Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus

F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont

Unité Mixte Bull-IMAG/Systèmes, 2, avenue de Vignate, 38610 Gières, France - Internet: guide@imag.fr - Phone: +33 7651 7879

Abstract:
This paper describes our experience in the implementation of the Guide object-oriented distributed system on Unix, Mach and Chorus. A full version of Guide has been developed on Unix. While a full micro-kernel based version of Guide does not yet exist, the basic mechanisms for the support of the Guide virtual object memory and computational model have been implemented on Mach and Chorus. The paper reports the preliminary results of this experience and provides an evaluation of the adequation of micro-kernel technology, as compared to Unix, for the support of object-oriented distributed systems. Microkernels provide a great modularity and flexibility for the conception of an operating system with the server architecture. Execution of lightweight activities, efficient communication and distributed shared virtual memory are very useful fonctionalities for managing the distribution.
Keywords: Object-oriented distributed systems, Mach, Chorus, microkernels

1 Introduction

This paper describes our experience in implementing the object-oriented distributed operating system Guide (Grenoble Universities Integrated Distributed Environment) on top of two micro-kernels, Chorus and Mach. Based on this experience, we compare these two implementations with a previous one developed on top of the Unix system [1] [2].

The aim of the Guide project is to explore distributed computing, structured in terms of objects and based on a set of heterogeneous workstations interconnected via a local area network. The system is targeted towards cooperative applications such as document processing and program developement. Therefore, *object sharing* is an important feature of our model : objects are persistent and are the principal way of communication between concurrent activities. The project aims to define an architecture which hides most of the problems inherent to distribution, for storage, execution and resource allocation, thus providing location, access and execution transparency. The object-oriented model allows us to integrate several concepts related to operating systems, programming languages and databases, in a single approach.

Guide is a component of Comandos (Construction and Management of Distributed Open Systems), a project supported by the Commission of European Communities under the ESPRIT program.

The first Guide prototype was implemented on top of the Unix system. This implementation aimed to provide rapidly a version of the system capable of supporting simple distributed applications based on the object model. This goal has been achieved and the current version of Guide supports several experimental applications such as document editing, mailing and distributed diary.

While the Unix implementation of Guide provides an adequate testbed to investigate the developement of distributed applications, it suffers from several limitations, both in functionality and in performance. More precisely, we identified a lack of adequate support for the following critical aspects:

- shared objects,
- synchronisation mechanisms,
- lightweight activities,
- communications.

In the recent years, a new organization has emerged for operating systems, in which a "micro-kernel" provides the basic functions of memory management, process scheduling and communication, whereas more elaborate services are implemented as specialized servers. Two representatives of this new organization are Chorus [3] [4] and Mach [5] [6]. We expect that implementing the Guide object-oriented architecture on top of one of these kernels would improve over the Unix prototype both in performance and in functionality. In addition, we hope to improve the modularity of the system, and to be able to experiment with different kind of architectures. In this way, a version of Guide has yet been implemented both on top of Mach and Chorus, which is sufficient to evaluate these improvments.

The objective of this paper is to report on this experience. We give a preliminary assessment of the adequacy of the microkernels for the support of an object-oriented distributed system and we compare them, in this respect, with Unix. More precisely, the discussion is focused on the mechanisms that allow to implement a virtual object memory in which shared objects are the main support for communication between concurrent activities or applications.

We are aware of two related recent experiments aimed towards providing an object-oriented distributed environment on top of Chorus and Mach, respectively: the COOL project at Chorus Systèmes [7] and the MachObjects approach at Carnegie Mellon [8]. COOL and MachObjects have a very similar functionality. In both systems, applications are structured in terms of object servers responding to client requests. As opposed to Guide, both systems support an active object model, i.e. objects are the units of sequential execution management. As a consequence, neither COOL nor MachObjects provides a mechanism for object sharing. However, they achieve an equivalent function by allowing memory segments to be shared between objects. No explicit support is provided for persistent objects.

The rest of this paper is organized as follows. Section 2 is an overview of the Guide object-oriented model. Section 3 is a survey and a comparison of Chorus and Mach fonctionality, with special emphasis on the mechanisms for virtual memory support and activity management. Section 4 describes and compares the implementations of Guide on Unix and on both micro-kernels and presents the lessons learned in this experience. Conclusions and perspectives are presented in section 5.

2 Overview of the Guide object-oriented model

This section provides a general description of the architecture of the Guide system, independently of any specific implementation. We first present the object-oriented model which is the basic foundation of the system. We then describe the computational model, which defines the organization and interaction of activities during the execution of an application. We finally present the object storage subsystem, which is in charge of the storage of persistent information.

2.1 The object-oriented model

The choice of an object model for Guide is the most important design decision of the system. Objects not only provide a convenient and powerful means for application structuring; they also allow to unify the concepts of procedural and data abstraction, execution units and long-term storage units.

An object encapsulates data (the state of the object) and operations, also called methods. The data may only be accessed through method invocation. A type describes a common behavior that is shared by all the objects of that type. This behavior is defined by the signatures of the methods. A class defines a specific implementation of a type: it contains the internal description of the representation of the data and the programs of the methods. A class is used to generate instances, i.e. objects whose representation and methods are defined by the class.

Objects are not only the units of applications structuring, but also the support for permanent storage. Objects are said to be *persistent*, i.e. their lifetime is not related to that of the execution unit in which they were created. The functions of a traditional file system are subsumed in the persistent object store.

Within the system, objects are named by low-level, location-independent identifiers called *references*. References are used internally for object invocation; they also provide a way to refer to objects within other objects, thus allowing to build complex, possibly distributed, structures.

The object model is accessible to users through a specific language [9] whose run-time system is implemented on top of the Guide kernel. This language allows to use all the concepts of the Guide object model.

2.2 Execution management

The computational model provided by Guide aims to allow a user to control the concurrent execution of the activities of his application, while preserving location transparency. The user has the vision of a *virtual machine,* which hides the details of distribution, but which provides mechanisms for concurrency control. The system implements this virtual machine by sharing the load between the nodes of the distributed system.

2.2.1 Parallelism and synchronisation

The main abstraction provided by the computational model is called a *job*. A job is defined as a multi-processor, transparently distributed virtual machine, which groups together in a common address space a set of objects and a set of threads of control, also called *activities,* operating upon these objects. Guide objects are passive, in so far as they are dissociated from threads. The composition of the virtual machine can evoluate dynamically. The execution of an activity within a job consists of a sequence of invocations of methods on objects.

To express concurrency, an activity may create at any time a set of child activities which are executed in parallel within the same job, through a `COBEGIN-COEND` construct. The parent activity is suspended until a termination condition is satisfied (e.g. termination of the first child or termination of all children, etc).

2.2.2 Object sharing

The notion of object sharing is central in the Guide model. Shared objects provide the main communication facility between activities, within a single jobs or between different jobs.

The set of objects within a job at a given time is called the *context* of the job. The job context is implicitly shared between all the activities of the job. In addition, objects may be shared between the contexts of two jobs. For example, in Figure 1, the two jobs *J1* and *J2* share the object *d.*

Object sharing must be controlled in order to ensure that the shared data remain in a consistent state. This is achieved by synchronization constraints associated to each method of a shared object. Thus, the data of an object may be accessed according to a specific policy such as readers and writers policy, mutual exclusion or a more complex, application dependent, policy. This synchronisation mechanism is not further described in this paper (see [9] for more details).

2.2.3 Distribution

The virtual machine defined by a job is potentially distributed, i.e. a job may be represented on a set of nodes. Since execution transparency is a major goal of the project, there is no direct link between jobs and nodes. A job can be represented on several nodes and a node can be visited by several jobs. After its creation, a job is only represented in one node, its creation node. Guide provides a *diffusion* mechanism, which allows a job to dynamically extend to several nodes. Thus a job may be viewed as a set of representatives, also called *local jobs,* one on each node on which the job has diffused. The diffusion mechanism is an important feature for the implementation of the execution model, in so far as it allows to exploit the parallelism offered by the network within a job, and allows a great flexibility for distributed resource management.

Figure 2 illustrates job distribution for two jobs, J1 and J2. J1 is entirely represented on node N1, whereas J2 is composed of two local jobs J2/N1 on node N1 and J2/N2 on node N2.

2.3 Object management

We make a distinction between two levels of object management: support for object persistence, which is provided by the secondary storage system (SS) and which includes all objects potentially accessible to jobs, and the virtual object memory (VOM) which includes the objects bound to jobs at a specific time. Figure 3 illustrates this architecture.

The secondary storage system is implemented by the cooperation between the set of nodes provided with a secondary memory reserved for permanent storage. The movement of objects between VOM and SS is automatically managed by the system, according to the policy defined for object loading after an object fault. The default policy is as follows: if the invoked object is already loaded on a remote node, execution takes place on that node; if not, the object is loaded on the node on which the object fault occurred.

To conclude this brief survey of the Guide architecture, we summarize its main relevant aspects: applications are organized as a set of transparently distributed "object spaces" in which concurrent activities may be executed; objects may be shared beween applications, and between activities within an application; objects are potentially persistent and are mapped from a secondary storage system.

3 Survey of micro-kernel fonctionality

The Chorus [4] and Mach [6] kernels supply several similar abstractions wich appear to provide adequate support for an object-oriented model like Guide. Before describing our experience with the implementation of Guide on top of Chorus and Mach, we briefly recall the basic concepts of these two systems.

Section 3.1 presents the concepts common to both systems and section 3.2 describes their main differences. Section 3.3 describes the implementation of a shared memory server, a basic mechanism for the support of shared objects. The common concepts and mechanisms are described using the terminology of Mach.

3.1 Common concepts

The first fundamental abstraction is called a *task* : a task is a set of resources such as memory or communication ports. It can be viewed as a virtual address space within which activities are executed.

The second abstraction is called a *thread* and represents a sequential activity. In first approximation, it consists of a processor state and an execution stack. A thread runs within a task; all resources of a task are shared between its threads. Threads are a basic tool for structuring parallel applications. The traditional concept of process is represented by a single thread running within a task.

A *port* is a communication channel.

A *message* is a unit of communication between threads.

Memory management is very similar in both micro-kernels. Memory is managed outside the kernel by a server called a mapper (in Chorus) or an external pager (in Mach). An external pager provides a set of functions that respond to requests sent by the kernel (i.e. page-fault) or by users tasks (i.e.

allocate memory). The main advantage of this memory management architecture results from its flexibility. Thus, a kernel developer can implement his own design in his own external pager as long as he respects the interface between the kernel and the external pagers. An external pager allows the users tasks to get virtual memory within their virtual address spaces. It also provides different mechanisms for memory sharing between tasks such as sharing by inheritance, sharing by mapping, etc.

We were especially interested in using an elaborate external pager, called a network distributed shared-memory server, to implement our object-oriented model. This is developed in section 3.3.

3.2 Specific concepts

Although Chorus and Mach have been designed with similar goals, their philosophy is not always identical ; differences appear in the following aspects:

- Naming
- Protection and typed messages
- Functional port addressing and port groups

We discuss these differences and we try to evaluate their impact on the design of a distributed operating system.

The first fundamental difference between Chorus and Mach is the naming scheme. Names in Chorus are network-wide global names called Unique Identifiers (UI). A port, a task or a thread is represented by an UI. Because the naming is global, a task which migrates on another node is still accessible by the same UI. Mach objects (i.e. tasks, threads, message queues) are referenced by local identifiers represented by ports. These local identifiers are meaningless outside of their address space (i.e. their task). The knowledge of ports is controlled by the kernel, in order to provide a strong protection mechanism between tasks. The kernel associates port rights with ports. These rights express the semantics of the access a task has to a port, e.g. send right, receive right and ownership right. Thus, ports may only be exchanged through typed messages, which are controlled and analysed by the kernel. This is a strong constraint for the system designers who do not need protection. On the other hand, Chorus allows to exchange names between entities using files, shared memory and messages, since the knowledge of a UI is assumed to guarantee protection. The kernel is not concerned by these exchanges and any additional protection mechanism is left to the designer of the sub-systems. Thus a system which does not need strong protection between tasks (e.g. a real-time subsystem) does not have to pay for it. This contrasts with the Mach point of view that protection is too important to be left to sub-system designers.

Another difference between Chorus and Mach deals with the manipulation of ports. In Chorus, there is the notion of port group which is not present in Mach. A port group allows the user to regroup a set of ports in a single entity. Threads can send messages to the group by broadcast or functional addressing. Groups are very interesting since distributed services are often supplied by a collection of servers distributed over the network rather than by an unique server. Moreover, a port group provides a mechanism to support fault-tolerance : the failure of a server within a group is transparent to clients who continue to address requests to the group. Mach provides a slightly less flexible mechanism for fault-tolerance: when a server fails, its port receiving rights are transferred to a specified backup server which is now responsible for providing the failed service. The other difference is about the notion of set of ports, wich exist in Mach and not in Chorus. A port set allows a thread to block waiting for a message sent to any of several ports. This mechanism may be very useful for protection and authentification management.

3.3 Architecture of a network shared-memory server

A network memory server allows a user to create memory objects, i.e. chunks of memory identified by ports (in Mach) or by capabilities (in Chorus). To address a memory object, a thread maps it within its virtual memory (i.e. the address space of its task). Once an object is mapped, page-faults on this object are treated in the same way as "normal" page faults. The kernel sends a page-fault message to the server to which the object belongs.

A network memory server may be implemented by two main architectures. We define the first one as centralized and the second as cooperative. Both have advantages and shortcomings regarding some specific features.

3.3.1 The centralized architecture

The consistency of an object is managed by an unique server. However, several servers may be active, managing disjoint object sets at the same time. A thread may also access several objects managed by different servers. This architecture is called centralized because all kernels send page-faults on an object to the same server. There is no cooperation between servers to maintain consistency over the network. This architecture is very simple, but it has the main drawback of generating a high network traffic, because all page-fault messages are sent over the network. Figure 4 illustrates this mechanism.

Task T1 and T2 share the object O1. The pager P1 is responsible for preserving the consistency of object O1. Each page-fault on O1 is directly transmitted to P1 through the network. There is no cooperation between pagers.

3.3.2 The cooperative architecture

Consistency is managed by a set of cooperating servers, one per node. This architecture is more complex than the centralized one, but generates less network traffic because page-fault messages are always local (from the kernel to the server of its node). Each server on each node behave like a cache. Thus only page updates and page invalidations are sent over the network between servers. A complete description of such an implementation is given in [10]. Figure 5 illustrates this mechanism.

There is one pager per node. Each task access objects throught its local pager. Task T1 and T2 share the object O1. Each page-fault is transmitted to the local pager of the task node. Pager P1 and P2 cooperate in order to maintain the consistency of object O1.

4 Description of the Unix and micro-kernel Guide implementations

The present section describes the caracteristics of the implementations of Guide and compares the implementation on top of Unix with the implementation as a sub-system of a micro-kernel.

The implementation on top of Unix currently runs on the following workstations: Bull DPX 1000 and DPX 2000 running the SPIX system (System V based with BSD extensions), Sun 3-60, Sun 3-80 and Sun 4 running Sun OS (BSD based with System V extensions), and DEC 3100 DecStations running Ultrix.

The implementations on top on micro-kernels is realize on BM 600 running the Chorus V3.2 and Mach 2.5.

4.1 Implementing execution structures

There are two major problems for implementing the execution structures of Guide on top of Unix. As described in section 2.2, jobs are distributed, i.e. a job may be represented on a set of nodes. The first major problem is to manage this distributed execution. On the other hand, the activities of a job share the object space of the job and objects may be shared between different jobs. Therefore, sharing a virtual space between activities belonging or not to the same job is the other major problem.

4.1.1 Implementing distributed structure of execution

Implementing distributed execution structures is achieved in the same way in Unix and the micro-kernels because neither system directly provides distributed execution structures. Thus a job may be considered as a collection of local components (*local jobs*) on each node on which the job exists. Similary, an activity, which is a distributed thread of control, can diffuse on another node through a remote object invocation. An activity may be considered as a collection of local components (*local activities*) on each node on which it has diffused ; only one of these local activities can be active at a given time.

4.1.2 Sharing objects between activities

Sharing objects between activities is not achieved in the same way in Unix and the micro-kernels. The next sections describe and compare the two implementations.

a) Unix implementation

On top of Unix, there is no basic mechanism to share objects between processes on differents nodes. We therefore only allow this sharing between processes on the same node. Thus an object shared between two jobs must be shared between two local jobs on one node.

Several experimental distributed object oriented system have been built on top of Unix. They have shown three ways for sharing resources between threads on one node. For convenience, we use the term "job" to mean a shared object address space, although a different term is used in each system.

- One process per node. In this solution, all the execution structures on a node are built within one process. This solution is used in Emerald [11]. A scheduler must be provided by the system in the process of each node to manage concurrent threads. All the local activities of all local jobs are in the same address space and data sharing between local activities of one or different local jobs is implicit. This solution has two drawbacks: a scheduler must be supplied, and there is no memory protection between jobs in the same address space. The major advantage is the implicit data sharing.
- One process per local job. In this solution, each local job is implemented by a Unix process. This solution is used in Argus [12]. A scheduler must again be implemented in each Unix process to allow concurrent threads in a local job. Data sharing between local activities in one local job is implicit, because these local activities are threads in the same address space and protection between jobs is guaranted by the separation of their address spaces. However, if resources must be shared between activities in different jobs, the processes which implement these jobs must share memory. This may be done in Unix by means of the shared memory facility, but protection between jobs may be lost if they share a memory region.
- One process per local activity. In this solution, each local activity is implemented by a Unix process. There is no scheduler to write, but data sharing between local activities of one or different local jobs must again be implemented with shared memory, with the same drawback as in the last solution.

In these solutions, one found two basic mechanisms to share data between threads : implementing a scheduler in a Unix process and using Unix shared memory. The motivation for the choice of the third solution in Guide was to minimize the implementation work for rapid prototyping.

b) Micro-kernel implementation

Micro-kernels provide concurrent threads in the same address space (task). Thus data sharing between threads in one task is implicit. Micro-kernels allow tasks to share data through the mapping mecanism, which provides a cheap way of sharing between threads in different tasks. Then a *local job* is represented by a *task,* while a *local activity* is represented by a *thread* within the task that implements the job to which the activity belongs.

This implementation on a micro-kernel combines all the advantages of the three solutions on top of Unix : a direct data sharing mechanism and an implicit memory protection between jobs. The aspects related to object sharing in virtual memory are further developed in the next section.

4.2 Implementing an object virtual memory

4.2.1 Implementation on top of Unix

In the Unix implementation of Guide, shared objects are implemented in shared virtual memory. Two solutions may be envisaged for the mapping of shared objects to virtual memory regions.

In the first solution, an object is associated to one shared memory region. This allows to protect object accesses, since an object can not dynamically access another object's memory. This mechanism was used in an early Guide prototype. However, some disavantages have appeared. The cost of the *attach* and *detach* operations on shared memory data is high. In addition the size of Guide objects is variable since the size of shared memory regions is fixed ; finally the minimum size of a shared region is usually much larger than the average object size.

Therefore, we implemented a second version in wich the virtual memory of a node is represented by one single large shared memory region, within which objects bound on this node are mapped. The main drawback of this solution is that it looses the property of dynamic protection between objects. On the other hand, a significant gain in performance was immediately noticed for the binding of an object within the virtual address space (i.e. only one *attach* operation is necessary, when the activity is created on the node). Figure 6 illustrates this implementation.

4.2.2 Implementation on top of a micro-kernel

A *job* is now represented by a *task* (cf section 3.1). A task defines a protected address space which is a set of regions. Regions are access windows upon memory objects (also called segments). A memory object may be used as the unit of persistent representation in secondary memory. Thus a Guide object is represented by a region in virtual memory and by a segment in permanent storage. Memory objects are managed by external pagers; they are the units of Virtual Object Memory and Secondary Storage control.

Mechanisms for memory management provided by the micro-kernels appear to be well suited for Guide objects management. They have the following advantages:

- Encapsulating an object within a region provides an implicit protection mechanism for access to objects. Accessing an object does not give access to objects that are in a contiguous address space.
- Threads which represent the activities of a job automatically share the set of objects bound in their job, since the address space of a task is shared by the threads of this task.
- To share an object, activities in different jobs on the same node use the mapping mechanism (i.e. a memory object may be mapped in a set of regions within the same node) provided by an external mapper.

The representation of objects in virtual memory has been presented. Thus the management of object invocation, which introduce the problem of sharing objects between different nodes, is discussed in the next section.

4.3 Mechanisms for object invocation

4.3.1 Implementation on top of Unix

Two mechanisms may be used for remote object invocation. In the first solution, objects are replicated. Invoking an object on a node always involves a local copy of the object. A replication protocol is needed to ensure that the copies of an object remain consistent. This solution is used in Orca [13]. In the second solution, a single copy is maintained, and invocation always takes place on that copy, either by moving the object to the invoking node or by creating a representative of the calling activity on the node that holds the object and performing the invocation there ("diffusion").

We have used the second solution in Guide, for the two following reasons: first, the small granularity of the Guide objects would make a consistency protocol expensive. Second, the mechanisms of activity diffusion provide additional flexibility which should allow to implement load balancing policies. We now describe the invocation mechanism, as implemented on Unix.

When a method is invoked on a remote object, two cases may occur. If the object is not bound to a job on the remote node then the object is brought on the current node and invocation is performed locally. Otherwise, the *extension mechanism* is used, i.e. the calling activity diffuses to the remote node. A representative of the activity is created on the remote node and the invocation is locally performed on that node. This synchronous invocation scheme is illustrated on figure 7.

The creation of a remote local activity is called *activity extension.* This activity extension can also cause the creation of a local job on the remote node if the job does not exist yet (e.g. as a result of a previous call). This creation of a local job on the remote node is called *job extension.*

The process A/N1 which implements the invoking activity is blocked on the initial node N1 waiting for the results of invocation. Then the process A/N2 on the remote node N2 resumes its execution, terminates the method and transfers the results back to the invoking activity.

4.3.2 Implementation on top of a micro-kernel

The mechanisms used for object invocation depend on the strategy used to manage memory objects. The first strategy consists to use an external pager which constrains a segment to be located on a single node. In this case, a remote invocation is done using the same extension mechanism as in the Unix implementation (the object migration can also been used). The external pager which implements the second strategy allows an objects to have copies on several nodes, thus implementing a Distributed Virtual Object Memory (DVOM). In this case, all invocations are local. This simplifies object management, but the cost of maintaining consistency must be considered. This cost depends on the behavior of the applications that the system supports, and especially on how they use shared segments (i.e read to write ratio).

The extension mechanism and object migration may be used concurrently with the DVOM, e.g. to use the potential for parallelism provided by multiple work-stations or to implement load-balancing over the network. The choice of the strategy can be left to the user or controlled by the system..

4.4 Lessons learned

In this section, we summarize our comparative experience in implementing the basic object management mechanisms of Guide on Unix and on the Mach and Chorus micro-kernels. We consider the following aspects:

- Distributed object memory
- Persistent memory
- Global naming
- Fault-tolerance

Distributed object memory

As mentioned in section 4.2, the Guide implementation on Unix uses shared memory to allow multiple flows of control (i.e. activities) to run within the same address space. The Virtual Object Memory on a node is implemented using a single shared memory region containing the objects bound on the node. This solution has an important drawback: it does not allow any memory protection between activities belonging to different jobs.

Using Mach or Chorus allows to implement a job by a task. Thus, the virtual memory of the task is shared between the activities of a job, which are represented by threads. This architecture also uses the strong protection mechanism provided by the kernel between tasks, i.e. threads within a task can only access the objects that are maped in the address space of this task.

Using the Unix shared memory does not allow to build an efficient mecanism for managing multiple copies of an object on different nodes, since Unix does not directly provide any tools to preserve consistency of replicated data. Such a mechanism can be efficiently implemented using micro-kernel functionalities such as external pagers. A variety of consistency policies may be tested using the modularity of the external pagers without modifying the architecture of the operating system. One can choose to use the standard tools provided by the system, or one can choose to develop specific servers for a given application.

In the Guide implementation on the micro-kernels, this architecture is embodied in the concept of a distributed set of shared objects, which we call the Distributed Virtual Object Memory. Using the DVOM, one can exploit real parallelism between activities accessing the same set of objects, since they can execute on different nodes while accessing the objects.

Persistent memory

The management of the page-faults using an external pager can be integrated with the management of the secondary storage for persistent objects. An external pager manages a swap area on the disk to store the pages. This swap area can be used as a storage area for persistent objects. By directly swapping on the objects area, storage management is implicit. Thus, one can expect good performances with such a pager. Using a set of cooperating external pagers, one can also implement a protocol to manage a reliable duplication of the objects storage.

Global naming

In the Unix implementation, a global naming scheme has been explicitly developed using unique global identifiers (uid's). Such a mechanism is directly provided through capabilities in Chorus and ports in Mach. However, the global naming mechanism built on top of Unix also allows object migration through forwarding pointers, which is not provided by the micro-kernels' global naming.

Communication

The management of communication is an important difference between Unix and the micro-kernels. Communication primitives are an essential component of the micro-kernels and they are closely integrated with virtual memory, while they have been added to Unix as an upper layer. In the Guide implementation on top of Unix, we use standard mechanisms such as sockets and messages queues for internal kernel communications, but we have developed our own communication protocol to support a reliable and efficient remote object invocation mechanism. The micro-kernels directly provide such communication functionalities.

Fault-tolerance

The architecture of systems built on top of a micro-kernel has a good potential for fault-tolerance. If a failure occurs, it is local to the server on which it happens. The whole system is not corrupted. Thus, a server replication mechanism or a regenerative algorithm can allow the system to restart. A set of tools for dynamic reconfiguration of the system using functionalities such as port group, port migration or port backup can also been developed. This will be an area for further research.

Performance aspects

The implementation of Guide on top of Mach and Chorus is only at a preliminary stage ; but we can now [january 1990] run distributed applications on Guide on top of Mach 2.5 :

- we didn't modify the Guide compiler,
- we chose a centralized architecture for the shared memory server,
- we provided object migration and activity diffusion mechanisms.

We have 2 years' experience with the Unix implementation and only a few month with the microkernels one. Therefore, we cannot supply significant performance figures at the time of writing. However, we can expect better performances on the micro-kernels because:

- Activities that were implemented with processes in the Unix prototype are now implemented with threads which are light-weight processes.
- Most of distributed protocols which were developed at an upper level on top of Unix are now essential components of micro-kernels (communication, naming).

5 Conclusion

We have presented our experience in implementing the object-oriented distributed operating system Guide on top of two micro-kernels, Chorus and Mach. The objective of the paper was to assess this experience and to compare it with the first implementation of Guide which has been done on top of Unix for fast prototyping.

The Guide system provides an execution environment for an object-oriented programming langage. The main features of the system are: persistent shared objects, supported by a distributed two-level storage, transparent distribution of objects, execution model based on concurrent, distributed jobs and activities, support for synchronisation and transactions.

The first remark about the implementation of a distributed object-oriented system relates to the architecture of the support system. Kernels like Mach and Chorus are organized as a set of servers which are managed by a minimal micro-kernel. Such an organization provides a large facility and flexibility for implementing distributed systems. The modular aspect allows to experiment separately different parts of the system, and to evaluate different strategies.

Secondly, micro-kernels integrate basic mechanisms for distribution. Most services which had to be built at an upper level in the distributed systems of the Unix generation, are directly integrated in the micro-kernels. For example, Mach and Chorus provide a location independent global naming scheme. Thus, object naming is more efficient and easier than on Unix. Mechanisms for remote procedure call are also directly provided by micro-kernels. Thus, Guide remote object invocation can be readily supported.

We finally notice that some functionalities provided by micro-kernels allow to implement strategies that cannot be implemented on top of Unix. For example, micro-kernels directly support distributed virtual memory management. Coherence preserving is ensured by the kernel. For the Guide system, this method allows to implement a Distributed Virtual Object Memory that seems to be more efficient than the object management strategy on Unix, which relies on remote object invocation with a single object image. In the same way, micro-kernels allow to implement easily fault-tolerance protocols with the possible data replication or by using the notions of port and port group, wich dissociate server location and server naming, to implement reconfiguration algorithms.

On the other hand, while it appears to be easier to implement distributed systems on top of micro-kernels than on top of Unix (since most of the mechanisms that are required for distribution management are directly provided), it seems be more difficult to exploit all the power of these mechanisms. For example, the notion of a port group should allow us to represent a job as a distributed entity and to use group functionality to implement job mechanisms. Execution structures that efficiently use this mechanism are still to be explored. Another example concern object management. As we noticed, micro-kernels allow to share objects on different nodes using the DVOM. Thus, all object invocations can be made on the local image of the object. However, Guide also provides the job extension mechanism which allows a job to perform efficiently a remote object invocation while keeping a single object image. Further study, including performance evaluation, is needed to compare these two mechanisms and to design strategies that use the most appropriate mechanism according to the conditions.

Acknowledgments

We would like to thank Professor Sacha Krakowiak for his help in reviewing this paper, and Frederic Dajean for helping us in the implementation of DVOM on top of Mach.

This work was done with the support of the Open Software Foundation Research Institute in Grenoble ; special thanks to Philippe Bernadat for his assistance.

The Guide project is supported by the Commission of European Communities through the ESPRIT program in project COMANDOS (Construction and Management of Distributed Open System), 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.

Bibliography

[1] D. Decouchant, A. Duda, A. Freyssinet, M. Riveill, X. Rousset de Pina, R. Scioville and G. Vandôme. Guide: an implementation of the Comandos object-oriented architecture on Unix. *Proc. EUUG Autumn Conf.*, (Lisbon), pp. 181-193, oct. 1988.

[2] D. Decouchant, M. Riveill, C. Horn and E. Finn. Experience with implementing and using an object-oriented, distributed system. *Proc. Usenix Workshop on Experiences with Distributed and Multiprocessor Systems.*, (Fort Lauderdale), pp. 301-310, oct. 1989.

[3] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser. The Chorus distributed operating system. *Computing Systems*, 1(4), pp. 305-370, dec. 1988.

[4] *Chorus Distributed Operating System*. Chorus Systèmes, feb. 1989.

[5] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young. Mach: A new kernel foundation for Unix development. *Proc. Summer Usenix Conference*, pp. 93-112, july 1986.

[6] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rachid, A. Tevanian and M. Young. *Mach Kernel Interface Manual*. School of Computer Science, Carnegie Mellon University, sep. 1988.

[7] S. Habert, L. Mosseri, V. Abrossimov. COOL: Kernel support for object-oriented environments. *Proc. OOPSLA'90*, (Ottawa), oct. 1990.

[8] D. Julin. *Mach Objects Reference Manual*. School of Computer Science, Carnegie Mellon University, aug. 1989.

[9] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin and X. Rousset de Pina. Design and implementation of an object-oriented, strongly typed language for distributed applications. *Journal of Object-Oriented Programming.*, 3(3), pp. 11-22, sept.-oct. 1990.

[10] A. Forin, J. Barrera, M. Young and R. Rashid. *Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.* School of Computer Science, Carnegie Mellon University, aug. 1988.

[11] A.P. Black, N. Hutchinson, E. Jul and H. Levy. Object structure in the Emerald system. *Proc. OOPSLA'86*, Portland, Oregon, ACM SIGPLAN Notices, 21 (11), pp. pp 76-86, nov. 1986.

[12] B. Liskov, D. Curtis, P. Johnson and R. Scheifler. The implementation of Argus. *Proc. 11th Symp. on Operating System Principles*, ACM SIGOPS Review 21(5), pp. 111, 122 November 87.

[13] H. Bal, M.F. Kaashoek and A.S. Tanenbaum. A distributed implementation of the shared data model. *Proc. Usenix Workshop on Experiences with Distributed and Multiprocessor Systems*, (Fort Lauderdale), pp. 1-20, oct. 1989.

Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus

1 Introduction . . . 1
2 Overview of the Guide object-oriented model . . . 3
2.1 The object-oriented model . . . 3
2.2 Execution management . . . 3
2.2.1 Parallelism and synchronisation . . . 3
2.2.2 Object sharing . . . 4
2.2.3 Distribution . . . 4
2.3 Object management . . . 5
3 Survey of micro-kernel fonctionality . . . 6
3.1 Common concepts . . . 6
3.2 Specific concepts . . . 7
3.3 Architecture of a network shared-memory server . . . 8
3.3.1 The centralized architecture . . . 8
3.3.2 The cooperative architecture . . . 9
4 Description of the Unix and micro-kernel Guide implementations . . . 9
4.1 Implementing execution structures . . . 9
4.1.1 Implementing distributed structure of execution . . . 10
4.1.2 Sharing objects between activities . . . 10
4.2 Implementing an object virtual memory . . . 11
4.2.1 Implementation on top of Unix . . . 11
4.2.2 Implementation on top of a micro-kernel . . . 11
4.3 Mechanisms for object invocation . . . 12
4.3.1 Implementation on top of Unix . . . 12
4.3.2 Implementation on top of a micro-kernel . . . 13
4.4 Lessons learned . . . 13
5 Conclusion . . . 15