Architecture and Implementation of Guide, an Object-Oriented Distributed System

R. Balter, J. Bernadat, D. Decouchant, A. Duda,
A. Freyssinet, S. Krakowiak, M. Meysembourg,
P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill,
C. Roisin, X. Rousset de Pina,
R. Scioville, G. Vandôme
Unité Mixte Bull-IMAG

ABSTRACT: This paper describes the architecture and implementation of an object-oriented distributed operating system. The system is called Guide (Grenoble Universities Integrated Distributed Environment). Its main features are the following: an object model is embodied in a language including the notions of type and class with single inheritance; execution units (jobs) are multi-threaded virtual machines that may dynamically diffuse to several nodes; objects are dynamically linked to jobs, and may be shared between jobs; the system provides a distributed object memory for the storage of persistent objects; the system supports synchronized objects and transactions. The paper describes the general organization of the system, execution structures, object memory and transactions. The first implementation on top of UNIX is described. Some performance figures and a first qualitative evaluation are given.

1. Introduction

The motivation of project Guide is to explore scientific and technical issues related to the support of large-scale applications distributed on a network of possibly heterogeneous machines. The goal of the project is to design and implement an experimental system which provides an environment for the development and operation of distributed applications. This paper describes the architecture and the implementation of the system.

Guide was started in 1986 as a joint research project of Bull Research Center and Laboratoire de Génie Informatique, IMAG (Universities of Grenoble). Guide is also a component of Comandos (Construction and Management of Distributed Open Systems) [Alves Marques et al. 1988], a project supported by the Commission of European Communities under the ESPRIT Program (projects 834 and 2071). While the overall design principles and the global object-oriented architecture are common to Comandos and Guide, the language and the system implementation described here are specific to Guide.

1.1 Requirements and General Orientations

Typical examples of the application classes that we plan to support are software development environments and advanced document processing systems. These applications are characterized by the following features. The data used by the applications consist of many pieces organized in complex structures. The data have different structures (e.g. text, pictures, programs, etc.) which involve different representations and access procedures. The data are persistent, i.e. their lifetime is independent of that of the programs or processes that use them. They exist in multiple versions. The applications usually involve the cooperation of several users; this involves concurrent computation, frequent communication and a high degree of data sharing.

Large scale applications impose some qualitative requirements on the supporting system and on the development environment. These requirements are essentially motivated by a trend towards the reduction of development costs through higher programmer productivity. The main requirement is *reusability*: it should be easy to reuse the investment, both in design and implementation. Mechanisms should be available to develop applications as extensions or specializations of existing programs. Another requirement concerns *maintainability* and *ability to evolve*. These properties are essential for long-lived applications, which are likely to undergo a number of revisions to adapt to changing user needs and to the progress of technology. The ability to evolve is especially important in a distributed environment, because the structure of the system favors evolution through incremental change. A final requirement, specific to the distributed and possibly heterogeneous nature of the system, is *integration,* i.e. the ability of the system to provide the illusion of using one large integrated virtual machine. The basic design decisions of the system were made in response to these requirements. We list them with their main motivations:

a) **Object-orientation.** The adoption of an object model is the most important choice of the system. It was motivated by the powerful structuring mechanism provided by objects: encapsulation of data together with code, independence between interface and implementation, object composition for building complex structures. It improves reusability through specialization, using the subtyping mechanism. This model also gives a possibility to deal with heterogeneous systems by using multiple implementations of an object interface. Moreover, dynamic substitution of one implementation by another (with a "compatible" interface) is an important mechanism for the design of evolving systems. In addition, an object model appeared to be a convenient vehicle for the integration of concepts and methods from operating systems, programming languages and databases, which resulted from the requirements.

b) **High-level language support.** The use of a high-level language is essential for maintainability and ability to evolve. In addition, we strongly support the view of an operating system as an execution environment for a programming language. This environment may be provided to the user as a set of primitives, or as a full-fledged language whose run-time environment is supported by the system. In Guide, we have adopted the second approach, which provides a better integration of the system and applications, and we have designed the Guide programming language for the expression of distributed applications.

c) **Object storage.** The system provides a permanent repository for objects, a substitute for a traditional file system. Objects give the user a unifying view of the system: they may be regarded both as a support for procedural and data abstraction, and as long-term storage units. This is the *persistent programming* approach. Objects may be combined together to form complex structures.

d) **Distributed virtual machine.** The execution mechanism available to the user should allow concurrent programming while hiding distribution as much as possible. The main execution abstraction provided by the system, called a *job*, satisfies these requirements. A job may be viewed as a multiprocess virtual machine, composed of distributed concurrent *activities* operating on objects. Since the objects may be located on different nodes, jobs and activities may be distributed. However, the distribution is hidden from the user. Communication between jobs, or between activities within a job, is done through shared objects. The system also provides a support for synchronization and transactions.

1.2 Related Work

Several recent research projects have investigated object-oriented languages and systems for distributed applications. We now examine how Guide relates to these efforts, with emphasis on two aspects: the object model and the specification of the virtual machine.

Eden [Almes et al. 1985] was one of the first experiments in the design and implementation of object-oriented distributed operating systems. An application in Eden is built as a collection of objects called

Ejects and written using the Eden Programming Language, derived from Concurrent Euclid. An Eject is an active entity and includes a number of processes which communicate via monitors. An Eject is typed, mobile and its invocation is location independent. The model integrates the object and virtual machine aspects: each Eject defines a multiprocess virtual machine. However, it contains a major part of the operating system, which complicates object management.

In Argus [Liskov 1985], a special emphasis is placed on reliable distributed computation. Argus extends the CLU language to support atomic transactions in a distributed environment. An Argus *Guardian* encapsulates the notion of a virtual machine. A Guardian contains data objects and processes. A Guardian communicates with another one by calling a handler in the target Guardian, to which data are passed by value. The Argus and Eden programming languages both support two kinds of objects: large, persistent units (like files) and small, non persistent units (like integers and records). This requires two different implementation mechanisms. A major difference between Argus (and Eden) and Guide is the granularity of objects. While Guardians (in Argus) and Ejects (in Eden) are usually large units, such as servers, objects in Guide are usually small (like files in traditional systems), and may be combined into larger units. This resulted in the decision to make Guide objects passive, in contrast with both Argus and Eden.

In Emerald [Black et al. 1987], a successor project to Eden, static type checking is introduced, and objects may be active or passive. The main features of the system include a single object model (unlike Eden and Argus), support for abstract types and an explicit notion of object location and mobility. The type conformity rules adopted for Guide, as well as the overall design of the run-time structures for method selection, are similar to those of Emerald. However, Emerald does not provide support for persistent objects, which is a major goal of Guide.

The main goal of Clouds [Leblanc & Appelbe 1988] is to develop a distributed, fault-tolerant computing environment (operating system and applications), which appears as a uniform, integrated computing resource to the users. The system is based on the notions of passive objects, location independent invocations, and nested actions. All objects can be viewed as existing in a shared global object space which is supported by a single-level store. As opposed to Guide, there is no global virtual object memory mechanism: an action may span several nodes, but internode communications are explicit. Threads are active entities performing invocations. Threads and objects can be associated with recovery attributes so that objects can maintain consistency in presence of failures. The system does not support class hierarchy nor inheritance.

Mach [Jones & Rashid 1986] defines a computational model composed of tasks (shared virtual memory) and threads (sequential processes running within a task). Communication uses shared memory (within a task), messages and ports. Jobs and activities in Guide may be compared, respectively, to Mach tasks and threads. An important difference, however, is that a job in Guide may span several nodes, and dynamically diffuse to other nodes. Mach was not initially designed as an object-oriented system, but it provides the foundation to build such a system. Chorus [Rozier et al. 1988] provides a computational model similar to that of Mach.

SOS [Shapiro et al. 1989] is a distributed, object-oriented operating system based on the *proxy* principle: the interface to a service, implemented by a set of cooperating objects, is a single communication object, a "proxy" for that service. The unit of allocation, communication, and storage is the object. The system implements only basic mechanisms for creating, invoking, migrating, and deleting objects. It does not provide a new language; applications are written in C++, enhanced by a dynamic binding mechanism. Communication between jobs in Guide also uses shared objects, but is not restricted to a client-server relationship.

Amoeba [Mullender et al. 1990] is a kernel primarily providing communication and remote execution services, based on a fast remote procedure call mechanism. Objects are implementations of data types such as files, directories, and processes, and they are managed by servers. A client process carries out operations on an object by sending a request message to the server which manages the object. All objects are named and protected by cryptographically secure capabilities.

In Guide, we attempted to draw on the experience of all these systems and to develop new ideas. Guide integrates several existing concepts: uniform object model with single inheritance, abstract data types, strong type checking, type and class hierarchy, hidden distribution, location-independent object naming, object mobility and atomic transactions. Guide is a language-based system that embodies recent advances in object-oriented techniques: separation between type and classes, multiple implementations of a type, mechanisms for the construction and conservation of complex objects, persistent two-level storage with garbage collection. The integration of all these features, individually present in several experimental systems, is an innovative aspect of the project. Another new aspect is a computational model based on multi-threaded virtual machines providing "windows" on the global object space, and a high-level synchronization mechanism based on activation conditions and counters. We plan to use the system as a base for exploring further domains: multiple inheritance, object clustering for rapid access, object replication, management of complex distributed structures, load balancing, distributed debugging and monitoring.

The first Guide prototype was implemented on top of the Unix system, in order to provide in a short time a version of the system able to support simple applications. It currently runs on the following workstations: Bull DPX 1000 and DPX 2000 running the SPIX system (System V based system with BSD extensions), Sun 3-60, Sun 3-80, Sun 386i and Sun 4 running Sun OS (BSD based system with System V extensions), and DEC 3100 DecStations running Ultrix. The prototype has been operational since September 1988 and several applications have already been written for it (a brief description of the first implementation appeared in Découchant et al [1988]). Further experiments and measurements will serve as a basis for an improved design.

2. *Architecture of Guide*

Guide can be viewed as a distributed, multiprocessor virtual machine. The distribution is hidden from the user; the parallelism is visible. Applications are structured in terms of objects. Objects are stored in a multi-site, location transparent secondary storage. They are loaded on demand into a virtual memory for execution. The system provides a support for synchronization and transactions.

The main execution unit used by applications is the job, which provides an object space and multiple concurrent activities. A typical application is implemented as a single job, which is started on a node and may dynamically diffuse to remote nodes according to the location of the objects that it uses. A complex application that needs to use several different address spaces (e.g. for protection) may be implemented by several cooperating jobs. At the application level, communication between jobs or activities entirely relies on object sharing.

We now present the details of the object model, object storage, execution structures and the support for distribution, parallelism and synchronization.

2.1 Object Model

This section gives a summary of the main features of the object model. This model is supported by a language (also called Guide) used to build distributed applications, which is briefly described in this section. A more detailed description, with examples, may be found in Krakowiak et al. [1990].

An object encapsulates data (the *state* of the object) and operations (also called *methods*). The data may only be accessed via method invocation. A *type* describes an object behavior shared by all objects of the type. This behavior is defined by the signatures of the methods and public variables. A *class* defines a specific implementation of a type: data representation and method code. Classes are used to generate instances, i.e. the objects whose data and methods are defined by the class. A given type may be implemented by different classes. Thus, different instances may coexist with the same interface and with different implementations. The word "object" is used here in two senses. The first one refers to objects as storage units, the second one refers to objects as instances of a class, i.e. entities containing private data and a reference to the class defining the methods. The objects (instances) and classes are uniformly stored as objects in secondary storage.

Subtyping allows the specialization of a given type. In the current version of the model, a type may only have a single supertype. Parallel to the subtype hierarchy there is a subclass hierarchy between classes implementing the types. A class inherits the methods of its superclass and it can overload them or specify other methods. Generic types and classes are also provided.

Objects are persistent, i.e. their lifetime is not related to the execution of a program: a persistent object exists as long as it is referred to by at least one other persistent object. A persistent object has a system wide, location independent, typed unique name called a *system reference*. References are not visible to the users; they are used internally for object invocation and for the construction of composite objects.

An exception model is also provided. The scope of exception handling is object invocation. This approach fits well our object model, where any object invocation may either terminate normally returning a result, or terminate abnormally raising an exception.

The above notions are illustrated by a few examples of type and class definitions. The following type declarations respectively define a document description and a library (a collection of documents).

```
TYPE Document_descr IS
     key: Integer;
     title, author: String;
     date_borrowed, date_returned: REF Date;
     METHOD Init;
          {initialize internal data}
     METHOD Get_Info;
          {display information about document}
     METHOD Get-Contents: REF Document;
          {gives access to the text of the document,
           defined by type Document, not specified here}
END Document_descr.

TYPE Library IS
     METHOD add_document (IN doc: REF Document_descr):
       Integer;
          {add a document to the library, returns a key}
     METHOD search (IN key: Integer; OUT doc:
       REF Document_descr);
              SIGNAL not_found;
          {look for a document specified by a key}
     METHOD borrow (IN doc: REF Document_descr);
              SIGNAL is_out
          {borrow a document from the library}
     METHOD return (IN doc: REF Document_descr): Integer;
          {return a borrowed document to the library}
     METHOD list;
          {print the contents of the library}
END Library.
```

In the above definitions, the keyword **REF**<type> defines a typed variable which refers to an object of type <type>; such a variable is essentially a container for a system reference. The semantics of **REF** parameters in method signatures is call by reference. Call by value is also possible, but it is restricted to objects of elementary types, such as Integer, Char, etc. The keyword **SIGNAL** specifies that a typed exception may be raised under specified conditions.

A class definition describes a particular implementation of a type. It includes the representation of the internal state of the object as a set of instance variables and the program of the methods. For instance, a possible implementation of type Library is defined by the following class descriptions:

```
CLASS Library_list IMPLEMENTS Library IS
     doc_list: REF List OF REF Document;
     METHOD search(IN key: Integer; OUT doc:
        REF document);
            SIGNAL not_found;
            {looks for a document specified by its key}
            BEGIN
            <program of search using list representation>
            END search;
     ...
     <program of other methods>
     ...
END Library_list.
```

As usual in object-oriented languages, an instance of this class is created by calling the method New of the class. If lib is a variable of type Library, the statement lib:=Library_list.New creates a new (uninitialized) instance of class Library_list and assigns it to lib. Thereafter, methods may be invoked on this object by calls like lib.search(...). The variable lib may be reassigned to another object of type Library, possibly with a different representation (e.g. an instance of another class Lib_array that would use an array as an internal representation).

2.2 Object Storage

Objects in Guide reside in a distributed permanent secondary storage. An object is entirely located on one node, i.e. there are no fragmented distributed objects; however, an object may contain references to other objects on remote nodes. The distribution of objects is hidden from the programmer, who does not need to know where a particular object is located. In addition, objects can migrate from one node to another.

The addressing and execution space provided by jobs may be related in two possible ways to the permanent storage supporting objects. Either it is an addressing window on a large, single-level global object space, or it acts like a cache for a permanent repository where objects are stored, but cannot be directly manipulated (a two-level store). The concept of a *persistent object memory* that appeared in recent work on object-oriented databases [Thatte 1986], is an example of the first design choice. In this approach, every object is always directly addressable without explicit loading and linking operations. The second approach is similar to distributed file mapping. An example of a distributed implementation is proposed in the Apollo/Domain system [Leach et al. 1983]. We adopted the second approach because we do not think that a single-level distributed object storage can be efficiently implemented in the current state of the art (remember that it has to be garbage-collected). Therefore, our global object memory is implemented as a two-level storage. At the lower level, a Storage Subsystem (SS) is in charge of long-term conservation of persistent objects. At the upper level, a Virtual Object Memory (VOM) acts as a cache supporting the execution of jobs (i.e. objects linked to jobs are addressed in the VOM). Both VOM and SS are distributed.

The Virtual Object Memory is composed of the contexts of all jobs. It can be viewed as a uniform space of all objects currently linked to jobs in the system. Thus, only a single image of an object (class instance) should be present at a given time in virtual memory. The code of classes is shared between activities executing on the same node. Several copies of classes may exist on different nodes. A mapping between object references and virtual addresses is maintained for all objects used by a job.

The Secondary Storage subsystem is organized as a set of logical containers. A container is an abstraction of a physical secondary storage unit. It may be composed of several replicated physical containers for better safety and availability. The physical containers are mapped on different nodes. The secondary storage provides a set of primitives supporting persistent objects in a distributed environment.

2.3 Execution Structures

2.3.1 Jobs and Activities

A job is composed of sequential threads of control called *activities*, which operate on passive objects. A job may span several physical nodes, it may dynamically extend itself or shrink, according to the pattern of object invocations. The execution of an activity consists of successive invocations of methods on objects. The invocations may take place on any node in the system. During each invocation, the referenced object is located, loaded and dynamically linked to the virtual memory of a job. The set of all objects linked to a job is called the *context* of the job. A job provides an addressing window on the global object space through the mapping of objects shared by its activities. Jobs and activities are system-wide, distributed notions.

A job is created by an activity of another job using a creation primitive. The primitive specifies an initial method of an object and creates a main activity to invoke the method. Initially, the new job entirely resides on the creation node. Once started, the main activity invokes the initial method. Then, as other (possibly remote) objects are invoked, the context is extended. New activities may be created and the job may diffuse to other nodes. There is no explicit communication between activities; however, they can communicate through invocations of shared objects. Jobs and activities are represented in the system as instances of predefined system objects. The user can control their execution by invoking methods such as *create, start, suspend, resume, destroy*.

The concepts of jobs, activities and job context are illustrated in Figure 1. There are two jobs: the context of job *J1* is composed of ob-

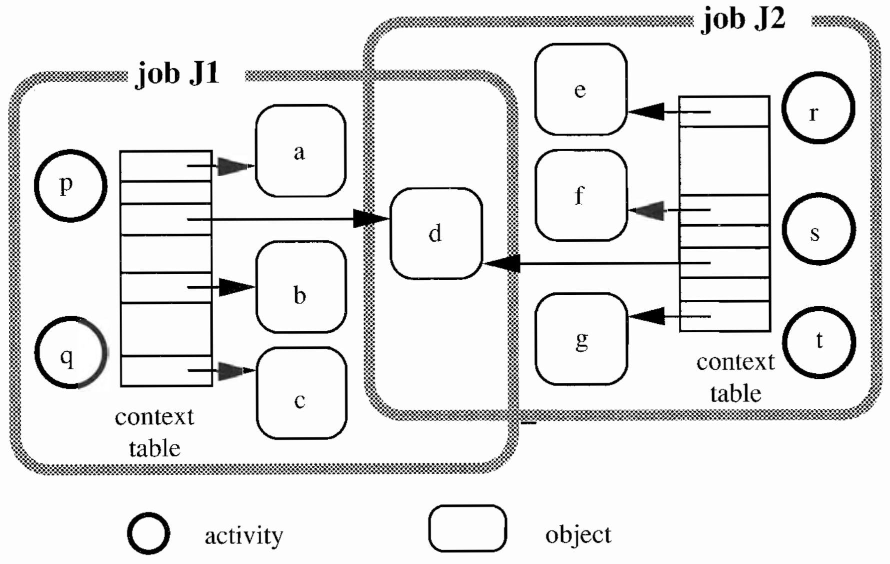

Figure 1: Jobs and activities

jects *a, b, c, d* and the context of job *J2* is composed of objects *d, e, f, g*. Object *d* is shared by the two jobs. Activities *p, q* execute within job *J1* and activities *r, s, t* execute within job *J2*.

The context of each job is represented by a context table maintained by the kernel. Each object present in the context of the job has a descriptor in this table. More details will be found in section 3.2.

An important design decision was how to relate objects to execution structures. Two solutions could be adopted: a) associate execution structures with objects, i.e. define active objects, each object containing a fixed or variable number of processes; and b) separate objects from execution structures, i.e. define passive objects executed by independently defined processes. We did not find strong arguments in favor of either solution. Both have been adopted in existing object-based systems (e.g. active objects in Emerald [Black et al. 1987]; passive objects in Clouds [Leblanc & Appelbe 1988], Amoeba [Mullender et al. 1990], and SOS [Shapiro et al. 1989]). The two solutions are dual: both can offer the same functionality to the user, but in different ways. The choice is mostly influenced by considerations of efficiency and adequacy to the hardware and to the applications. Our system is intended to be a collection of services, each service being visible as an object. The user, or more precisely a job representing the user behavior, selects a desired service and executes it by calling one of its methods (i.e. entry points). In the active object model, execution structures are associated with objects acting as servers and providing some service to passive users. The passive object model can be compared to a self-service supermarket and the active object model to a traditional store. In the first case, the user chooses goods by himself; in the second case, he is served by a server. It appears to us that passive objects contain less context information than active objects. Therefore their creation, invocation and migration can be implemented more efficiently than for active objects. The classes of applications that we intend to support involve creating many (usually small) objects, and building large compound structures out of object components. Parallelism is likely to be coarse-grained. Passive objects are well suited for such applications. The active object model is conceptually simpler, since a single abstraction encompasses the two concepts of processes and data structures. It is also suitable for ensuring synchronized and protected access to objects, because it is easier to control the execution inside an active object. On the other hand, the composition of active objects leads to nested virtual memory structures. All these considerations led us to adopt the passive object model.

2.3.2 Object Invocation

A basic operation in the system is *object invocation*. An invocation specifies the system reference (unique internal name) of the invoked object, the name of a method and the parameters of the invocation. The reference contains a unique object identifier, called *oid,* and location hints. If the object is not currently loaded in Virtual Memory on the node where the execution takes place, an *object fault* occurs. The object is located in secondary storage using the reference. A node is selected for the execution and the object is loaded on that node. The method is then invoked like a traditional procedure call. The selection of the execution node is described at the end of this section.

Object invocation is supported by two operations essential to the management of the object memory: *loading* and *linking*. Loading retrieves an object from Secondary Storage and creates an object image in the Virtual Object Memory on the execution node. Linking adds it to the job context and maps it into the virtual memory. The operations are executed dynamically on object invocation. An invocation may take place locally or on a remote node. In both cases, it is synchronous, i.e. the activity is blocked waiting for results.

A *remote invocation* takes place if the node selected for execution is different from the node on which the object fault occurred. In this case, the job and the activity *diffuse* to the remote node and the object is invoked there.

The choice of the execution node is determined by an execution policy which is separate from the basic kernel mechanisms. Ideally, such a policy should be based on system information about the availability of resources and about performance indices such as processor load or response time, in order to do *adaptive load balancing*. Load balancing is not implemented in the current version of Guide. The default execution policy is as follows: if the invoked object is already loaded in virtual memory on a remote node, the execution takes place on that node; if not, the object is loaded on the node where the object fault occurred. An object may also be specified as *unmovable,* in which case it always remains on its creation node.

2.4 Parallelism and Synchronization

The execution of concurrent activities within a job is controlled by the following construct (concurrent clause):

```
COBEGIN
  label_1: <invocation of object1>
   . . .
  label_n: <invocation of objectn>
COEND <optional condition>
```

Each object invocation enclosed between **COBEGIN** and **COEND** is executed as a separate activity. All these activities share the context of the current job. The calling activity (i.e. the activity in which the concurrent clause is called) is suspended. Its resumption is controlled by the termination condition, which has the form of a boolean expression using the labels of the concurrent invocations. Each of these labels is interpreted as a boolean variable that specifies whether the corresponding invocation is terminated. By default, if no termination condition is specified, the calling activity is resumed when all concurrent invocations are terminated.

Activities communicate through shared objects. Object sharing must be controlled in order to ensure that the shared data remain in a consistent state. Our basic choice is to express synchronization as a set of constraints associated with objects, not as primitives appearing within activities. This is fully consistent with the object approach, since the specification of the synchronization constraints is concentrated in the class that describes the object instead of being spread out in a number of methods that use the object. In addition, this synchronization specification is shared by all instances of the class.

The only way for an activity to access or modify an object is to execute a method of this object. Therefore, we specify synchronization as a set of activation conditions, which form a control clause. Each activation condition is attached to a method and must be satisfied before the execution of this method may start. If no activation condition is attached to a method, then the execution of this method is unconstrained. The syntax of the control clause is as follows:

```
CONTROL [<method name> : <activation condition>]*
```

where <activation condition> is a boolean expression which may contain the following parameters: instance variables which represent the internal state of an instance; actual parameters of the method; and synchronization counters. These counters are internal variables which specify, for each method of a given object, the total number of invocations, the total number of completed executions, the current number of pending invocations, etc. These counters are automatically updated by the system. Synchronization based on counters was introduced in another context by Robert and Verjus [1977].

Synchronization constraints are illustrated by the following class definition, which implements a bounded buffer used for communication between activities. The types ProducerConsumer and Item are supposed to have been introduced elsewhere:

```
CLASS FixedSizeBuffer IMPLEMENTS ProducerConsumer IS
        CONST size=<a constant>;
        buffer : ARRAY[0..size-1] OF Item;
        first, last: Integer = 0, 0:
    METHOD Put (IN m: Item);
        BEGIN
        buffer[last]:=m;
        last:=last+1 MOD size;
        END Put;
    METHOD Get (OUT m: Item);
        BEGIN
        m:=buffer[first];
        first:= first +1 MOD size;
        END Get;
    CONTROL
        Put: (completed(Put) - completed (Get)<size)
                    AND current (Put) =0;
        Get: (completed (Put) > completed (Get))
                    AND current (Get) =0;
END FixedSizeBuffer.
```

In this example, *completed(m)* and *current(m)* are counters that record, respectively, the number of completed executions and currently active invocations of a method *m*.

It should be noted that activation conditions are expressed using only boolean expressions. As a consequence, some synchronization schemes cannot be directly expressed. This limitation may be overcome by introducing additional methods. The implementation of this synchronization scheme is described in Decouchant et al. [1986].

2.5 Transactions

Transactions are defined as consistent units of computation. In the current well accepted model, transactions are characterized by four properties: failure atomicity, failure isolation, permanence and serializability. To ensure consistency and safety, a transaction in Guide must start and end within the same object method. A transaction is entirely contained within a single activity and within a single job. In the current version, transactions are limited to a single node. Multi-node transactions are envisaged for a future version.

Objects are divided into two categories: atomic and non-atomic; consistency and permanence properties are only guaranteed for atomic objects. The system requires all updates of atomic objects to be done within a transaction. The *atomic* property of an object is fixed at object creation and cannot be modified afterwards.

2.6 Interface of the Virtual Machine

The Guide virtual machine provides a set of primitives supporting programs written in the Guide language. The main primitives are related to the management of jobs, activities, objects and transactions. They are summarized in Table 1. Some primitives related to the I/O are not specified here. Clearly, the virtual machine has two interfaces: the low-level interface of the Secondary Storage and the high-level interface of execution structures, object invocation and transactions. The interface of the Secondary Storage is not visible at the higher level.

3. Implementation of Guide

The implementation of Guide on top of UNIX is described in this section. We first present the implementation of jobs and activities as UNIX processes, then the organization of the Virtual Object Memory and the mechanisms for local and remote object invocation; afterwards, the communication protocol and the organization of the Secondary Storage.

	Jobs and activities
CreateJob	Create a job
KillJob	Kill a job
StartJob	Start an initial activity of a job
SuspendJob	Suspend a job
ResumeJob	Resume a job
JobStatus	Returns job status
JobRef	Returns job reference
NewActivity	Create and start an activity
KillActivity	Kill an activity
StartActivity	Start an initial activity of a job
SuspendActivity	Suspend an activity
ResumeActivity	Resume an activity
ActivityWait	Wait for parallel activities
ActivityStatus	Returns activity status
ActivityRef	Returns activity reference
	Objects
ObjectCall	Call a method on an object
ObjectCreate	Create an object
ObjectInfo	Get information about an object
	Transactions
BeginTrans	Begin of a transaction
CommitTrans	Transaction commitment
AbortTrans	Transaction abortion

Table 1: Primitives of the Guide virtual machine

3.1 Job and Activity Management

3.1.1 Principles

The Guide system provides two execution structures: jobs and activities, which can be implemented in two ways. The first way is to use one UNIX process to represent a job, activities running as lightweight processes managed internally inside the job. Their addressing space is the virtual memory of the UNIX process. This solution requires the implementation of a lightweight process kernel inside a UNIX process. The second way is to implement an activity as one UNIX process. A collection of processes implementing activities represents a job. The processes must be able to map objects at the same addresses in their virtual memory and job management should be done by a separate process. We chose the second approach because it minimizes the implementation effort by directly using UNIX process management and makes debugging easier. Also, we wanted to share the processor time equally among activities and not among jobs. Moreover, the common virtual memory of activities can be easily implemented using the shared memory facility of UNIX System V.

The management of jobs and activities requires cooperation of Guide systems running on different nodes, because jobs and activities are distributed execution structures and have system-wide meaning. One UNIX process per node is in charge of this cooperation and of the management of local jobs and activities. The process is called a Guide daemon. It maintains data structures for job and activity management and serves queries requesting operations on jobs and activities.

An activity communicates with the Guide daemon by queries to request some operations on jobs and activities (creation, destruction, suspension, resumption and diffusion). The processing of a query may involve daemons on other nodes as well. Communication between the daemon and activities is based on two different mechanisms provided by UNIX 4.2BSD and UNIX System V, namely *sockets* and *message queues*. The daemon sends messages to local activities via message queues. The daemon may receive messages from local activities as well as from remote daemons. Since remote communication involves sockets and the daemon waits for messages using the *select* primitive, the same mechanism must be used by the daemon for local communication. Thus, activities send messages to the daemon via the socket mechanism. Figure 2 presents the structure of the UNIX implementation.

The Guide daemon maintains a data structure called a Node Object Table (NOT) representing the Virtual Object Memory of the node. It contains the information about all objects currently linked on the node. The Guide daemon also maintains a data structure for each job representing its context. The data structure is called a Context Object Table (COT). It contains the information about the objects mapped into the addressing space of a job. Since these tables are shared by all activities of a job, they are implemented as segments of shared memory mapped into the addressing space of the activities.

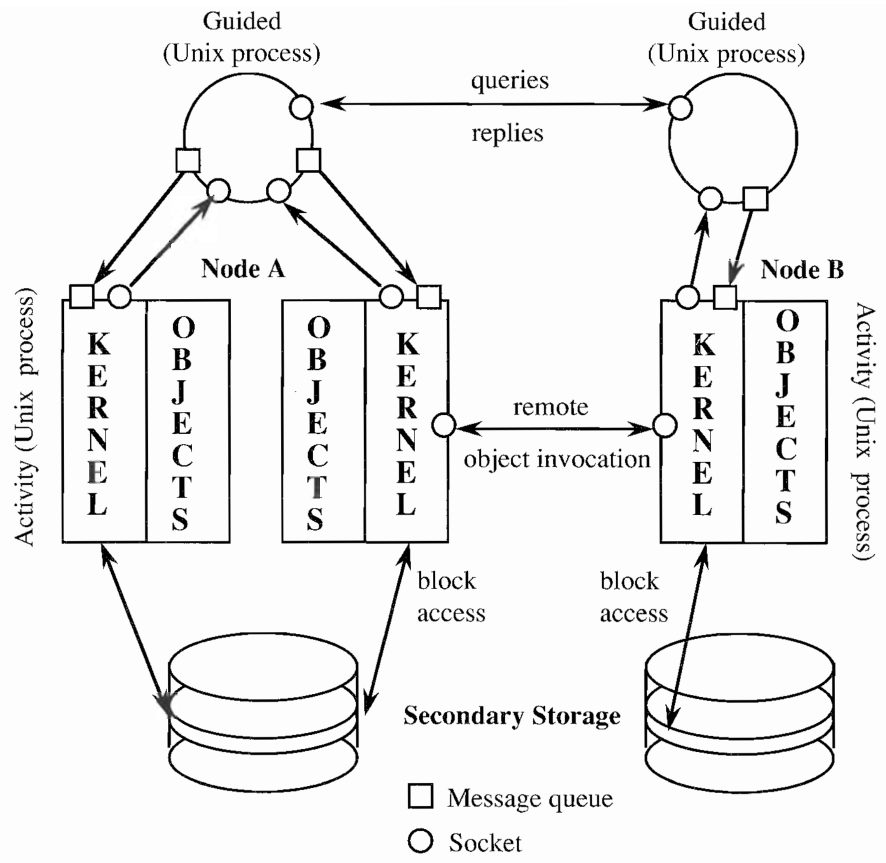

Figure 2. Structure of the UNIX implementation

3.1.2 Internal Synchronization Primitives

Several data structures in the Guide kernel are shared among several UNIX processes. The access to these data must be synchronized. Synchronized objects and the implementation of **COBEGIN** – **COEND** also use synchronization primitives. In the first prototype, we used UNIX semaphores. However, measurements showed that almost 90% of object invocation execution time was spent in non-blocking semaphore operations. Therefore, we developed a UNIX driver implementing the *P* and *V* operations on a semaphore. In addition, non-blocking *P* operation and *V* operation with an empty process queue (the most common case) are implemented as assembly language routines incrementing or decrementing a counter atomically. If a *P* operation needs to block a process or if a *V* operation must wake up another process, a corresponding operation on the driver is executed. The synchronization primitives are about 100 times faster than the UNIX System V semaphore primitives. For more details see Decouchant et al. [1989].

3.1.3 Job Management

Operations on jobs are executed by the Guide daemon. In addition to the information in the NOT and COT, it uses other tables containing: the internal name of a job (the system reference of the object representing a job), the UNIX identifier of the shared memory segment containing the COT of a job, the number of job diffusions, the address of each diffusion node, the address of the job creation node (called the *root* of the job). The diffusion of a job is a result of an object invocation on a remote node not previously visited by the job. The information about the job diffusion is updated after the first diffusion to a given node and is used later to reply to queries requesting diffusions to the node. The address of the job creation node is necessary for the destruction of a job. It is done recursively: the **KillJob** query is sent to the daemon residing on the root of the job, it is then transferred to all nodes where the job has diffused and the job management information is updated.

3.1.4 Activity Management

The management of activities has the following aspects: creation of the main activity of a job, management of processes representing an activity, and management of activities created by the **COBEGIN** – **COEND** primitives. When a job is started, the main activity represented by a UNIX process is created and started using the **NewActivity** primitive. If an object invocation takes place on a remote node, the invoking activity and its job must diffuse to that node. A UNIX process representing the invoking activity and serving remote object invocations is created on the remote node. Only one such process for a given activity exists on a node. The Guide kernel maintains data structures concerning remote processes and their communication ports for each activity.

The **COBEGIN** primitive creates parallel subactivities. The activity executing **COBEGIN** is blocked on a semaphore and waits until the condition associated with the **COEND** primitive is satisfied. Then any remaining activities are killed and the parent activity resumes its execution.

3.2 Virtual Object Memory

The Virtual Object Memory is implemented by the virtual memories of the UNIX process that implement all activities currently being executed on the node. The activities of a job share the Guide kernel code, the objects linked to the job and some management data structures: COT, NOT and other information concerning the job and its activities. Sharing is achieved via shared memory segments.

3.2.1 Structure of the Virtual Memory

The virtual memory of the UNIX process representing an activity is composed of three parts: a part shared with other activities, a private part owned by the activity, and a part reserved for the stack. The shared part contains the Guide kernel (mapped into the UNIX *.text* segment), the NOT, the COT and the object memory used to handle the objects currently linked on the node. The private part of the virtual memory contains Guide kernel data and the heap of the activity. The shared segments of the NOT, the COT, and object memory as well as the private part reside in the UNIX *.data* segment. The activity stack corresponds to segment *.stack* of the UNIX process.

Each shared memory segment has a key used by the UNIX *attach* and *detach* operations. The management of the key is done by the kernel. The list of current keys is a part of the data contained in the NOT. When an activity invokes a method on an object not yet loaded into the virtual object memory, it requests a region in the shared object memory. Storage allocation is done by the kernel using a generalized buddy algorithm. Then, the object is loaded into the segment. Other activities of the same job which need to access the object only have to link it to their context. The objects not linked to any job remain in the object memory and are asynchronously transferred to the Secondary Storage. Figure 3 presents the structure of the virtual object memory on a node.

In this example, two jobs (Job_1 and Job_2) are currently active on the node. Job_1 has two activities, while Job_2 has only one. The figure represents the organization of the virtual memory of the UNIX processes which implement the three activities and shows the mapping

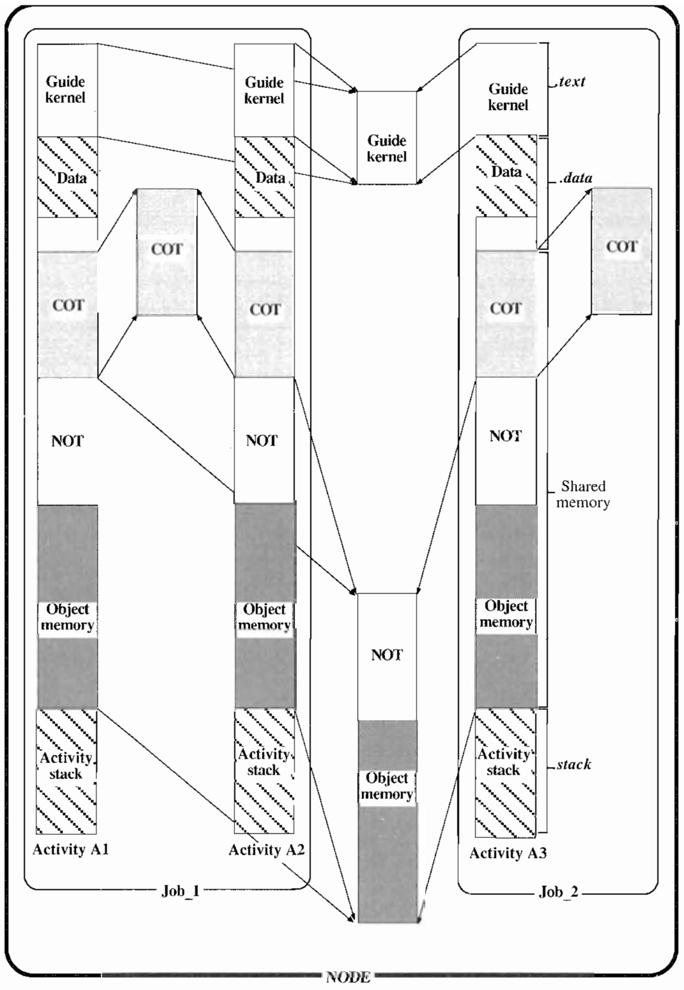

Figure 3: Virtual Object Memory of a node

of memory regions on segments of shared memory. The Context Object Table (COT) of a job is shared by all activities of that job; the Node Object Table and the object memory itself are shared by all activities on the node. The code of the kernel is also shared by all activities.

3.2.2 Object Invocation

The invocation is done by a system primitive **ObjectCall.** It receives a *parameter block* containing the parameters, the reference to the class of the invoking method, the identifier of the method and the reference to the invoked object. If the object is not loaded in virtual memory on any node, the invocation may be local or remote independently of its current location in secondary storage. If the object is already loaded and linked to a job context on a node, the invocation takes place on that node. Otherwise, as mentioned in section 2.3.2, the default policy in the current version of Guide is to choose the node on which the object was created.

a) Local object invocation

Consider the case when a method is to be executed locally. The Guide kernel performs the following operations.

1. The current context of the UNIX process representing an activity is pushed onto the stack.
2. If the object is not in the job context, the Virtual Object Memory on the local node is looked up. If the image of the object does not exist on the node, the class and the object (class instance) are loaded using the primitive **LoadObject.** The class is loaded from the local Secondary Storage. If the object is stored on a remote node, it is located and transferred.
3. The class and the object are linked to the job context. This operation may be simple (a link counter is incremented) if the class and the object are already present on the node, or it may require updating the COT and NOT tables. Because of inheritance, several classes may need to be loaded until the invoked method is found.
4. A new context is constructed: the addresses of the method and the object are calculated using the data prepared by the Guide compiler and the parameter addresses are pushed onto the stack.
5. The method invocation is done as a standard procedure call.
6. After the return from the method, the object and its class are unlinked.
7. The context of the UNIX process representing the activity is pulled from the stack.
8. The control returns to the invoking method.

b) Remote object invocation

The following operations are performed during the remote object invocation.

1. The kernel checks whether the invoking activity and its job have already diffused to the remote node. If not, their diffusions are

performed: the extension of the job and the process representing the invoking activity are created on the remote node; the process creates a socket used for all subsequent communications; the local daemon is notified and the address of the remote process is given to the invoking activity.

2. The parameter block is prepared before the transfer to the remote node: all parameters passed by address are replaced by their values. The data is translated into a machine independent format if machines are of different types. In the first implementation, the XDR format is used.
3. The parameter block is sent to the UNIX process which represents the activity on the remote node.
4. The receiving process performs the operation inverse to the preparation: the data are translated from the machine independent format and the parameters passed by address are copied into temporary variables.
5. The process performs a local **ObjectCall** passing the parameter block.
6. After the execution of the local **ObjectCall,** the results passed by address in the parameter block are prepared and their format is translated.
7. The parameter block is sent back to the kernel at the origin node and the control is returned to the invoking activity.

This scheme of execution is similar to the standard Remote Procedure Call, as described e.g. in Birrell & Nelson [1984]. Nevertheless, there are two important differences. First, the binding is dynamic: the reference to the called object and the identifier of the method are passed in the parameter block and the method is linked dynamically just before execution. Efficiency is improved by using a method cache. Second, *nested back-calls* between two nodes are possible and are conveniently handled by the UNIX processes which implement activities. A nested back-call refers to a situation in which an activity executing on a node *N* performs a remote invocation which eventually causes an invocation to an object located on *N*, before the initial invocation has returned. This is illustrated in Figure 4.

The process which implements the invoking activity is blocked in the **ObjectCall** on the initial node waiting for the results of the invocation. Thus, it can accept the remote invocation, execute it and return results. Then, the process on the remote node resumes its execution,

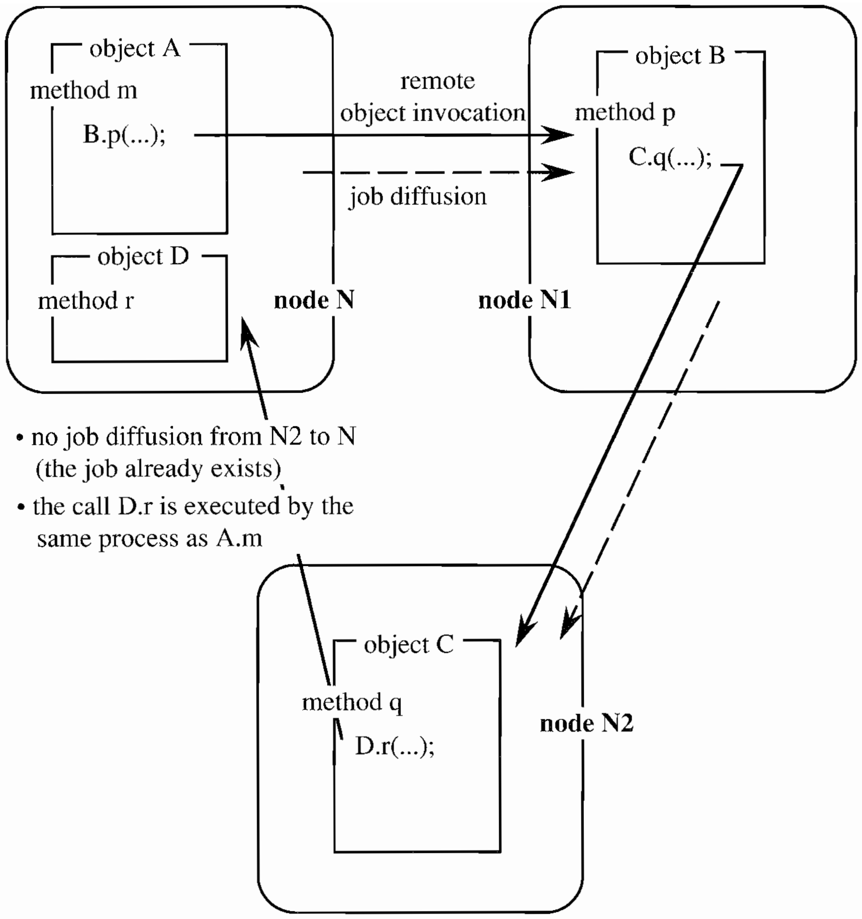

Figure 4: Remote object invocations and activity diffusion

terminates the method and transfers the results back to the invoking activity.

3.3 Communication

Different communication primitives are needed in the Guide implementation. First, a blocking request-reply message exchange protocol is necessary for the remote object invocation. The protocol should be able to send a request message to a destination and wait for a reply arriving from any destination (possibly different from the receiver of the request message). It is used by an activity and its remote process during object invocation. Each process has only one pending message and expects to receive only one message. This semantics simplifies the protocol and the message transfer delay can be optimized. Second, the management functions of Guide daemons require a non-blocking request-reply message exchange protocol. Each Guide daemon acts as a server providing job and activity management functions to other daemons. This implies that a daemon may receive several messages and should treat them asynchronously. Communication between daemons is infrequent, so the execution performance does not strongly depend on the performance of the communication. Third, a non-blocking message transfer protocol with high throughput is needed. It is used by the secondary storage daemons for object transfers and migration. The throughput of this protocol is optimized by overlapping transfers with disk operations. These three types of communication are supported by a connectionless specialized protocol built on top of UDP/IP and the socket layer.

Operations requiring communication could have been implemented using standard protocols, e.g. TCP/IP, but there are several reasons for using a specialized protocol. For instance, the remote invocation has a request-reply nature. Thus, if there are no transmission errors, using a connectionless protocol minimizes the number of packets sent: the reply is at the same time the acknowledgement of the request. As a new request is the acknowledgement of the last reply, only two transfers are necessary for one invocation in the best case. A specialized, request-reply oriented protocol can be built on top of a lightweight, rudimentary and unreliable protocol. The end-to-end argument justifies this approach minimizing the time necessary for an invocation. Another issue is the limitation that UNIX imposes on the number of sockets simultaneously connected to a process: they cannot exceed 20 or 30 depending on the UNIX implementation. This is insufficient for our system, where a process representing an activity may potentially communicate with remote processes on any node of the system.

Our protocol provides a reliable message transport service similar to the Birrell-Nelson packet level transport protocol designed for RPC. The main difference is the way of recovering from transmission errors (selective retransmission) and the possibility of supporting nested back-calls. The protocol handles the retransmission of lost messages and the suppression of duplicated messages. The standard Internet addressing scheme is used and the following function is provided: sending a message to a remote node and receiving a message from any node (not necessarily from the node where the message was sent). The protocol provides a blocking primitive **sendRecv** optimized to be used for the remote object invocation; there only being one pending invocation per activity. Servers can be implemented using asynchronous primitives **sendTo** and **recvFrom.**

Two timeout values are associated with each message. If the message is not received before the first timeout, it is retransmitted. If the processing of a received message takes more than the second timeout, a separate acknowledgement is sent back. A message specifies the type of a requested service (e.g. remote object invocation, object migration, beginning of a transaction). The reception of a message is acknowledged by a reply message or by a separate acknowledgement if the service lasts too long. The protocol is implemented as a library of procedures linked to the Guide kernel code.

3.4 Secondary Storage

Our first implementation of the secondary storage provides basic primitives for supporting persistent objects in a distributed environment. Replicated physical containers are not provided in the current implementation. Each logical container is mapped on one physical container implemented using disk partitions accessed by the UNIX kernel in raw mode; a disk access is a synchronous operation that bypasses the UNIX buffer cache. Disk operations are optimized using an internal cache. There is a single cache on each node on which at least one container is supported and this cache is common to all containers located on this node.

Figure 5 presents the structure of the Secondary Storage subsystem. It has four layers: object management, block management, container management and cache. The object management layer provides the interface to the Secondary Storage subsystem. It checks if the object is stored on the local node and requests remote operations if necessary. The block management layer supports the mapping between disk blocks and objects. A disk partition provides the physical support for storage and is managed by the container layer. The cache module, implemented as a region of shared memory, is similar to the UNIX

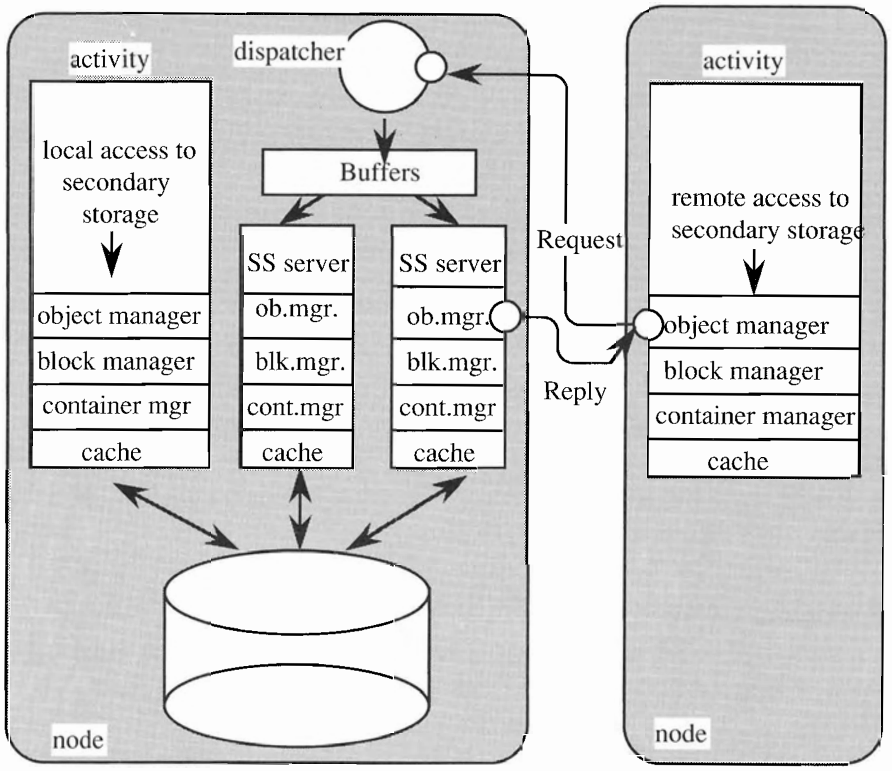

Figure 5: Organization of the Secondary Storage subsystem

buffer cache and thus makes disk accesses more efficient. Modified blocks in the cache are stored asynchronously by a dumping process. All layers of the Secondary Storage subsystem are linked to the UNIX processes which implement activities. The primitives are summarized in Table 2.

	Secondary Storage
CreateObject	Create an object
DeleteObject	Delete an object
GetDesc	Get an object descriptor; lock the object
LoadObject	Load an object into VOM
StoreObject	Store an object
ReleaseObject	Unlock an object

Table 2: Primitives of the Secondary Storage System

Objects are loaded on demand. If an object is present on the local node, its blocks are loaded from the disk via the cache. The Secondary Storage subsystem guarantees the uniqueness of the image of loaded objects. If an object is already loaded on a remote node, the Secondary Storage subsystem attempts to fetch the object from that node. Job and activity diffusion are used for all remote operations: remote requests to Secondary Storage are sent to the UNIX process which represents the invoking activity on a remote node; this process acts as a local secondary storage server. The process receives request, executes an appropriate local operation and sends a reply to the requesting node. If the object is not linked to any job, the Secondary Storage subsystem of the remote node returns the permission to fetch the object. If the object is linked on the remote node, the identifier of the node is returned; it will be used for subsequent remote object invocations.

Physical containers are organized in 512 bytes blocks. An object descriptor is stored in one block. Since the size of the descriptor is less than the block size, the remaining space in the block containing the descriptor may be used to store the corresponding object data. If the object size extends beyond this space, its contents is stored in independent data blocks using a mechanism similar to that of the UNIX file system. The resulting maximum object size is 8,287 Kbytes.

An object is considered garbage when there is no way to access it from the root of the object graph, formed by inter-object references. Since the lifetime of an object is usually independent of the execution structures lifetime, garbage collection is needed. Our current garbage collection algorithm uses the mark and sweep method.

4. Evaluation and Experience

The first prototype of Guide has allowed us to assess both the architecture of the system and the object model, as well as to evaluate the implementation techniques.

4.1 Evaluation of the Model

Several distributed applications have already been developed. They include a mail service, a distributed diary, and a simple browser for types; we also have modified a document editor to allow multiple authors to work on shared documents. Some useful conclusions result from this experience. Guide execution structures seem to be well suited for developing complex applications in a distributed environment. They offer the programmer a simple way to exploit the parallelism of applications. Synchronization structures are powerful enough to deal with programmer requirements. Persistent objects free the programmer from the burden of explicitly saving and restoring files. Indeed, the notion of a file system disappears, since it is subsumed in the object management system. Another useful feature of the system is location transparency. The location of the objects is invisible to the programmer and is controlled by the system. At the language level, there is no distinction between local and remote method invocations. As a consequence, it is easy to develop and debug an application on a single node and to distribute it over the network without change. The mail application was developed in this way. However, explicit control of the object location may still be applied if needed. For instance, server applications can be developed to provide a service on a particular node.

4.2 Supporting Object-Oriented Systems on UNIX

Implementating Guide on top of UNIX has allowed us to rapidly provide a portable version of the system capable of supporting simple applications. Some conclusions as to the suitability of UNIX to implement object-oriented systems can be drawn from this experience.

The Virtual Object Memory on a node is implemented using a single shared memory segment for all objects. The segment is mapped at the same address in all jobs. Regions in the segment are allocated to objects by the Guide kernel. Variable size objects can be handled in an efficient way and there is no memory fragmentation. However, since all objects are visible to all jobs, there is no memory protection between jobs. A preliminary version of Guide used another approach: each object was mapped into one separate shared memory segment. However, the management of shared memory in UNIX System V has some limitations, e.g. only the process that has attached a shared segment can detach it. Sometimes, a segment should be detached by another process. Also, the attach and detach operations on shared memory segments are inefficient and introduce a considerable overhead. As these operations are used almost each time an object is linked or unlinked, the performance of object invocation was strongly affected. Furthermore, the size of shared memory segments is fixed at their creation and cannot be changed. This feature is not useful for implementing variable size objects. Moreover, the minimal size of a segment on DPX 1000 and DPX 2000 is 8Kbytes, resulting in memory fragmentation for small objects. This approach has been abandoned, and we use a single shared memory segment per node; as a consequence, the protection provided by separate segments is lost.

As was discussed in Section 3.3, UNIX System V synchronization mechanisms are not suitable for implementing systems like Guide. Synchronization is extensively used in our implementation, and the performance of the Guide virtual machine primitives strongly depends on the synchronization mechanism. We developed efficient synchronization primitives based on a semaphore driver and atomic increment and decrement routines written in assembly language. The performance of object invocation was increased by a factor of 20 after replacing the standard UNIX synchronization mechanisms.

The Secondary Storage uses the raw disk access scheme. The disk space is managed internally by the Guide kernel; the standard caching mechanism of UNIX is therefore bypassed.

The issues related to communication were discussed in 3.3. The standard UNIX mechanisms (sockets and message queues) were used for internal communication in the kernel, but we developed our own communication protocol to support object invocation. This was motivated both by the limitation on the munber of sockets per process and by efficiency considerations.

As it can be seen from this discussion, UNIX imposes strong limitations on the implementation of systems such as Guide. In almost all parts of the kernel, standard UNIX mechanisms had to be bypassed, modified or used with care. However, the following UNIX features were useful: development environment, memory management, process management, I/O management, sockets and message queues. Another positive point is the portability resulting from using UNIX as a base for the development. Guide can be ported to a new machine in a few days. The main work involves modifying the synchronization driver and rewriting the loader.

4.3 Performance

We expected that the gain in development time and portability which resulted from the use of UNIX would be paid in performance. System calls incur an important overhead which adds to the Guide kernel overhead. However, we tried to optimize the use of Unix system calls to improve performance. Some measurements are presented in this section to show the performance of the principal operations.

We measured the following operations at the language level: local invocation of `Self` object, local invocation of another object and remote object invocation. The measurements have been done on Sun 3-60 workstations connected by a 10MBit/s Ethernet. The operations were repeated a large number of times and the elapsed time of the loop was divided into the number of iterations to obtain the elapsed time of a single operation. Object invocations were done without any parameters and the invoked object executed a null method. For comparison, the elapsed time of a procedure call within an object is given. Table 3 summarizes the performance figures. As it can be seen from this table, the operations have an acceptable performance. However, comparable systems directly implemented on bare hardware clearly perform better [Schroeder & Burrows 1990]. Efficiency can be improved by increasing the granularity of objects within an application. Further experiments and measurements will be done to evaluate the overhead of using UNIX and will serve as a basis for an improved design.

Operation	time (ms)
null procedure call	0.011
null local **ObjectCall** of *Self*	0.294
null local **ObjectCall**	0.387
Null remote **ObjectCall**	6.960

Table 3: Performance of some operations

5. Conclusions

The paper has presented the architecture and the implementation of the object-oriented distributed operating system Guide. The system provides an execution environment for an object-oriented programming language. The main features of the system are: persistent objects supported by a distributed two-level storage, transparent distribution of objects, execution model based on concurrent, distributed jobs and activities, support for synchronization and transactions.

A first implementation has been done on top of UNIX for fast prototyping. Using UNIX avoided the need for implementation of low-level machine-specific and device-specific software. We had to overcome some limitations imposed by UNIX and to tune the system to obtain satisfactory performances. Our first experience shows that the architecture of Guide is well suited for supporting a language with object persistence, concurrent activities, synchronization and transactions.

We summarize the main conclusions that we have drawn from this first experience.

- The use of a high level language tightly coupled with the underlying system provides an effective development tool for programming distributed applications. The main mechanism that the run-time system has to support is remote method invocation with dynamic binding.
- Location transparency proved to be an essential feature. Applications developed on a single node were ported to a distributed environment with virtually no additional work. Object location may still be controlled by the user if necessary.
- Using persistent objects greatly simplifies storage management for the user. However, the system bears the load of garbage collection, which involves a performance penalty. Mechanisms related to the expected lifetime of objects (e.g. generation scavenging) could be a way to reduce this penalty.
- Dynamic binding and resolution of object references are well-known sources of performance overhead in persistent object systems. The situation can be improved by the use of clustering mechanisms (to transfer a group of objects as a whole) and partial static binding.

In the next phase of the project, we plan to investigate the above mentioned proposals for performance improvement and to develop a new implementation of the system on top of a low-level kernel, which we expect to provide process and memory management primitives better suited than those of UNIX to support a distributed object system. The selection of this kernel is under way.

6. *Acknowledgements*

We wish to acknowledge the contribution of the COMANDOS project object-oriented working group: especially our partners from INESC (Lisbon, Portugal) and Trinity College (Dublin, Ireland).

In addition to the funding of the Commission of European Communities through Esprit, the Guide project is supported by the Universities of Grenoble (Institut National Polytechnique de Grenoble - Université Joseph Fourier) and Centre National de la Recherche Scientifique. Initial support was provided by Centre National d'Etudes des Télécommunications.

References

Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D., "The Eden System: A Technical Review," *IEEE Trans. Software Engineering,* vol. SE-11, no. 1, pp. 43–59, January 1985.

Alves Marques, J., Balter, R., Cahill, V., Guedes, P., Harris, N., Horn, C., Krakowiak, S., Kramer, A., Slattery, J., and Vandôme, G., "Implementing the Comandos Architecture," *Proc. 5th Annual Esprit Conference,* pp. 1140–1157, Brussels, November 1988.

Birrell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems,* vol. 2, no. 1, February 1984.

Black, A. P., Hutchinson, N., Jul, E., Levy, H., and Carter, L., "Distribution and Abstract Types in Emerald," *IEEE Trans. on Software Engineering,* vol. SE-13, no. 1, pp. 65–76, 1987.

Decouchant, D., Duda, A., Freyssinet, A., Paire, E., Riveill, M., Rousset de Pina, X., and Vandôme, G., "GUIDE: an Implementation of the COMANDOS Architecture on Unix," *Proc. EUUG Autumn Conf.,* pp. 181–193, Lisbon, October 1988.

Decouchant, D., Krakowiak, S., Meysembourg, M., Riveill, M., and Rousset de Pina, X., "A Synchronization Mechanism for Typed Objects in a Distributed System," *SIGPLAN Notices,* vol. 24, no. 4, April 1988.

Decouchant, D., Paire, E., and Riveill, M., "Efficient Implementation of Low-level Synchronization Primitives in the Unix-based Guide Kernel," *Proc. EUUG Conf.,* pp. 283–294, Vienna, October 1989.

Jones, M. B. and Rashid, R. F., "Mach and MatchMaker: Kernel and Language Support for Object-Oriented Distributed Systems," *Proc. OOPSLA '86,* pp. 67–77, Portland, 1986.

Krakowiak, S., Meysembourg, M., Nguyen Van, H., Riveill, M., Roisin, C., and Rousset de Pina, X., "Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications," *Journal of Object-Oriented Programming,* vol. 3, no. 3, pp. 11–22, September–October 1990.

Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., and Stumpf, B. L., "The Architecture of an Integrated Local Network," *IEEE J. Selected Areas in Comm.,* pp. 842–856, 1983.

Leblanc, R. J. and Appelbe, W. F., "The Clouds Distributed Operating System," *Proc. 8th Int. Conf. on Distributed Computing Systems,* pp. 2–9, San Jose, Calif., June 1988.

Liskov, B. H., "The Argus Language and System," in *Distributed Systems: Methods and Tools for Specifications,* pp. 343–430, Lecture Notes in Computer Science no. 190, Springer-Verlag, 1985.

Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H., "Amoeba: A Distributed Operating System for the 1990s," *IEEE Computer,* vol. 23, no. 5, pp. 44–53, May 1990.

Robert, P. and Verjus, J., "Toward Autonomous Description of Synchronization Modules," *Proc. IFIP Congress,* pp. 981–986, North-Holland, 1977.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Léonard, P., and Neuhauser, W., "The Chorus Distributed Operating System," *Computing Systems,* vol. 1, no. 4, pp. 305–370, December 1988.

Schroeder, M. and Burrows, A., "The performance of the Firefly RPC," *ACM Trans. on Computing Systems,* vol. 7, no. 1, February 1990.

Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C., "SOS: An Object-oriented Operating System - Assessment and Perspectives," *Computing Systems,* vol. 2, no. 4, pp. 287–338, December 1989.

Thatte, S. M., "Persistent Memory: A Storage Architecture for Object-Oriented Database Systems," *Proc. Int. Workshop on Object-Oriented Database Systems,* pp. 148-159, Asilomar, 1986.

[submitted Oct. 17, 1990; revised Nov. 30, 1990; accepted Jan. 24, 1991]