecture and Specification, Object Management Group, 1995
15. R. K. Raj et al: Emerald: A General Purpose Programming Language, Software - Practice & Experience, 1991
16. F. Q. B. da Silva, H. K. E. Liesenberg e R. Drummond: Programação em Cm, Proceedings of XV SEMISH, pp 101-102, Rio de Janeiro, RJ, 1988
17. F. J. Wang, J. L. Chen e C. H. Hu: A Distributed Object-Oriented System with Multi-threads of Services, IEEE, 1993
18. Silva, F. J. S.: Ambiente de Apoio ao Desenvolvimento de Aplicações Distribuídas, Master thesis, UFMA, 1997.