Design of the Masix Distributed Operating System on top of the Mach Microkernel

Rémy Card, Hubert Lê Văn Gông and Pierre-Guillaume Raverdy

Laboratoire MASI, Institut Blaise Pascal, Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France

Abstract

Traditional operating systems have been designed as monolithic kernels implementing the whole set of the system services. This approach leads to important maintenance and evolution problems. This paper discusses a new design methodology. The operating system is implemented as a set of servers using the services provided by a micro-kernel. Splitting up the operating system into a set of servers leads to a modular and evolutive structure. The system exposed in this paper is made up of three layers. The host system provides services which allow the dynamic inclusion of new servers and different user interfaces. The Unix environment contains a set of servers which provide Unix abstractions and semantics. It is based on new and optimized implementations, using the Mach micro-kernel facilities. The virtual system is made up of a set of servers providing distributed features. These servers are dynamically included in the system by using the host system services. Numerous improvements included in the system imply performance optimizations in spite of context switches and messages exchanges needed to make different servers communicate with each others. The resulting structure is innovative and easy to extend. This system is the basis for future distributed operating system researches.

Keyword Codes: C.2.4; D.4.7
Keywords: Distributed Systems; Operating Systems, Organization and Design

INTRODUCTION

Masix[**1**] is a new operating system, currently under development at the MASI laboratory. Masix is a multi-environments operating system, i.e. it allows the parallel execution of different environments. Thus, different users may use different interfaces.

Our system has been designed with evolution in mind and allows the dynamic extension of its semantics while it is running. It is based on a multi-servers model which allows each component of the system to be written without interactions with the other ones. This also leads to a high level of maintainability.

Since Unix is a widely used system in research environments, the first environment provided by Masix offers a Unix compatible interface. This allows us to re-use the whole set of Unix programs.

Masix has also been designed to allow the optimal use of resources in a networked environment. Thus, it includes distributed extensions that optimize the use of a set of communicating stations.

In this paper, we first present the principles that we have used to design Masix. Then, we focus on the way the servers operate to build the local system. Last, we describe the distributed extensions included in the system.

1. DESIGN PRINCIPLES

1.1. System Structure

A current trend in operating system research consists in implementing the system in user mode with a minimal kernel. This kernel, called a microkernel, provides the resource management and some elementary services, e.g. physical memory management, process management... Examples of current microkernels include Amoeba[2], Chorus[3], Mach[4] and V-Kernel[5].

The design of the Masix operating system[6] has been based on the microkernel approach. We have chosen to use the Mach microkernel. Basically, Mach provides tasks, communication by messages through ports, devices and memory management[7]. Three kinds of operating systems have been built on top of Mach:

1. OSF/1 IK[8] has been designed as a traditional monolithic kernel. It uses Mach as a basis for its low-level layers.
2. Single-server systems, e.g. OSF/1 MK[9], implement Unix semantics in a single task which relies on Mach for the low-level operations.
3. Multi-server systems, e.g. CMU's Unix Multi Server[10], contain several tasks, which cooperate to provide Unix semantics.

We believe that it is worth designing an evolutive system in which components can be added in a dynamic way. Therefore, we have chosen to build Masix following the multi-servers approach. Implementing the operating system as a set of communicating user mode tasks offers many advantages:

- each server can be written and tested without interactions with the other ones,
- each server is smaller than a traditional kernel and, thus, easier to maintain,
- adding a server in the running system is easy if a dynamic service management has been implemented.

Nevertheless, the multi-servers approach has also its drawbacks:

- shared data are difficult to implement because each server has its own address space,
- the execution of system services often involves several servers and implies message exchanges and context switches. This leads to lower performances due to the context switch overhead.

Lots of optimizations have been included in the design of the system to minimize the number of servers involved in the realization of a system call.

System calls are implemented as remote procedure calls: a process sends a request message to a server and waits for the reply. The system contains three kinds of servers. Interface servers receive requests from the user processes. Internal servers provide features needed by other servers. Generic servers are contained in the host system and which provide services needed to run several environments.

1.2. Naming and Protection

In Masix, each server is responsible for providing a set of well-defined abstractions and semantics. The combination of the whole set of servers forms the complete operating system.

Each server implements the operations that act on typed objects. The server implements a fixed dialog protocol related to the type of the objects. When a client process needs to act on an object, it sends a request message to the server that manages the object. This message is sent on a Mach port, which is used as the object identifier. The server holds the reception right on this port.

Accessing an object only requires to know its port and its associated dialog protocol. The request sent to the object port is forwarded by the Mach microkernel to the server which manages the object. This communication scheme allows several servers, which manage the same objects types in different ways, to coexist in the system as long as they implement the same dialog protocol.

For instance, if a station running Masix mounts local and remote file systems, two file servers run in the system. The first server implements I/O operations on local files and the second one implements I/O operations on remote files. When a process accesses a file, it sends the same request wether the file is local or remote. This request is forwarded to the appropriate server.

Using Mach ports as objects references allows a high level of protection. To act on an object, a process needs to hold a send right on its port. When a process accesses an object for the first time, i.e. when it creates or opens the object, the server returns the send right if the calling process is allowed to act on the object. Otherwise, it does not return any send right. Thus, if the server has denied access to the object, the process cannot send any request since it does not hold any send right to the port.

1.3. Server Structure

In Masix, each server is a Mach task. This task contains several threads which wait for requests and execute them. Each object is named by a port and every object port is contained in a single port set. Every thread waits for requests on this port set. Requests are thus dynamically distributed between the threads contained in the server.

It is worth noting that this scheme implies synchronization between the threads. As the threads accesses the same data contained in the server's address space, each thread needs to lock the data it uses to avoid inconsistencies if it is preempted by the Mach scheduler.

1.4. System Calls and Library Functions

In Masix, system calls are implemented as remote procedure calls and imply messages passing and context switches. This scheme is much more expensive than the traditional one in which processes trap into the kernel to make a system call. Thus, to limit the overhead, we have worked to reduce the number of actual system calls. Many system calls have been moved as library functions which are executed in the calling process address space. These calls do not require context switching and are more efficient.

Of course, some system calls cannot be moved as library routines:

- Some calls need a global knowledge of the system state. Implementing them as library subprograms would require making these informations available to the calling process in its address space.
- The parameters of some calls must be checked by the system. To make the system safe, this check must be made by a separate server. This way, an erroneous or malicious program cannot modify crucial data.
- Some calls modify data shared by several processes. Since each process has its own distinct address space, the calling process cannot modify the data by itself. Thus, the data must be stored in a server address space and the processes must send requests to the server to read and update them.

1.5. Interactions between servers

Each server needs to maintain some informations relative to its objects. Two servers often need to use the same information. In a monolithic kernel, these informations are stored in common tables which are accessed by every component which needs to. In a multi-servers system, we cannot use global tables since each server has its own address space. In Masix, each server maintains a subset of the system informations. This subset contains only the informations that the server needs to know in order to manage its objects.

Sharing informations between servers is an important problem. We have distinguished three cases:

1. Read-only informations, e.g. a process identifier, are duplicated in each server which needs to know them. This way, each server can access the informations in its address space.
2. Informations that are more often read than modified are replicated in each server. When a server modifies an information, it sends its new value to the other ones.
3. Informations that are more often modified than read are managed by a server. Other servers must send requests to this server to access the data.

2. THE LOCAL SYSTEM

2.1. The Administration and Configuration Server

Masix is a multi-environments system. It includes an environment manager, which is responsible for keeping track of the environments running in the system, providing a way for the user processes to communicate with the environments, and sharing the system resources between the environments.

The Administration and Configuration Server (ACS) is the first server to run when Masix is booted. It is in charge of maintaining the list of the servers which run in the system. The ACS allows clients to dynamically communicate with the servers. For this purpose, it offers a name service: it holds the reception right on a global port known by every task and uses this port for answering the name requests.

When a server starts its execution, it registers itself with the ACS by sending a message on this port. This message contains a character string used as the name of the server, a flag, which is true if the server is essential for the system, false otherwise and a send right on its request port.

The ACS maintains the list of registered servers and can receive three kinds of requests:

1. register requests, sent by servers when they start their execution,
2. client requests, sent by the clients when they need to know the request port associated to a server,
3. unregister requests, sent by servers when they terminate their execution.

The ACS also monitors the system execution. It is responsible for ensuring that the essential servers are active in the system. If it detects the termination of an essential server, e.g. when a server crashes, the ACS sends termination orders to every server. When receiving this termination order, each server saves its data and terminates its execution. Then, the ACS reboots the system. This allows a system crash recovery.

2.2. Process Management

Masix provides an Unix environment, therefore it must implement processes with Unix semantics. Unix processes are emulated with the help of the Mach tasks and threads. A process is a task containing a single thread. Mach provides the internal process management, e.g. scheduling, context switches...

The process server provides the higher level process management. It maintains Unix like system informations, e.g. user identifier, and offers the process system call interface.

2.3. Signal Management

The process server is also in charge of emulating Unix like signals. In order to manage the synchronous signals, mapped as Mach exceptions, it holds the reception right on the threads exceptions ports.

When a signal, either synchronous or asynchronous, is sent to a process, the process server saves the thread state and sets up its stack to call the signal handler.

2.4. Memory Management

Masix contains servers that deal with memory management. These servers are external pagers and interact with Mach.

Masix contains two kinds of pagers:

1. a data pager, which deals with the swap management. This pager reads and writes pages belonging to the data and stack segments of the processes.
2. a code pager, which reads code pages. As long as a process does not modify its code, there is no need to save the code pages in the swap area and this server can read them from the executable files contained in the file system.

The data pager manages swap partitions. It uses memory objects to communicate with Mach: a memory object is associated with each data and stack segments. A set of blocks is associated with each memory object.

2.5. Files Operations

In Masix, we have chosen to integrate the file operations with the virtual memory subsystem. The files are managed by pagers used to load pages in memory.

When a process reads or writes data, the I/O operation is converted, by the library routines, to a memory access. Thus, the process reads or writes data in memory. If the referenced page is not loaded, Mach detects a page fault and calls an external pager which is responsible for loading the page from the file blocks. The pager is also called when a modified page is evicted by Mach and must be written back to the disk.

The file servers can receive two kinds of requests:

1. reads and writes requests sent by Mach,
2. file manipulation requests, e.g deletion of a file, sent by the client processes.

Masix is able to manage several file systems structures in a transparent way. It contains two sorts of file servers:

1. the Virtual File Server (VFS) receives the file based requests and calls the appropriate file server,
2. several Physical File Servers (PFS) implement the low-level operations on files. Each PFS is in charge of managing a certain structure of file systems, e.g. MS/DOS file system, System V file system, BSD Fast File System... The operations provided by a PFS are used by the VFS when it needs to access the physical data.

While the VFS can only be called by client processes, each PFS is also an external pager. When a process opens a file, a memory object associated with the file is created and Mach can call the PFS paging operations with the memory object as an argument.

3. THE VIRTUAL SYSTEM

Masix has been designed to allow the easy extension of the system by adding or replacing servers. The virtual system is built on top of the local system by using this possibility.

3.1. Distributed features

Masix includes distributed extensions. These features allow a transparent sharing of resources on a network of stations running Masix. This resource sharing is implemented by servers running on top of the local system. These servers communicate by using a network communication layer.

Resources sharing concerns:

- processors: process placement and migration allows us to balance the load on the network,
- files: distributed file systems allows processes to access remote files in a transparent way on every station,

- devices: in a network of stations, some devices are often in limited number. Thus, the system allows processes running on stations to access remote devices. This access cannot be done in a transparent way since the user has to know the station connected to the device,

- system informations: numerous system informations, e.g. user informations, must be shared between the stations.

3.2. Network Communication

Masix contains a networking layer which is in charge of maintaining the list of the stations connected to the network. This layer is also responsible for maintaining the list of remote servers used on the local station.

Masix uses local ports as identifiers of remote objects and servers. Each port related to a remote server is managed by a local interface server which forwards the requests that it receives to the remote server.

This way, client processes use an uniform communication scheme wether the server is local or remote. Both types of servers use Mach port for receiving the requests.

3.3. The Virtual System Configuration

In the Masix virtual system, the number of resources and most of all their types are very important. Therefore, the need for a configuration management system becomes obvious. This configuration management system manages both physical and logical resources. It also provides structures such as domains ([**11**]) and a directory to store informations. Those structures are also useful for the other aspects of management of system (i.e. fault management, security, ...).

Our management structure is composed of management nodes. Each node has a management agent which is responsible for all the resources attached to it. Decisions may be taken at the node level for the less important ones (e.g. modification of some unimportant state flags...) and at a higher level for the others. There are two ways for an agent to communicate with its superior after an extraordinary event.

- The first one is to send a trap to the higher level of the management hierarchy. The main advantage of this solution is to be the quickest one. However, there may be rapidly jams on the network if several importants events occur in short laps of time. Whatsoever the trap sent by the node has to contain enough information for its superior[12].

- The other solution uses a polling approach. It has the advantage of avoiding network traffic jam because the superior periodically queries its lower level nodes about their resources (as some AI based management systems do). The problem is to determine when and how often it must do it.

To solve this problem, we have introduced a notion of priority level scale for each resource of the system. The level accorded to a resource depends on the impact the resource may have on the distributed system. For instance, the environments supported by Masix are of high level whereas a standalone machine or a printer has a low level.

Thus resources with a high level priority are not queried because the node which manage them knows those resources will send it a trap if an extrardinary event occurs. Therefore the polling-based approach just involves low level priority resources. Those priorites have to be defined at during the initalization of the management system and are dynamically modifiable.

The notion of priority scale is managed by a server which is a distributed counterpart to the local ACS.

3.4. Distributed Files

In Masix, files operations are integrated in the virtual memory subsystem as explained in Section 2.5. Adding a new file system type consists in launching a new PFS, which is responsible for managing the files.

Remote files are managed by a local PFS. The physical file access is done by the PFS by forwarding the I/O requests to a distant file server (DFS), which runs on the station that owns the file system on its local disks. The DFS is in charge of physical I/O on files: it replies to the requests sent by the PFSs on the client stations.

The DFS is also responsible for maintaining the files consistency. Since several processes running on different stations can access the same files, the system must ensure that the latest version is always used. The DFS maintains a table of open files and, for each file, the list of blocks sent to remote stations. When it receives a block read request, the DFS defers the reply if the block has already been transmitted to a process which holds write access to the file or if the request concerns a file open in write mode and the block has already been transmitted to another process.

Actually, the DFS uses a token mechanism to protect the file access. It manages a token for each file block. This token can be distributed to several processes if the file is open in read-only mode but only one token can be sent if the file is open in write mode.

Since the DFS maintains a state concerning the open files, a recovery algorithm is needed in case of a crash. Each PFS also maintains the list of the tokens. If the DFS crashes and restarts its execution, it broadcasts a request to every PFS and rebuilds its state from their replies.

3.5. Distributed Tasks Management

A dynamic task management (DTM) allows applications and users to benefit of all the processing power over the network and to have a better reliability on the system. To achieve this, each host needs a view of the system's global state. Hosts also need to take distributed agreement about their choices to improve system performance. Last, they need an efficient process transfert mechanism.

The basic characteristics of our DTM are a full transparency to users, migration of tasks for load balancing and a fault tolerance objective. Our DTM provides a set of services like information exchange between hosts (e.g. load, state) or naming and location transparency of entities over the network.

Processes in a system are never executing alone. They interact with other processes that build up an application and with the resources managed by the system. Since the migration of a process may influence on the performance of the others, we have defined processes group, which are based on relations between processes.

With non-preemptive placement, processes groups are easy to define with statistics obtained during previous executions or with hints provided by the user. On the contrary dynamic processes group definition is hard to obtain because hosts only have a partial view of the system. Definition of processes group for load distribution induce the definition of forces between processes (attraction-repulsion [13, 14]).

We define different groups of processes, based on memory and CPU usage, communications and devices access. Each group has a major gravity centre and fixed points. Distance between remote processes being in the same group can be evaluated. This distance represents the ratio cost/advantage of migrating them on another host.

We extend the multi-criteria algorithms defined in [13] to a dynamic system. We use the same refinement technics by using filters linked to the bottlenecks of each host. Static constraints (e.g. the binary architecture) allows us to first define the migreable processes. Thru the refinements each host computes a set of eligible processes. After a distributed agreement, the group of processes is migrated. The order in refinements is defined by the resources that will better improve the overall host performances.

Our migration mechanism concepts are :

- We define four migration states depending on the local load (acceptance, inactive, placement, migration).
- There are two migration axes. For the hosts based on multicriteria load (i.e cpu, memory, network, processes queue) and for each process based on multicriteria forces (i.e. memory usage, cpu utilization and system calls forwarded to the Mach microkernel, local or remote servers).
- Bottlenecks on each host define the refinements order in migreable process selection.
- Local set of inactive hosts is managed. This set is used for finding a destination host. The hosts belonging to this set send their load to the others. Each host can enter the set if its load is lower than the last one in the set.

The cost is the sum of the migration mechanism and the overhead due to original host servers access. The advantage is the performance speed-up for the migrating process and for the local process.

Knowing host load is not enough to ensure an efficient migration policy. Informations on the network's use are needed[15]. The gain for the process can be evaluated in term of number and size of messages exchanged and by the time the process has waited for replies from the remote servers. For each communicating object and for different periods of time, we maintain the communications statistics. By knowing which hosts are concerned by this objects we can evaluate the most interesting host for migrating a process in a local point of view.

4. CONCLUSION

We have described the Masix operating system. By using the multi-servers approach, Masix has been designed as a very evolutive and easy to maintain system. The system contains lots of optimizations needed to reduce the overhead due to the context switches and messages passing related to the multi-servers architecture.

The Masix multi-servers architecture allows the system to be dynamically extended during its execution by replacing or adding servers. This feature is used by the servers providing the distributed functionalities.

REFERENCES

1. R. Card, E. Commelin, S. Dayras, and F. Mével. The MASIX Multi-Server Operating System. In *OSF Workshop on Microkernel Technology for Distributed Systems*, June 1993.
2. S. Mullender, G. van Rossum, A. Tanenbaum, R. van Reneese, and H. van Straveren. Amoeba – A Distributed Operating System for the 1990s. *Communications of the ACM*, 15(5):44–53, May 1990.
3. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems*, 1(4):305–370, December 1988.
4. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the USENIX 1986 Summer Conference*, June 1986.
5. D. Cheriton. The V Distributed System. *Communications of the ACM*, 31:19–42, 1988.
6. R. Card. *MASIX - Un Système d'Exploitation Multi-Environnements utilisant le micro-noyau MACH - Conception et Réalisation*. PhD thesis, Laboratoire MASI, Université Pierre et Marie Curie, May 1993.
7. K. Loepere, editor. *Mach 3 Kernel Principles*. Open Software Foundation and Carneggie Mellon University, 1992.
8. *The Design of the OSF/1 Operating System*. Open Software Foundation, 1990.
9. F. Barbou Des Places, P. Bernadat, M. Condict, S. Empereur, J. Febvre, D. George, J. Loveluck, E. McManus, S. Patience, J. Rogado, and P. Roudaud. Architecture and benefits of a multithreaded osf/1 server. Technical report, Open Software Foundation Research Institute, 1992.
10. P. Guedes and D. Julin. Object-Oriented Interfaces in the Mach 3.0 Multi-Server System. In *Proceedings of the IEEE Second International Workshop on Object Orientation in Operating Systems*, October 1991.
11. A. Langsford. Evolution of the Managed Network. In *Networks '92*, June 1992.
12. M.T. Rose. *The Simple Book, An Introduction To Management of TCP/IP*. Prentice-Hall, 1991.
13. B. Folliot. *Méthodes et Outils de Partage de Charge pour la Conception et la Mise en Oeuvre d'Application Parallèles dans les Systèmes Répartis Hétérogènes*. PhD thesis, Laboratoire MASI, Université Pierre et Marie Curie, December 1992.
14. M. Tokoro. Computational Field Model: Toward a New Computing Model/Methodology for Open Distributed Environment. In *Proceedings of 2nd Workshop on Future Trends in Distributed Computing Systems, Cairo*, September 1990.
15. D. Milojičić, P. Giese, and W. Zint. Load Distribution on Microkernels. In *Proceedings of the IEEE Workshop on Future Trend in Distributed Computing Systems*, 1993.