The NanOS Microkernel: A Basis for a Multicomputer Cluster Operating System*

Francesc D. Muñoz-Escoí
José M. Bernabéu-Aubán
Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València
Camí de Vera, s/n
46071 València - SPAIN
Tel: 34 6 387-7069, **FAX:** 34 6 387-7358, **e-mail:** {fmunyoz, josep}@iti.upv.es

Abstract *NanOS is an object-oriented microkernel that has been taken as the basis to develop a distributed operating system. It provides a set of abstractions that can be used to build user-level applications. They are: objects, agents and tasks. An agent is a protection domain which holds a collection of objects and a set of references to other external objects. Tasks are execution threads that can invoke objects placed in external agents. So, tasks can traverse multiple agents as a result of an object invocation. A local object invocation service is also provided by the NanOS microkernel.*

To extend the microkernel-based services, which are intended to only a machine, to a distributed environment, an object request broker is being designed. This ORB manages inter-machine invocations. As a result, distributed applications may be built using the ORB services.

Keywords: Microkernels, Cluster Systems, Object Oriented OS, Reliability.

* This work was partially supported by the CICYT (Comisión Interministerial de Ciencia y Tecnología) under projects TIC 93-0304 and TIC 96-0729.

1 Introduction

Current trends in operating system development are based on a minimal kernel, which takes the form of either a microkernel [1, 2, 7, 12, 14, 15], cache-kernel [4] or exokernel [6]. The latter alternative provides the most extreme solution, since an exokernel only offers a multiplexing of the physical resources of the hardware without modeling any high level abstraction, such as processes, execution threads, objects or any other. In this case, the basic support to develop applications has to be offered in libraries which provide the traditional kernel interface.

Microkernels constitute a more conservative approach. They offer a minimal set of abstractions that are considered the required basis to develop user level servers or applications. Usually this basic abstraction is the object; i.e., a software module which maintains a state that is only accessible using a collection of operations which compose the object's interface. Moreover, a microkernel has to provide a dynamic abstraction which provides the unit of execution of the objects' code. This dynamic abstraction is the execution thread. To be able to execute multiple objects, the microkernel has to offer to the execution threads an object invocation service, allowing them to visit several objects invoking their interfaces' operations.

NanOS [10] follows the object-based microkernel approach. Thus, the object is the most important abstraction provided by our kernel. Execution threads, referred to as tasks, and the invocation service are the other two logical blocks used to build the system. In our case, the object abstraction is tightly bound to the protection domain, or agent. An agent models an address space where objects can be placed and protected. Also, the agent is the unit of resource distribution in our system. It is the entity which owns the object references that allow its tasks to invoke external objects and, in this way, move to other agents.

The aim of NanOS is to provide a flexible basis to develop a distributed operating system on top of it. To achieve this goal, an ORB [11] is being designed to extend the interdomain invocation service capabilities of NanOS to a distributed object invocation facility. We plan to give our ORB some support to replicated objects, allowing in this way the development of highly available applications. Thus, the ORB will offer a good platform to develop the upper levels of an operating system for a multicomputer cluster, offering a single-system image.

The rest of the paper describes first the basic concepts offered by NanOS to its upper levels. Next, in section 3, the overall structure of the microkernel is outlined. Section 4 describes other microkernels, comparing them with our approach. Finally, section 5 gives some measurements about the services provided and summarizes the most important characteristics of NanOS.

2 Services and Abstractions

As stated above, the most important abstraction provided by our microkernel is the *object* concept. NanOS deals with object registration, invocation and deregistering using one of its internal objects: the ObjectMgr. An object registration is the first step required to make an object publicly available. Once an object is registered, given its name and its local identifier in its server agent, other agents may get valid references (object descriptors) for it, resolving its name. Once an external agent has one object descriptor, its tasks are able to invoke any one of the object's interface operations. Finally, the kernel maintains a reference count for each registered object. Once a reference count decreases and gets a zero value, the object is automatically deregistered. An explicit deregistration operation is provided, too.

The *task* is the dynamic abstraction provided by our kernel. A task is an execution thread. Tasks are scheduled, dispatched and synchronized using the services provided by several objects that are internal to the kernel, but offer publicly their interfaces. The current implementation of all these objects is placed into the kernel, although future releases will allow a user-level scheduler, which use the interface provided by the Dispatcher object. Since, synchronizing objects only depend on the Scheduler operations, they may be placed at user-level, too.

Agents are the protection domains of our system. An agent is the unit for object descriptor assignment. So, all tasks running in the same agent have potentially the same set of accessible external objects. To support an agent, the microkernel associates to it a VirSpc object which represents its virtual address space. Two kinds of agents exist in our system. The first one is the user agent, which models each of the user-level protection domains. It is characterized by the virtual range of addresses assigned to its virtual space (the first three gigabytes in our PC implementation) and for the presence of a unique entry point to its incoming invocations (a fixed low address in its virtual space). Appropriate server code has to be attached at this entry point to maintain a set of pre-allocated stacks for the incoming tasks and to route the arriving calls to the suitable object and method. The other kind of agent is the kernel agent. The kernel has been modeled as an agent with its own virtual space (which encompasses the fourth gigabyte in our PC implementation) and its entry point (invoked using the appropriate system calls and managed by the CallMgr kernel object.) Figure 1 depicts an example of the agents distribution in a machine with the NanOS microkernel.

Some other kernel-level objects offer the required service to provide memory management, as in other kernels [7]. To begin with, the MemObj, or memory object, is the representative of some part of the system's memory. There are two types of these memory objects. The

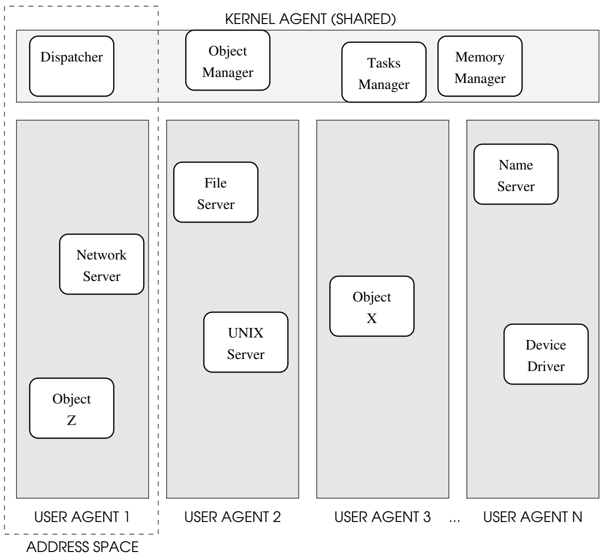

Figure 1: Agents, virtual spaces and objects in a system node.

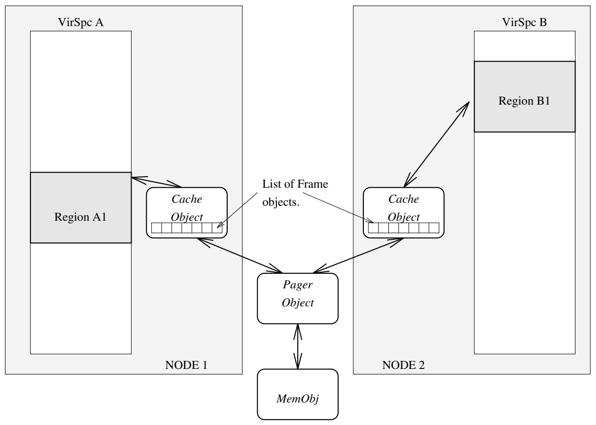

Figure 2: Relationship among MemObjs, Pagers, Caches, Regions and VirSpcs.

first one backs the image of an object maintained in secondary storage; i.e., a persistent object. The other type represents part of the main memory of the local machine.

The VirSpc object models a virtual address space which may be used by the kernel or by any of the user-level agents. A Region represents a part of a virtual address space that has been allocated to map (part of) a MemObj on it. A Cache holds the physical pages of a MemObj that are in use in a node. A Pager is associated to a MemObj and it must hold a reference to each one of the Caches that back part of the memory object pages, providing to the caches some methods to request new pages or release some of the cached ones. Finally, other machine-dependent objects take part in the memory management, as the MMU and the TableMgr, which model the MMU of the local machine and its page tables.

This relationship among MemObjs, Pagers, Caches, Regions and VirSpcs is depicted in figure 2. In this figure there is a memory object that has been mapped in two virtual address spaces located in two different nodes. To make possible this mapping, a Region has to be declared in each virtual space, allocating the range of virtual addresses where the memory object has to be mapped. Later, a Cache has to be created in each node to back the contents of the memory object needed in those machines. Finally, a Pager is associated to the memory object and bound to the Caches. This Pager is used to bring the contents of the frames that hold the memory object contents.

To map some memory in a VirSpc object some steps must be followed. First of all, a call to the `AllocMem()` method of the VirSpc reserves a Region object in that virtual space, registering a range of virtual addresses as "in use." Next, a valid reference to a MemObj has to be obtained. Finally, the `Map()` method attaches the memory object to the region created earlier. To do this, the virtual space has to check first if any Cache exists in the local node backing the memory object being mapped. If so, this cache was bound earlier to the MemObj, which provides also a Pager for the cache frames. If no local cache is found, a bind operation has to be initiated, creating a Cache and relating it to the MemObj and its Pager. In both cases, the Cache is finally attached to the Region, which knows the virtual addresses range associated to the cache's pages. This information is used to establish the virtual to physical translation data in the page tables of the MMU when the translation has to be set, i.e., when the Pager introduces some new pages in the Cache as a result of a page fault or an explicit request to get memory for a given virtual

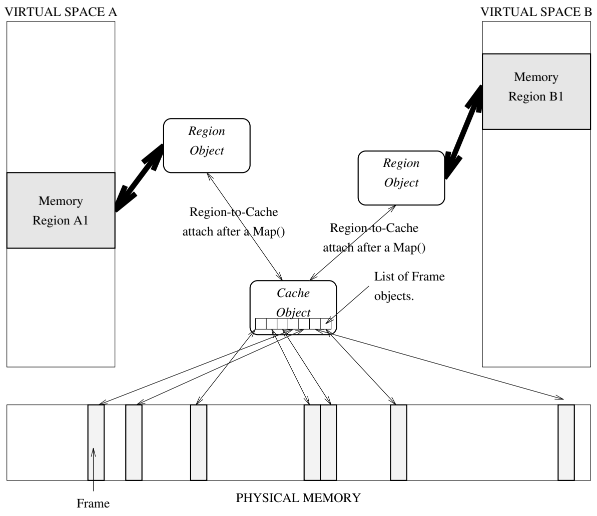

Figure 3: Memory shared between two address spaces in the same node.

address. Since only a Cache object is needed to back a memory object in a node, sharing memory among multiple address spaces is as easy as attaching multiple Region objects to the same Cache, as depicted in figure 3.

This organization of the memory management services and the object registration and invocation mechanisms allows the extension of the kernel with new objects. For instance, a device driver can be installed into the kernel if its code and data are stored in a memory object and this object is mapped in the VirSpc of the kernel agent. Later, the kernel Loader must relocate its code to the address where the driver has been mapped. Finally, the new device driver object must be registered in the ObjectMgr to generate an initial object reference which will be set in the kernel's name server.

Finally, let us take a look to the objects invocation service. This is the basic service provided by the microkernel. Other complementary services are available when some of the microkernel objects which have registered their interfaces are invoked by user-level tasks, but to use them the invocation service is indispensable, too. To start an invocation, the invoker task must have a valid reference to the object it wants to call. This object reference, or object descriptor, may be obtained either resolving the object's name in the microkernel's name server or receiving it as a returning parameter of a previous invocation. These object descriptors are maintained by the agents. When a new agent is created, it already holds a reference to the kernel-level name server. Thus, all agent's tasks have access to the name server and they are able to obtain new object references if they know the name of some external objects.

If the invoker task's agent has the appropriate descriptor, a call to the microkernel is initiated. When the microkernel serves the invoke request, it checks the validity of the object reference and, if it is correct, translates the object descriptor to an object identifier that is known only by the server agent for this object and by the microkernel. Then the server agent is located and installed. Later, the task returns to user-level at the entry point of the server agent, where it gets a new stack and executes the skeleton code that drives it to the invoked method. When the task finalizes the execution of the invoked method, releases its stack and calls again the microkernel, which finds out its state in its previous agent and returns the task to it. Note, that all these changes are made avoiding any task switch. In our current implementation in PC machines, an interdomain invocation and return as the one described, transferring five 32-bit arguments, needs 22 μs in a machine with an Intel Pentium at 90 MHz.

Since the microkernel is modeled as another agent, it also maintains an object descriptor table. So, tasks running in the kernel can invoke objects placed in other agents. This mechanism is used to invoke interrupt handlers or device drivers that are not installed at the kernel level when an interrupt is raised by a device. However, kernel services are also provided to make possible the installation of new device drivers at kernel level, if needed.

3 Microkernel Organization

The microkernel has been designed as a set of related objects which serve the invocation service and provide different interfaces that user-level tasks may invoke at any time. This set of objects comprises a reduced subset of machine-dependent objects (the CPU, CPUContext, MMU, TableMgr and IntrMgr, which have machine-dependent code but offer machine-independent interfaces) and a subset of machine-independent ones, which use the methods of the latter. Having a little number of machine-dependent objects, the migration of the microkernel to other hardware architectures results easy. In fact, we developed the initial version of NanOS on Sun SPARC machines and later we have ported this kernel to PC's.

Also, some extensions are being planned to the object invocation mechanism. An ORB architecture is being designed to extend the current invocation service, allowing the invocation of remote objects in a distributed environment.

3.1 Machine-Dependent Objects

The machine-dependent objects do the following tasks. The CPU is used to enable and disable interrupts. Also, if a multiprocessor implementation is considered, there are multiple CPU, MMU and TableMgr objects to model each one of the processors being considered.

A CPUContext models the set of registers of the CPU and allows the change of any of their contents. So, an object of this kind is used to model the state of a task when it has to release the CPU when the Scheduler requests this action. In the Intel x86 release of NanOS, a CPUContext maintains the general CPU registers in the format of a TSS segment [9]. As a result, the task switching capability of the x86 processors is used.

The MMU provides methods to change the current virtual address space and to update the mappings placed in the page tables of each processor. The TableMgr uses its services to provide a higher-level interface which hides the physical representation of the page tables and the number of page table levels maintained by the MMU, offering the image of a unique and large page table for each address space. At this level it is possible to associate physical pages to the any virtual address space.

Finally, the IntrMgr is used to install different interrupt handlers for each level of interruption of the CPU, routing the incoming interrupts to the appropriate interrupt handler. At this level there are multiple semi-active tasks, each one associated to a different interrupt level. When an interrupt arrives, the processor automatically does a task switch, blocking the current task and activating the one associated to the incoming interrupt level. Support is also provided to undo this task switching when the interrupt handling is terminated, and to report the task change to the Scheduler. This second alternative allows the interrupt task to be scheduled and to invoke objects placed outside the kernel agent. Thus, if an interrupt task is made schedulable, the device driver attached to its interrupt handler may be placed in a user-level agent.

3.2 Machine-Independent Objects

The set of machine-independent objects is a little bigger. We only describe the most important objects of this type. The ObjectMgr maintains all the registrations of objects that have been made, holding the identifier of the agent which serves the new object and the private identifier of it in that server. This information is needed to locate the agent where an object is placed when this object is being invoked. It also deals with object descriptor management, maintaining a reference count for each registered object, and with the naming service, allowing the association of names to objects.

The TasksMgr is used to create new tasks and to register the interdomain calls made by any of them, allowing in this way their return to their home agents. To achieve this, each time a task calls an external object, the TasksMgr saves the place where the invocation was made (to this end, the instruction and stack pointers plus the client agent identifier are saved,) and this information is chained to the current list of invocations made by that task. When a task returns from the called object, the kernel restores its previous state, returning it to the agent and routine where the call was initiated.

The Scheduler manages the state of all system tasks, arranging them in any of the 32 available priority classes. It provides methods to prepare and suspend tasks; so its services are required by the synchronizing objects. It uses a Dispatcher object to do the task switching in each one of the available processors. All machine Dispatchers have access to a global pool of CPUContext objects, which maintain the states that had all local tasks when they released the CPU.

Also related to tasks are the Semaphore, Lock and EventVar objects, which are three different classes of synchronization objects, each one with their own semantics:

- A Semaphore object provides the `P()` (test) and `V()` (increment) methods. They follow the behavior of the original semaphore primitive [5].

- Locks constitute a variant of the semaphore object that does not block the owner of the lock if it requests its suspending method several times. However, to release a lock, its owner has to call its `Release()` method as many times as it called the `Request()` one.

- In an EventVar object, many tasks may be blocked waiting for the signaling of an event. When its `Signal()` method is requested, all the blocked tasks are immediately restarted. Later, the EventVar remains signaled. As a result, if a new task calls its `Wait()` method again, it is not suspended unless its `Reset()` method was previously called.

Each type of synchronization object accepts in its suspending method an additional argument which may be used to provide a timeout value or to request the test of the blocking state of the object; i.e., if the current state of the synchronization object will block the requesting task or not.

The CallMgr serves the kernel entry point, distinguishing between user-level targets, which receive the generic invocation service and the kernel-level ones, whose routing is optimized.

There is also an AgentMgr which allows the creation and destruction of agents in the local node.

Device drivers are not included as a collection of objects placed in the kernel. However, the IntrMgr object described in the previous section provides support to install the interrupt handlers associated to these drivers, either in the kernel or in user agents. Drivers can be also placed in the kernel or in user agents, but they do not belong to the set of objects that build the microkernel indispensable services. They have to be registered and installed in the agent chosen by its programmer.

Finally, a big number of objects are related to memory management. VirSpcs, Regions, MemObjs, Caches, Pagers and other objects have been already described in the previous section. To complete the picture there is a FrameMgr object which finds out at boot time the whole machine physical memory and manages the set of available physical pages that exist in the machine.

3.3 ORB Extensions

A little number of kernel-level objects is needed to manage object invocations. Therefore, it is easy to modify the object invocation service. We are planning now to develop an ORB for NanOS which will complete and extend the current invocation service to provide invocation support in a multicomputer cluster. To do this task, a new network device driver and a transport protocol server are being implemented. The ORB will use their services to communicate with other ORB's in the other nodes of the cluster.

A multicomputer cluster offering a single-system image is the kind of system we are developing on top of NanOS. A system like this offers the advantage of an easy increase in the computing power of the system adding more machines to the cluster. To allow that, a flexible configuration tool must be designed and included in the system. Also, we plan to extend the ORB with support for replicated objects, offering different replication strategies, ranging

Table 1: Executable and source code sizes.

Concept	**Source** (Lines)	**Exec** (Bytes)
Whole kernel	19.636	46.493
Language:		
C++	16.787	39.413
Assembler	2.849	7.080
Arch. dependency:		
Independent	14.785	35.577
Dependent	4.851	10.916

from passive replication [3] (a primary and multiple backups) to fully active replication [13] (multiple primaries offering different classes of consistency, and having also as many backups as needed). This will allow the development of highly available applications.

3.4 Code Size

Table 1 sums up the figures about the kernel size. All the source code is fully commented. Removing comments, approximately a 50 % of the lines would be eliminated.

This table shows first the entire size of the kernel source code (in lines) and that of the kernel text image (in bytes).

The second group of figures divides the kernel code according to the implementation language. As we can see, only a 15% of the source code has been written in assembler. Once this code has been compiled, its 7.080 bytes mean the 15% of the total object code, too.

Considering hardware dependency, the 25% of the source code must be translated to port the kernel to a different architecture. In the current PC release, this source code generates the 23% of the text image.

4 Related Work

The architecture of the NanOS microkernel is based on two layers of objects that provide a minimal set of services and abstractions that make possible the development of user-level applications and servers. The bottom object layer is composed by a set of architecture-dependent objects. On top of it, there are a set of architecture-independent objects. The abstractions provided are the object, the agent —or protection domain— and the task —or execution thread. The unique kernel service is object invocation. As a result of this architecture we have an efficient microkernel that can be easily migrated to different machines.

The exokernel approach proposed in [6] does not consider kernel abstractions as a convenient model to develop the operating system support. Thus, exokernels do not provide any abstraction to its upper levels, and its operating system interface has been lowered to the hardware level. As a result, the user-level code has to manage itself all the hardware. The advantages of this approach, according to its authors, are more performance, reliability, adaptability and flexibility than microkernels. But, this has to be achieved writing the appropriate code to manage the hardware at user-level. Therefore, the exokernels are not portable and require additional programming efforts to write a user-level application.

Cache-kernels [4] follow an approach similar to exokernels. So, they have the same advantages and drawbacks.

The *Paramecium* [15] and *Spring* [7] microkernels are also object-based. Paramecium places more emphasis in its support to integrate new objects at kernel level, using certification techniques to ensure a good level of security when these objects are included in the kernel. Besides certification services, it provides processor and memory management and a name service. The Spring nucleus, on the other hand, only provides the object abstraction and the object invocation service. All other services may be implemented at user level.

Our solution is similar to the ones provided by Paramecium and Spring. In our case, it is also possible to include new objects in the kernel, as device drivers, to decrease the time required to serve the interrupts raised by the device, or to improve its invocation time, since the kernel is always mapped and active, independently of the current active user agent.

5 Summary

The NanOS microkernel offers support for object-oriented services and applications. Its main services include object invocation, memory management and device driver support. NanOS also provides a reduced set of abstractions which comprises objects, agents and tasks.

The kernel itself has been organized as a set of cooperative objects. Some of them offer public interfaces to the upper levels. Synchronization objects, memory providers and virtual spaces are three examples of kernel objects of this kind.

Portability and efficiency were key design objectives of this system. To increase the portability of the kernel code a reduced set of architecture-dependent objects with architecture-independent interfaces was considered in the design stage of the kernel development. As a result, porting NanOS to other architectures implies the rewriting of all these objects. Currently, we already have two releases of our kernel running in two different architectures. So, the first objective has been accomplished. Efficiency is provided by the invocation service, which allows interdomain invocations without needing any task switch to do so.

We plan to build a distributed object-oriented operating system offering, among others, a UNIX interface to applications. The NanOS kernel is the base for this future system. It has to be extended with network support in the next stage of its development. At the same time, we are designing an ORB which will provide a good platform to develop distributed applications and servers. We plan to extend this ORB with additional services such as the support and management of replicated objects, which will provide the basis for highly available applications and servers.

References

[1] J. M. Bernabéu-Aubán, P. W. Hutto, Y. A. Khalidi, M. Ahamad, W. F. Appelbe, P. Dasgupta, R. J. LeBlanc, U. Ramachandran: *The Architecture of Ra: A Kernel for Clouds.* In Proc. of the 22nd Annual Hawaii International Conference on System Sciences, Jan. 1.989.

[2] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, E. G. Sirer: *SPIN - An Extensible Microkernel for Application-Specific Operating System Services.* In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 68-71.

[3] N. Budhiraja, K. Marzullo, F. B. Schneider, S. Toueg: *The Primary-Backup Approach.* In S. J. Mullender Ed., *Distributed Systems (2nd edition)*, pp. 199-216. Addison-Wesley, Wokingham, England.

[4] D. R. Cheriton, K. J. Duda: *A Caching Model of Operating System Kernel Functionality.* In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 88-91.

[5] E. W. Dijkstra: *Cooperating Sequential Processes.* Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1.965.

[6] D. R. Engler, M. F. Kaashoek, J. W. O'Toole: *The Operating System Kernel as a Secure Programmable Machine.* In Proc. of the 6th ACM SIGOPS European Workshop, Wadern, Germany, Sept. 1.994, pp. 62-67.

[7] G. Hamilton, P. Kougiouris: *The Spring Nucleus: A Microkernel for Objects.* Proc. of the 1993 Summer Usenix conference, Cincinnati, June 1.993.

[8] Y. Khalidi, M. Nelson: *The Spring Virtual Memory System.* Sun Microsystems Laboratories Technical Report SMLI-93-9,

Mountain View (California), March 1.993, 23 pgs.

[9] H. P. Messmer: *The Indispensable PC Hardware Book (2nd edition).* Addison-Wesley Publishing Company, Wokingham, England, Apr. 1.995, 1336 pgs.

[10] F. D. Muñoz-Escoí, J. M. Bernabéu-Aubán: *The NanOS Object Oriented Microkernel: An Overview.* Technical Report DSIC-II/1/97, Univ. Politècnica de València, València, Spain, Feb. 1.997, 17 pgs.

[11] Object Management Group: *Common Object Request Broker Architecture and Specification.* OMG Document Number 91.12.1.

[12] U. Ramachandran, S. Menon, R. J. LeBlanc, Y. A. Khalidi, P. W. Hutto, P. Dasgupta, J. M. Bernabéu-Aubán, W. F. Appelbe, M. Ahamad: *Clouds: Experiences in Building an Object Based Distributed Operating System.* Technical Report, Georgia Institute of Technology, Atlanta, GA, June 1.989.

[13] F. B. Schneider: *Replication Management Using the State-Machine Approach.* In S. J. Mullender Ed., *Distributed Systems (2nd edition)*, pp. 166-197. Addison-Wesley, Wokingham, England.

[14] T. Stiemerling, A. Whitcroft, T. Wilkinson, N. Williams, P. Osmon: *Evaluating MESHIX - A UNIX Compatible Micro-Kernel Operating System.* In Proc. of the Autumn 1.992 OpenForum Technical Conference, Utrecht, The Netherlands, Nov. 1.992, pp. 45-58.

[15] L. van Doorn, P. Homburg, A. S. Tanenbaum: *Paramecium: An Extensible Object-Based Kernel.* Technical Report, Vrije Universiteit, Amsterdam, The Netherlands, 1.995.