Is the Microkernel Technology well suited for the Support of Object-Oriented Operating Systems: the Guide Experience

R. Balter, P. Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte and X. Rousset de Pina

Unité Mixte Bull-IMAG/Systèmes, 2 avenue de Vignate, 38610 Gières, France
Internet: hagimont@imag.fr - Phone: +33 76 63 48 48

Abstract:
This paper describes our experience in the implementation of the Guide distributed object-oriented system on top of the Mach 3.0 microkernel. While many experimental distributed object-oriented environments have been implemented on Unix and much less on a bare machine, the emerging microkernel technology seems to provide a well suited trade-off between these two approaches. Microkernels provide modularity and flexibility for the design of a distributed operating system based on the client-server architecture, support of lightweight processes, efficient inter-process communication and the ability to implement flexible memory management policies. The goal of this paper is to provide an evaluation of the suitability of these features for the construction of distributed object-oriented operating systems.

1. Introduction

Several approaches have been considered in projects aiming at implementing a distributed object-oriented operating system. Some of them chose to build the entire system from scratch – i.e. on a bare machine (e.g. Clouds [Dasgupta 90]), but most of them have chosen to implement a layer on top of the Unix system (e.g. Emerald [Black 86], Argus [Liskov 87]). For some years a number of research groups have been experimenting the emerging microkernel technology – mainly Mach [Acetta 86] and Chorus[Rozier 88] - for building such distributed systems.

The goal of the Guide project[1] (Grenoble Universities Integrated Distributed Environment) is to provide a distributed platform for the support of object-oriented applications. The primary target applications are cooperative applications, such as multimedia document handling and software engineering, running on a set of heterogeneous workstations interconnected via a local area network.

A first phase of the project (1987-89) delivered a prototype on top of Unix. A second phase, started in 1990, intended to build a new version of the Guide platform on top of the Mach 3.0 microkernel. The purpose of this paper is to describe the main lessons learned from this experiment about the adequacy of the microkernel technology for building a distributed object-oriented operating system.

A preliminary experience with Mach 2.5 and Chorus, described in [Boyer 91], convinced us that the microkernel technology should provide a good framework for building such a system. This led us to redesign and implement a new version of the Guide system using Mach 3.0 and the OSF/1 MK server.

Guide provides a distributed virtual machine which is accessible to application programmers through object-oriented languages. Distributed computations, object sharing and persistence are key concepts of the Guide virtual machine. The Guide system provides the basic mechanisms which implement the functions of the Guide virtual machine.

---

[1] Guide is a component of the Comandos ESPRIT project [Balter 91].

Our main concern when designing this system was to define a modular and open architecture and to provide a generic support for object-oriented languages (i.e. to support different object-oriented languages such as the Guide language and a persistent distributed extension of C++). The choice of a microkernel as hosting environment was a good approach as it provides the required functions for building such a system, especially as far as memory management and communication are concerned.

In the next section, we present a brief overview of the Guide model, including its object and execution model and its protection model. Section 3 briefly presents the Mach features that were used for the implementation of the system. Section 4 describes the management of execution structures. Section 5 is devoted to object management. Section 6 presents the lessons of this experiment and discusses the suitability of microkernels for system implementation. Finally, the conclusion and the perspectives are given in section 7.

2. Overview of the Guide model

In this section, we give a brief overview of the Guide model. This model is embedded in object-oriented languages such as an extended version of C++ or the Guide language. The characteristics of the Guide language may be found in [Krakowiak 90].

2.1. Object and execution model

In Guide, objects are passive, i.e. they are completely dissociated from execution structures. The execution unit is a *job* (which roughly corresponds to an "application"). A job is a potentially distributed virtual space, in which one or several *activities* (sequential threads of control) are executed. Objects are dynamically bound into a job's virtual space as a result of method calls; jobs and activities spread out to remote locations if they need to access remotely located objects. The virtual space of a job is composed of several virtual address spaces, possibly distributed on several machines; this distribution is transparent to the application. The location of the objects is determined by the system according to a location policy, currently fixed by default.

Jobs and activities communicate by means of shared objects and there is no explicit message passing. Objects are persistent (i.e. an object's lifetime is not related to that of the jobs or activities which use it). In order to start an application, a user needs to specify an initial object and an initial method. A job is then created, the initial object is bound within this job, and an activity is started by a call to the initial method of that object. Other objects are then linked into the job as needed according to the calling pattern. Applications may explicitly create new objects, new activities, and new jobs.

The object memory is organized as a two-level object store. Both levels are transparently distributed. The Virtual Object Memory (VOM) provides support for executing methods on shared, synchronized objects. The Secondary Storage (SS) provides permanent storage space for objects. The VOM acts as a cache for the SS. All objects are persistent; garbage collection is performed in the SS. Objects are named by unique system-wide identifiers, allocated at creation time.

2.2. The protection model

The protection model was defined to fulfil the following requirements:

- Ensure isolation between users (i.e. an error in a user's object must not affect the objects belonging to other users) and between applications (i.e. an error

in an application must not affect applications that do not share objects with it).
- Ensure consistency with the object model. In other words, access rules must be defined in terms of the access methods applicable to objects rather than in terms of read, write or execute operations. The unit of protection is the object; in addition, the model must support users and groups.
- Solve the delegation problem. In other words, it must be possible to extend temporarily the rights of a user on a given object for the execution of a specific operation. This problem is precisely described in [Hagimont 92].

The design of the protection model relies on the following concepts:
- *User*. A user is named by the system using an identifier (*Uid*).
- *Owner*. Each object is owned by a user; ownership on a given object is inherited from that of the creator object.

In this paper, we do not describe the implementation of all these capabilities, but we focus on user and application isolation for which the use of Mach was very helpful.

3. Presentation of some Mach features

This section presents the basic Mach abstractions that were used in the implementation of the Guide system.

Kernel Abstractions

The Guide object and execution model is based on the use of the following Mach abstractions:
- *Task*: a task is a set of resources such as memory or communication ports. It can be viewed as a virtual address space divided into regions within which threads of control are executed. It provides a protected access to system resources (such as processors, ports).
- *Thread*: A thread represents a sequential activity. A thread runs within a task; there can be multiple threads running within a task, with flexible scheduling facility. Resources of a task are shared by all the threads of the task. The traditional concept of a process is represented by a single thread running within a task.
- *Port*: A port is a communication channel, which is implemented as a message queue managed and protected by the kernel. A port is only accessible via send / receive capabilities (rights).

Memory Management

The Mach kernel provides mechanisms to support large, potentially sparse virtual address spaces [Mach 92b]. Each task has an associated address map (maintained by the kernel) which controls the translation of a virtual address in the task's address space into a physical address. The physical memory is used as a cache for the virtual address spaces of tasks. The Mach kernel does not implement by itself all of this caching; some special user tasks, called *memory managers* or *external pagers*, participate in this management.

A memory manager allows a task to create *memory objects* (i.e., chunks of memory identified by ports). 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 page-fault message to the memory manager to which the object belongs.

4. Execution structures management on Mach 3.0

We describe in this section the implementation of the execution structures of the Guide system on top of Mach 3.0. The first sub-section presents this mapping on Mach, using tasks and threads, and the second sub-section discusses how Mach features allowed us to fulfill the protection requirements that relates to execution structures.

4.1. Jobs and activities

The computational model of Guide defines two major entities: *jobs* and *activities*. These entities being potentially distributed, they are represented on the nodes they span by some local components.

A job is represented by a task on each node it spans (if we do not consider protection at this stage of the presentation). Activities in jobs are implemented by threads running in the tasks that implement the jobs. An activity that has visited several nodes will be represented by a thread in each task. An activity can change its execution node through a remote object call, following a synchronous scheme.

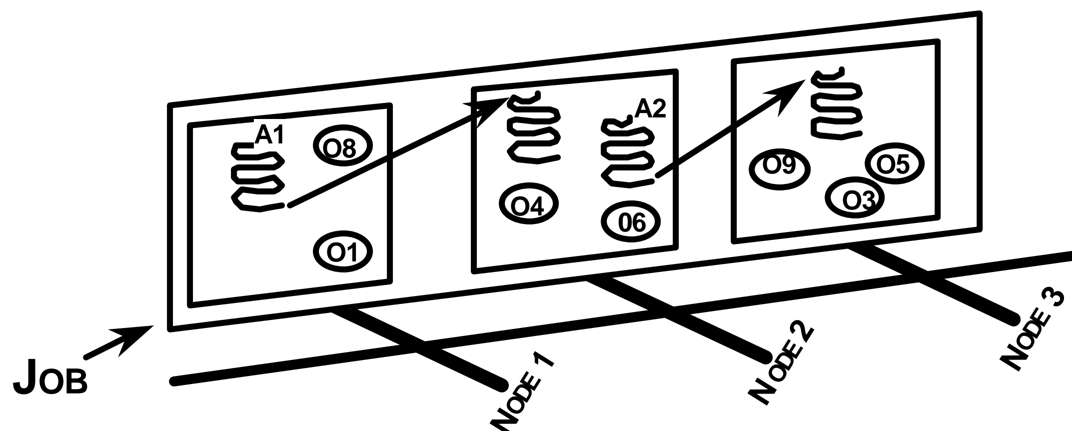

Fig. 1: Jobs and distribution

This figure shows a job running on three nodes. Activities may have spread on the three nodes. For example, activity *A1* is represented on node 1 and on node 2, while activity *A2* is represented on node 2 and on node 3. This spreading of *A1* and *A2* ensures that this job has some local components on these nodes.

Therefore, since the Guide execution model could be considered as a distributed version of the Mach execution model, Mach features were very well suited for its support.

Mach IPC is used for remote object call. A port is associated to each thread that implements a local component of an activity. This thread receives execution requests on that port from other components. At a time, there is only one active component of an activity; the others are waiting for a remote call request. For this purpose, Mach ports are very convenient because the sender of a message has no need to care about the location of the port in the network, and the interface is the same whether the task that owns the port is local or not.

Since Mach does not provide remote creation primitives, a particular daemon is present on each node to allow the creation of a local component for a job (or for an

activity) on that node. This node daemon registers the communication port associated to each component of a Guide activity. The first time an activity spreads from one node to another, the daemon is queried and it returns the port to which the request must be sent. For subsequent remote invocations, the port of the target activity is cached in the task of the calling activity.

4.2. Protection

The protection requirements that relate to isolation are two-fold: to guarantee mutual isolation between jobs and to enforce user isolation despite the sharing of objects.

Jobs do not share local components (tasks) because we want to guarantee mutual isolation between jobs.

In order to enforce isolation between users, we decided that objects of different owners must be mapped in different tasks. Therefore, a job may have several representatives on a node, each of them associated to a different object owner. An activity running in an object owned by user *X* which invokes an operation on an object owned by user *Y,* crosses the task boundaries and thus is interpreted by the system. Therefore an addressing error in a method of an object can only affect objects having the same owner. A truly object-oriented protected scheme would have required a separate task for each object, in order to prevent an error in an object from affecting another object, as it is the case in a real segmented-based operating systems like Clouds [Dasgupta 90]. This solution is not efficiently applicable here, because the average object size is small (see section 5). Figure 2 illustrates the structure of jobs including protection aspects.

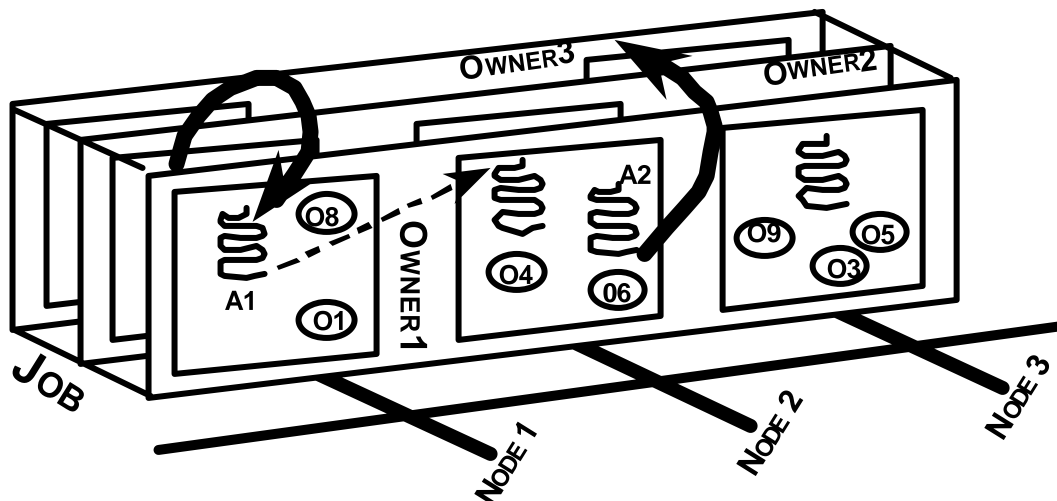

Fig. 2: Jobs and protection

This figure shows a job whose components (tasks) are distributed among three nodes. This job uses objects of different owners. These objects are respectively mapped in different tasks in order to guarantee owner's isolation. The plain arrows correspond to object calls between objects that belong to different owners.

The current implementation associates a Mach port to each local component of an activity (thread) in a task, and the thread waits for a new execution request as it is done in the distributed case.

Hence a task which is a local component of a job will only have rights on ports that are associated to local components of activities of the same job. It will not be possible for an activity to send an execution request to an activity that belongs to

another job. Using ports ensures that activities belonging to different jobs will not interfere.

Another benefit of Mach ports that relates to protection appears in our solution for the delegation problem. In our solution (that could be described in a longer paper), we have to make some checks each time an object call involves two objects that belong to different owners. When such a call occurs, the object call crosses task boundaries, and the check can be done in the called task, but the problem comes from the fact that the two tasks that are involved may be associated to the same object owner on different nodes, in which case no check is needed. In fact, we need to authentify the owner associated with the calling task. In our implementation, we allocate two ports for each local component of an activity: the first one (called the *twin port*) is given to all the tasks (local component of the job) that are associated to the same object owner in the job, the second (called the *public port*) is given to all the other local components of the job. Therefore, a thread in a task that receives an object call request has to do the check if the message is received on the public port. The figure 3 illustrates this mechanism.

We also have to authenticate a task which is a local component of a job when it asks for a service to the node daemon of a node. The port that identifies a task in Mach is used as a capability in our system. A task that requests a remote invocation to the node daemon of a node authenticates itself by giving its private port (*mach_task_self*). It allows the node daemon to authenticate the job which requests a service. The same mechanism is used when a mapping request is sent to a memory manager that will have to check if the mapping is allowed for the object owner associated to the task (see section 5).

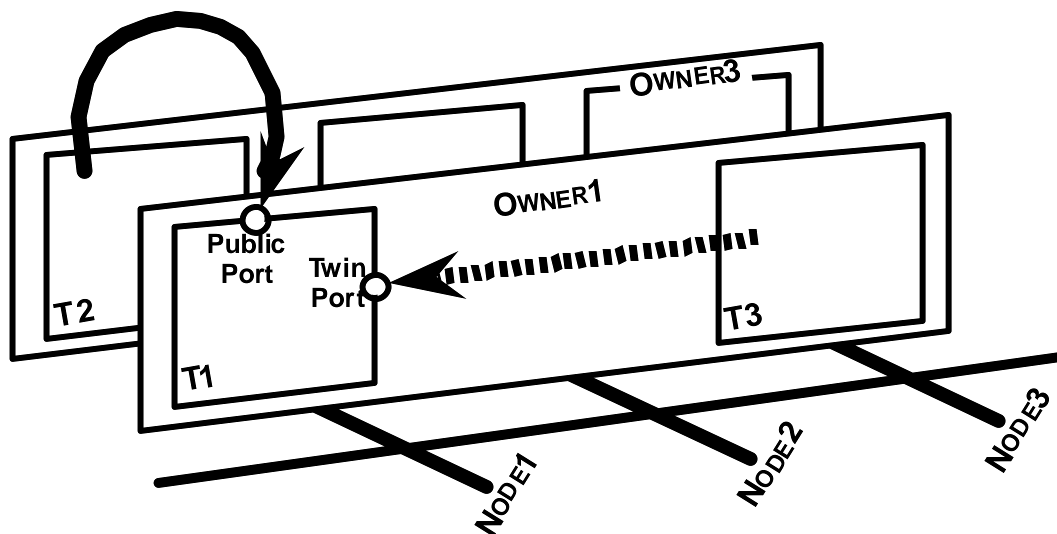

Fig. 3: the task authentication mechanism

This figure presents a job composed of five tasks. Tasks T1 and T3 contain objects belonging to Owner1, while task T2 is associated to Owner3. A method call from an object within T2 to an object within T1 will use the public port (plain arrow), while a method call from T3 will use the twin port (dashed arrow).

5. Object management on Mach 3.0

We describe in this section the implementation of the Guide Virtual Object Memory on top of Mach 3.0. The first sub-section focuses on object sharing using the memory manager facility; the second sub-section deals with persistent object storage.

5.1. Object sharing

Mach imposes two limitations for the design of the Guide system. First, Mach provides a limited number of system resources (such as ports) per task. This does not allow the potential sharing of many objects by two different jobs. Moreover, the unit of mapping in the address space of a task is a set of pages, which is much greater than the average size of our objects (a few hundred bytes). For these two reasons, we introduced the concept of *cluster* (a cluster is a set of objects) in order to support the sharing of fine-grained objects. Clusters also allow to group objects into pages in order to minimize object transfers between VOM and SS. As a cluster may contain several objects, the sharing of an object implies the sharing of other objects. Thus, all the objects in a cluster must belong to the same owner.

Microkernels provide the *memory manager* facility. A memory manager is a user process which allows other processes to map a chunk of memory, identified by a port, into their virtual address spaces. From the microkernel point of view, the memory manager is responsible for handling page faults that may arise on this chunk of memory. We used this facility to implement clusters. Clusters are managed by a set of memory managers that we call *cluster managers*. These managers are distributed among the network and may cooperate together. A cluster manager implements two kinds of functions:

- mapping, sharing and protection control;
- paging functions such as page-in/page-out requests from the Mach kernel (the interface between the kernel and the cluster manager is defined in [Mach 92a]).

Figure 4 depicts the interactions between jobs, the microkernel and the cluster managers.

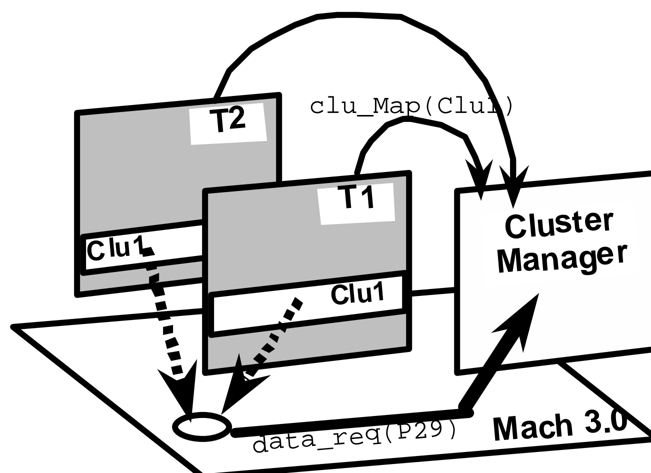

Fig. 4: Cluster mapping

This figure depicts the sharing of the cluster *Clu1* between tasks *T1* and *T2*. In order to map that cluster, both tasks send a *clu_Map* request to the cluster manager. After checking the rights of the tasks, the cluster manager returns a port (P29) which will be used by the tasks when mapping the cluster. Access to the cluster will cause page faults that will be redirected by the kernel to the cluster manager using the port P29.

5.2. Object storage

In the computational model defined above, we made the distinction between two levels of memory: the VOM which provides support for method executions and which consists of the union of physical memory and the swap disk spaces of each node, and the SS, which provides long term storage support. The interface between

VOM and SS is defined in terms of load and store operations. The separation between VOM and SS is useful for the following reasons:

- Modularity (i.e., ability to reuse existing storage servers such as AFS or Pegasus [Leslie 93]),
- Security (i.e., access control to the persistent storage interface),
- Quality of service (i.e., servers can provide different storage policies such as reliability, fast access, etc.).

Moreover, movements of clusters between the VOM and the SS are entirely managed by cluster managers. This allows this transfer to be performed very efficiently. Transfers are performed on demand (i.e. the whole cluster is not transferred in VOM at mapping time): a page is only loaded in VOM when the cluster manager receives a fault on this page. This ensures that the transfer occurs only for those pages that are actually used by jobs. Furthermore, when the cluster manager has to store the modified image of a cluster in SS, only dirty pages are copied to SS. This scheme allows the traffic between VOM and SS to be reduced.

6. Evaluation

In this section, we summarize the lessons learned from our experience in building the Guide system on top of Mach 3.0. We first discuss the adequacy of microkernels for the implementation of our system. Then, we present some performance figures performed on our prototype.

6.1. Adequacy of Mach 3.0

We have shown in the description of the implementation of the Guide system that the microkernel technology provides a well suited platform for the design of a distributed object-oriented operating system.

As mentioned above, the Guide model can be viewed as a distributed version of the Mach model. Consequently the mapping of the Guide abstractions (jobs and activities) on the Mach abstractions (tasks and threads) was straightforward. In Guide, an application involves a potentially distributed virtual address space called a job within which several activities may run. Jobs and activities are naturally represented by a set of tasks and threads running on the nodes visited by the application.

The benefits of the port concept are twofold:

- The port provides a location transparent address for message passing between tasks. Therefore, in our implementation, we were able to develop and debug the entire Guide kernel on a single machine, since message passing between tasks on a single machine or on different machines have the same interface. The step between a centralized prototype and the fully distributed version was quite small compared to the former experiment achieved on top of Unix.
- Mach ports are protected in the sense that a port cannot be used unless it has been explicitly given by a task that has the required rights on this port. This allows the implementation of the isolation requirements for jobs and users. This also allows the authentication of these user-level tasks when they cooperate with the Guide kernel.

The ability to design our own memory manager was one of the key benefits from using the microkernel approach:

- It allows a simple implementation of object sharing between different nodes, which was not straightforward on Unix.
- It allows the implementation of flexible consistency policies according to application requirements.
- It allows the efficient management of fine-grained objects. The use of memory managers allows a clear separation between the object as unit of addressing, the cluster as unit of mapping, and the page as unit for I/O transfers. This allows an optimization of the multiple facets of memory management.
- Cluster managers enforce the protection model by controlling user rights on objects into a protected server. In addition, clusters are not made visible to applications. This also improves the protection scheme of the system.

However the implementation of Guide suffered from some missing features which are listed below:

- The ability to create a task on a remote node would greatly simplify the overall architecture and especially the management of the execution structures.
- Protected ports have a major drawback in a distributed environment: applications cannot share ports. Should this facility be available, finding a specific task within a job would have been improved, for example by removing the need to contact the node daemon.
- Port groups would have been convenient for managing distributed entities such as jobs and activities. They would also have been useful for providing fault tolerance facilities.
- Implementation of synchronisation tools such as semaphore objects shared between task located on the same or different nodes is not an easy work. The designer should either provide a semaphore server or ensure that all threads sharing a semaphore object know each other's ports.
- As proposed in [McNamee 90], it would be interesting to provide the ability to manage page replacement in physical memory at the level of a cluster manager, since a cluster manager does have some knowledge about cluster usage that the kernel cannot manage. For example in the current implementation, cluster managers collect and store information about page access or cluster sharing which could allow improved paging.
- It is not possible currently to develop servers on Mach independently of OSF/1 (for example to get hight level I/O functionalities).

6.2. Performance

The implementation of Guide-2 started at the end of 1991, using first Mach-3.0 and then Mach 3.0/NORMA both with OSF-1 /MK-13 on Bull-Zenith P.C. i486 (33 MHz) connected to a 10 Mb Ethernet. Mach-3.0/NORMA is a version of Mach-3.0 that allows to consider a set of workstations connected by a LAN as a multiprocessor. NORMA integrates the network communication inside the microkernel, with a substantial gain of performance (a factor of about 50). Table 1 gives some preliminary figures. These figures measure the elapsed time of the basic functions provided by the cluster manager: cluster creation, cluster mapping, handling a read or write page fault and cluster unmapping. In this experiment clusters in secondary storage were simply implemented by Unix files. A more elaborate version of the storage server is under way.

Clusters Operations	*Time (in ms)*
clu_Create[1]	130,0
clu_Map	10,0
page_read[2]	6,8
page_write	6,8
clu_Unmap[3]	124,0

Creating a cluster is a cost intensive operation since it requires to create two Unix files: one for storing the cluster descriptor and one for the actual cluster. In addition, this operation involves an access to a specific file containing the information used to allocate a global unique name for the cluster. The cost of this operation will be significantly reduced when using the final version of the storage server.

Mapping a cluster requires to read the descriptor of the cluster in order to pick up its size and its owner. In the upcoming version of the storage server cluster descriptors will be cached in main memory to reduce the overhead of accessing the cluster descriptor.

The figures related to page faults (*page_read* and *page_write*) take into account the cache management associated to the Unix file system.

Cluster unmapping implies to store all modified pages in SS. In our scenario all pages were modified; this explains the relatively high value for this figure. Real applications are not expected to update all pages at each execution; if it were the case, the cluster manager would obviously become a bottleneck for the whole system.

To conclude this section, we are convinced that these results are very promising regarding the level of functionality provided by the cluster machine (i.e.:, object sharing, persistence and protection) and the benefits, in terms of security and modularity, offered by our architecture. The performance is expected to be slightly improved when the implementation of a full storage server will replace the current implementation based on Unix files.

7. Conclusion

We have shown in this paper how we used Mach 3.0 features in the implementation of the Guide system. The Guide system provides an execution environment for an object-oriented programming language. The main features of the system are: persistent shared objects, supported by a two-level distributed storage, transparent distribution of objects, execution model based on concurrent, distributed jobs and activities, and support for protection.

The objective of this paper was to assess our three year experience in using Mach and to present the lessons learned from this work concerning the adequacy of microkernels for building distributed systems. Systems like Mach and Chorus are organized as a set of servers which are managed by a minimal microkernel. The flexibility provided by this architecture greatly helps the design of distributed systems. Furthermore modularity allows different parts of the system to be designed and tested separately on the one hand, and various resource management policies to be experimented on the other hand.

---

[1] Clusters are 10 pages long (40960 bytes).

[2] Pages are transfered from SS to VOM (in read or write case).

[3] Pages are stored from execution memory to persistent store.

The Guide prototype is currently running on a network of i486-based machines using Mach 3.0 and the OSF/1 MK server. The implementation is nearly complete. Several applications such as a distributed co-operative spreadsheet are already running, that allow debugging and tuning of the system. These experiments have shown that the standard version of Mach 3.0 performs poorly over the network. Therefore, in co-operation with OSF-RI, we are currently experimenting the NORMA version of Mach 3.0 to reduce the cost of remote invocations. Preliminary experiments with this advanced version are very encouraging.

Acknowledgments. We would like to thank Professor Jacques Mossière for his help in reviewing this paper. The Guide project is supported by the Commission of European Communities through the ESPRIT project COMANDOS (Construction and Management of Distributed Open Systems), the Universities of Grenoble (Institut National Polytechnique de Grenoble – Université Joseph Fourier) and Centre National de la Recherche Scientifique.

The Guide implementation on top of Mach 3.0 was carried out in collaboration with the OSF-Research Institute (Grenoble).

Bibliography

[Acetta 86]
M. J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, M. Young, Mach : a new kernel foundation for Unix Development, *Proc. of the USENIX 1986 Summer Conference*, Jul. 1986, pp. 93-112

[Balter 91]
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, Architecture and implementation of Guide, an object-oriented distributed system, *Computing Systems*, vol. 4, n° 1, 1991, pp. 31-68

[Black 86]
A.P. Black, N. Hutchinson, E. Jul, H. Levy, Object structure in the Emerald system, *Proc. First ACM Conf. on Object-Oriented Systems, Languages, and Applications (OOPSLA)*, Portland, Sept. 1986

[Boyer 91]
F. Boyer, J. Cayuela, P. Y. Chevalier, A. Freyssinet and D. Hagimont, Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus, *Proc. Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, Mar. 91, pp. 283-299

[Dasgupta 90]
P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahmad, R.J. LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, and C.J. Wilkenloh, The Design and Implementation of the Clouds Distributed Operating System, *In Computing Systems*, vol. 3, n° 1, Winter1990, pp. 11-45

[Hagimont 92]
D. Hagimont, S. Krakowiak and X. Rousset de Pina, Protection in an Object-Oriented Distributed System, *Proc. of the 4th Int. Workshop on Object Orientation in Operating System*, Paris, Sep. 1992
[Krakowiak 90]
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*, vol. 3, 3, pp 11-22
[Leslie 93]
I. M. Leslie, D. McAuley and S. J. Mullender, Pegasus - Operating System Support for Distributed Multimedia Systems, *ACM Operating Systems Review*, 27(1), Jan. 1993, pp. 69-78
[Liskov 87]
B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, Implementation of Argus, *In Proc. 11th Symp. on Operating System Principles*, vol. 5, 21, Nov.1987, pp. 111-122
[Mach 92a]
Mach 3 Kernel Interfaces, *Open Software Foundation and Carnegie Mellon University*, edited by K. L. Loepere, Jan.1992
[Mach 92b]
Mach 3 Kernel Principles, *Open Software Foundation and Carnegie Mellon University*, edited by K. L. Loepere, Jan.1992
[McNamee 90]
D. McNamee and K. Armstrong, Extending the Mach External Pager Interface to Accomodate User-Level Page Replacement Policies, *Proc of the Mach Usenix Workshop,* Burlington, VE, Oct 1990, pp 31-43
[Rozier 88]
M. Rozier, V. Abrossimov, F. Armand, J. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois and V. Neuhauser, Chorus Distributed Operating Systems, *Computing Systems*, vol. 1, n° 4, 1988, pp. 305-370