Experience with Chorus

Christian Bac, Guy Bernard, Denis Conan,
Quang Hong Nguyen, Chantal Taconet

Institut National des Télécommunications
9 rue Charles Fourier, 91011 EVRY Cedex, France

Abstract. This paper summarizes works done at I.N.T. with Chorus [1] Operating System. It briefly describes Chorus' concepts and abstractions useful to understand our work.
Then it focuses on Chorus micro-kernel capabilities and explains how these capabilities have been used to make an experiment allowing the cohabitation between the Chorus micro-kernel and the Macintosh Operating system.
Then, it describes how a new subsystem can be built over the micro-kernel. As an example, it shows how a subsystem which emulates the Macintosh Operating System has been built.
Then, it explains how new capabilities can be integrated into an existing subsystem. To illustrate this point, it gives two examples of work we are currently doing on Chorus/MiX running on PCs. The first project adds "Quality of Service" support for distributed multimedia applications; the second one allows "Fault Tolerant" aspects to be taken into account in distributed applications.
Finally, we discuss some limitations of Chorus, especially in supporting large networks, and how the system should be extended to address this new feature.

1 Introduction

The "distributed operating system" research group of INT has been working with the Chorus Operating System, since 1989. The first experience was the port of a Chorus simulator in the A/UX [2] environment. The Chorus simulator is composed of a UNIX process and a library that can be used to program and test Chorus applications in a UNIX environment. This port was done to gain experience in the Chorus area. The simulator was later used to develop graphical applications using the distributed IPC of the Chorus system in an attempt to anticipate future research. Then we ported a Chorus kernel to Macintosh hardware, and we made the two operating systems (Chorus and MacOS) cohabit and cooperate [Bac93].

After this we built a new subsystem [Bac94] that encapsulated the Macintosh Operating System and allowed Chorus active entities (called actors) to use the MacOS operating system interface. The resulting subsystem is composed of two

[1] Chorus is a registered trademark of Chorus Systèmes.
A/ X is a version of NIX for Macintosh hardware.

actors: a Supervisor actor and a Server which executes at the user level. The supervisor actor handles MacOS system calls and interrupts.

Presently, we are working in three different directions with Chorus:

- the first adds support for quality of service [Nguy95] to handle multimedia communications;
- the second addresses the problem of fault tolerant distributed computing in networks of workstations [Bern94];
- the third focuses on some limitations in the localization service [Taco94].

This paper is organized as follows:

- Section 2 briefly describes Chorus' concepts and abstractions, as well as notions about subsystems.
- Section 3 focuses on some Chorus micro-kernel capabilities and explains how these capabilities have been used to allow the cohabitation between the Chorus micro-kernel and the Macintosh Operating system.
- Section 4 shows how a subsystem which emulates the Macintosh Operating System has been built over the Chorus micro-kernel.
- Section 5 explains how we can integrate new capabilities in an existing subsystem. In this part we explain how quality of services and fault tolerance may be added to a Chorus/MiX subsystem.
- Finally, in section 6, we discuss some limitations of Chorus, especially in supporting large networks.

2 Chorus

A Chorus System is composed of a small-sized Nucleus and of possibly several System Servers that cooperate in the context of subsystems to provide a coherent set of services and user interface. The interfaces [Walp92] to the subsystem and to the kernel are described in Figure 1.

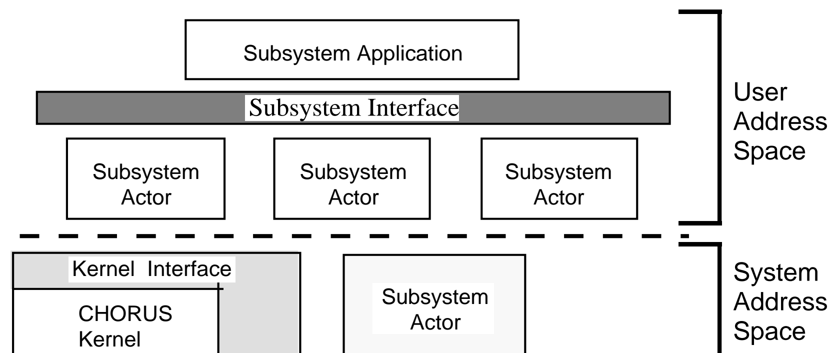

Fig. 1. Chorus Subsystem Interfaces

A Chorus domain consists of a set of Chorus sites. Each site hosts the Chorus Kernel and some actors. A detailed description of the Chorus system and of the Chorus/MiX UNIX subsystem can be found in [Rozi88].

2.1 Basic Abstractions

The Chorus kernel provides the following basic abstractions (see Figure 2):

- The actor defines an address space that can be either in user space or in supervisor space. In the latter case, the actor has access to the privileged execution mode of the hardware. User actors have a protected address space.
- One or more threads can run simultaneously within the same actor. They can communicate using the memory space they share.
- Threads from different actors communicate through the Chorus IPC that enables them to exchange messages through ports named by global unique identifiers.
- The Chorus IPC provides the ability for one thread to communicate transparently with another thread, regardless of the nature or location of the two threads.
- Multi cast is achieved through group communications. Ports can be inserted into “port groups”. On the sender behalf, messages that are sent to a port group can reach all the ports, one “random” port, or one port targeted to a machine.

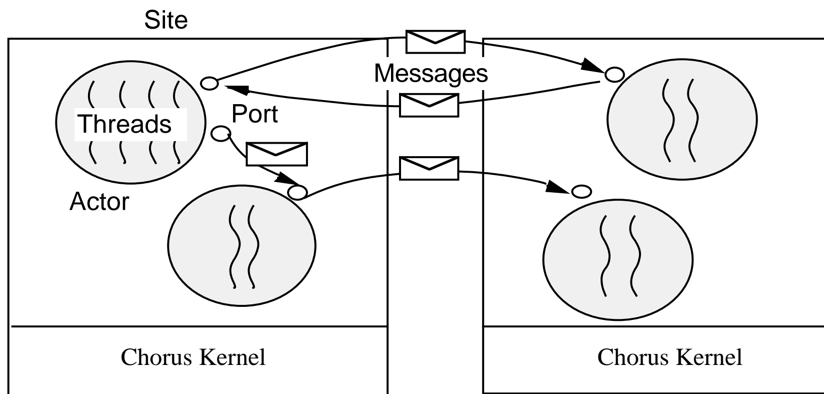

Fig. 2. Chorus Basic Abstractions

2.2 Chorus kernel design

As shown in Figure 3, most of the Chorus kernel [Rozi93] is portable. It is composed of four independent elements:

- The supervisor dispatches the interrupts, traps and exceptions delivered by the hardware.
- The real-time executive controls the allocation of the processor.
- The virtual memory manager is responsible for manipulating virtual memory hardware and local memory resources.
- The inter-process communication manager provides message passing facilities.

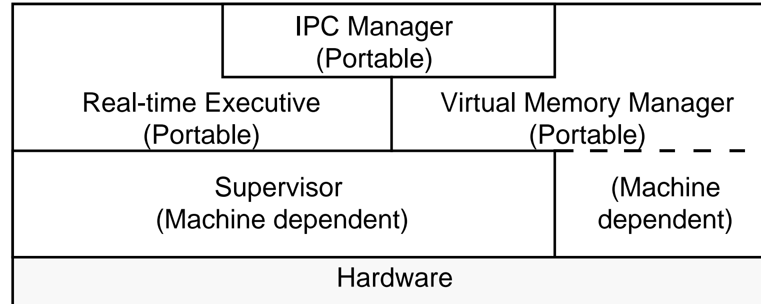

Fig. 3. Chorus Kernel Architecture

2.3 Kernel Services to Subsystems

Chorus kernel allows user or system threads to take control of exceptional events within an actor.

For supervisor actors, the kernel provides facilities to connect hardware interrupts to interrupt handlers within the actors. When an interrupt occurs, the kernel sequentially calls a prioritized sequence of user routines associated with the given interrupt. Any of the individual routines may initiate a break in the sequence if necessary.

The supervisor supports a similar facility, through which actor routines may be associated with hardware traps or exceptions. This facility allows subsystem managers to provide efficient and protected subsystem interfaces.

3 Chorus micro-kernel capabilities

This section tries to explore some of the technologies that allow the Chorus Kernel to be adapted to new configurations. The first part deals with some aspects of porting Chorus to new hardware and the second deals with features like interrupt relays in the kernel.

3.1 Some aspects of Porting Chorus to new Hardware

The target machine was a Macintosh [3] II CX. It is based on a MC68030. We used the Chorus sources for the MC680X0 0. In this area our work was simpler than the one usually done to port a kernel to new hardware.

Chorus hardware requirements Chorus hardware requirements are small. In order to run, the kernel needs a timer that performs an interrupt every 1/100th second, and a terminal interface that reads and writes characters. It also requires correct initialization of the memory management unit (MMU). Timing has been achieved by programming a timer from a Versatile Interface Adaptor (VIA). We decided to connect the terminal to a port from the Serial Communication Controller (SCC). The low level routines allowing Chorus to control the port were written in C.

Translation tables In general, the management of the memory management unit must be written; in our case the greater part had already been done, due to the sources used. Initialization of translation tables remained the main problem.

Chorus on MC68030 uses a virtual address space composed of 4 K Bytes memory pages and four levels of memory index (called levels A to D). Translation tables are set up at boot time. When Chorus is running, the first level A entry describes the actor in user space, and the second level A entry describes the Chorus kernel space. This space includes actors running in supervisor mode.

In the Mac version, the other level A entries are reserved for the kernel and map the physical address space allowing the kernel to have access to the I/O space and extension bus.

Conclusions on Porting Chorus The use of the Macintosh hardware by Chorus was limited to SCC and VIA. SCC was used to transmit and receive characters. VIA was programmed to emit clock ticks. Despite complexity of the Macintosh hardware (which is beyond the scope of this paper), Chorus has proved to be easily portable on it.

3.2 Features used to make Chorus and MacOS Cohabit

To allow Chorus and MacOS cohabitation, we changed the way Chorus managed the hardware to keep the MacOS system able to run. Conversely, we modified the boot phase so as to present Chorus as an application to MacOS. The modifications allow the two systems to share the memory space and to manage the interrupts.

[3] Macintosh, MacOS, and A/ X are a registered trademarks of Apple Computers, Inc.

Sharing memory space As the MacOS operating system needs the lower and the higher part of the physical memory, Chorus kernel access to memory is restricted to 6MB from address 0x80000. This is possible because Chorus can behave as if memory does not begin at physical address 0, and also because the kernel gets the memory size from the boot program.

The boot program asks MacOS for a 6.5MB fixed block of memory. There are small amounts of memory which are 8 in this process but this allows changes in both versions of system.

In this way, the physical memory is separated in three areas (see Figure 4 above):

- the memory reserved for the MacOS system (composed of two parts),
- the memory reserved for the Chorus system (for the kernel and actors),
- and a third area that first allowed a correct alignment of memory space and that was later used to go from one system to the other. This space is called "No Man's Land".

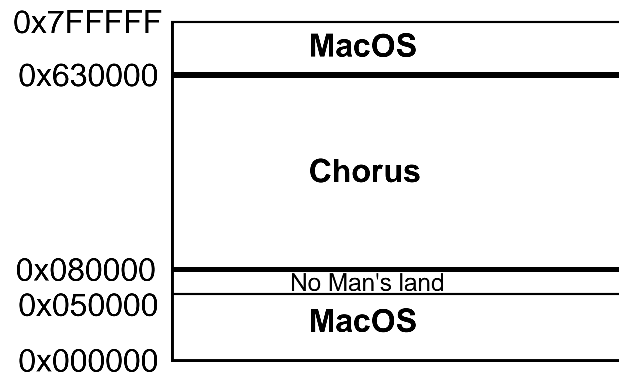

Fig. 4. Sharing memory space

In this configuration, the MacOS system is still able to run, and Chorus can stop and return back to MacOS as if Chorus was a Mac application.

Sharing system events In order to let MacOS control some types of interrupt, low level functions were added. These functions enable the CPU to commute from Chorus system to MacOS, so that Macintosh can run either system. These functions change the MMU setting and the value of the interrupt base register.

As Chorus can connect an interrupt to multiple tasks, interrupts can be connected to a handler in Chorus, a handler in MacOS or a handler in both. To call a MacOS handler, there is a standard function in the "No man's land" that creates an interrupt frame on MacOS stack. This jump into MacOS allows it to acknowledge the interrupts correctly.

Events in MacOS For preserving the MacOS feeling, the preboot program (MacOS space) has been modified so as it can perform some events every 1/50s.

This allows MacOS back to perform many operations:

- mouse moving is detected and mouse pointer is displayed on the screen,
- accessories like Clock and Super Clock display correct time,
- mouse position can be read from the program,
- and keyboard events are recognized.

At this point, the two systems are present in memory and they can share system events like interrupts.

3.3 Experience

Chorus kernel contains features that helped making it cohabit with MacOS. In our last version, both systems co-operate so that Chorus uses MacOS to read from the keyboard and write to the screen. To achieve this co-operation, the boot program in MacOS system space acts on behalf of the Chorus system. A part of the "No Man's land" is used as a buffer to exchange characters between Chorus and the boot application.

4 Building a new Subsystem

In this section, we try to show how a subsystem can be built. To illustrate our purpose, we take the Macintosh Operating System as an example. In this example, the subsystem reuses the MacOS software providing a Macintosh interface to Chorus actors.

Chorus Toolbox Design As shown in Figure 5, the Chorus Toolbox is a small subsystem consisting in two actors. As in every subsystem, there is an actor running in supervisor mode that handles system calls and interrupts. This actor is called **ChorusTbxSup** (later called Supervisor) and relays MacOS system calls as well as hardware interrupts. It provides a MacOS interface to user level actors.

In complex subsystems as shown in Figure 1, there are specialized actors that handle parts of the subsystem work. For example in Chorus/MiX [Rozi88], the Process Manager manages processes, the Object Manager manages the file systems, and the Stream Manager handles communications.

In the MacOS subsystem, there is only one actor that acts as a server. It is called **ChorusTbxServer** (later called Server) and runs at the user level. It calls the MacOS routines in RAM or in ROM.

The Supervisor and the Server actors communicate by means of the Chorus IPC. Exceptions resulting from MacOS system calls are caught by the Supervisor. The Supervisor builds a message and sends it to the Server.

There is an initial negotiation between the Supervisor and the application. This negotiation is used to map the application memory space into the server address space allowing a greater flexibility to exchange parameters and results.

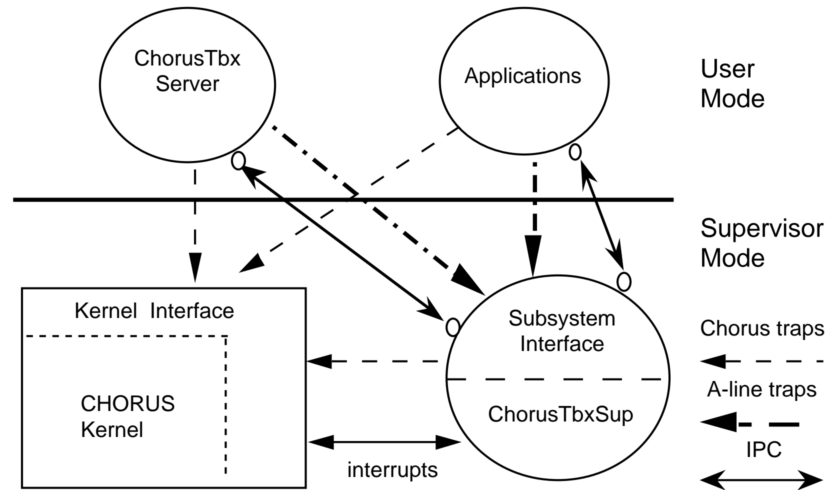

Fig. 5. ChorusToolbox subsystem

Supervisor Design This actor executes at the supervisor level. It handles the low level part of the MacOS system calls and the interrupts. It contains the following threads:

- The main thread initializes the subsystem when it starts. It connects exceptions and interrupts handlers. Then it waits for an application to start a MacOS session. When the application asks for MacOS operation, it sets the server memory mapping so that the server can use the memory segments of the application directly.
- The other thread is called **TbxSupIt**: it relays the interrupt handlers messages to the server. To avoid deadlocks, interrupt handlers are not allowed to use IPC. They must use a restricted form named mini-messages. Mini-messages are allocated in the actor memory and can be sent to a thread in the same actor. So interrupt handlers send mini-messages to TbxSupIt and TbxSupIt sends a message to the Server to service the interrupt.

Server Design This actor executes at the user level. It handles requests to Macintosh Operating System. It also encapsulates the MacOS interrupt handlers into separated threads. It contains the following threads:

- The **TbxServerInit** executes once when the Server starts its execution. It creates the communication ports, sends identifiers to the Supervisor actor allowing it to modify the Server address space and to send messages to the Server ports. It then launches the other threads (described below), and finally ends allowing them to execute.
- The **TbxStdAline** thread waits for messages describing system calls. It analyzes the system calls and launches a thread executing at the address of the MacOS routine. As will be explained later, this new thread uses the

application stack to avoid any copy of parameters and results. When the routine ends, the TbxStdAline sends results back to the supervisor which resumes the initiating thread.
- The **TbxStdIt** thread waits for messages describing interrupts and awakens the corresponding interrupt handler. For some interrupts, it creates one of the three threads described in the next item.
- Three threads serve interrupts that take a long time or may need to call many of the MacOS system calls.

Experience We were able to use many of the Chorus kernel features to integrate MacOS in a subsystem over Chorus that gives a MacOS interface at the user level applications.

Memory management features were especially useful for our purpose. The mapping of the application' memory in the Server address space gives a vision close to the one used in Macintosh Operating System. This mapping also allows quick and efficient exchange of data between the application and the Server.

Interrupt and exception handlers were also easy to add to the system. In order to allow MacOS interrupt handlers to execute at the user level, the subsystem catches the privilege violations and emulates the corresponding instructions.

5 Adding Services

In this section, we show how new capabilities, such as support for Quality of Service and Fault Tolerance, can be added to Chorus/MiX.

5.1 Quality of Service

We present an Object Manager (OM) able to support Quality of Services as described in [Nguy95].

Access Time Problems of OM OM is an independent system server of the Chorus/MiX [Rozi88]. It acts both as a *swap device* for virtual memory management and as a file server. The other system managers communicate with it exclusively by message exchanges in order to access data segments.

OM implements a UNIX System V file system semantic [Bach86]. It uses the same data structures and algorithms as in the standard UNIX System V File System.

In data structures and algorithms used by OM, we identify the three following reasons that make the I/O time for a data block indeterminate [4] (and unbounded):

[4] we only consider the block data reading case.

1. The *layout* of UNIX file data blocks makes the time to read a data block not homogeneous. UNIX file data blocks are organized into levels using indirect blocks that content pointers to next level indirect blocks or to real data blocks. As there are 3 indirection levels, the file system may read 1 to 4 disc blocks for a data block.
2. The *cache management policy* does not allow an estimation of the reading delay upper bound. The server manages a pool of cache buffers, and allocates a buffer from the cache before transferring a data block. It must wait if either the selected buffer is found in the pool but is busy, or if it isn't found but the free buffer list is empty.
3. The *strategy algorithm* used by the driver can delay the queuing time for block reading in order to optimize the magnetic head movement. This means that the handling order of block read requests at the disk driver is not the same as the arrival order. Due to *strategy* the queuing time at the disk driver is indeterminate.

Therefore, the standard UNIX system file is not suitable to support QoS requirements of multimedia applications. However, we have identified the sources of non-determinism in the file management. We will now show how to overcome this non-determinism.

QoSOM design We propose modifications of algorithms and data structures used by OM so that the resulting OM, called QoSOM, will have a deterministic behavior behind standard UNIX system file characteristics. Our first version only considers the *continuous reading* of multimedia streams which are stored as UNIX files.

The QoSOM can run in normal or QoS mode. In normal mode, it uses standard algorithms and data structures. When being switched in QoS mode, it uses new algorithms and abstractions as follows:

We define the *trace* of a file being read as the path composed of 0 to 3 indirect block(s) that leads to the current data block. In standard UNIX, the algorithm called *bmap* is used to do the mapping between the logical and physical block numbers of the current data block. This algorithm finds the *trace* leading to the current data block in order to extract the physical block number of this later. As the indirect block(s) are freed after use, then next mapping may have to read them again.

We propose an enhanced algorithm for *bmap*, called *cbmap* (standing for *continuous bmap*), that keeps the *trace* during a *continuous reading*. Using this algorithm, it is necessary to get indirect blocks only when the *trace* must change. The number of indirect blocs to read and the moment when they must be fetched from disk can be computed.

QoSOM applies a *FIFO strategy* in QoS mode. As QoSOM is an independent server and as all data requests go through a specific *port*, the processing order of I/O requests is easy to control. *FIFO strategy* means that the server handles one request at a time. Although this strategy penalizes the overall performance,

it allows to determine the time needed to process a read request. This strategy also excludes the cache buffer race problem that occurs in the standard strategy.

In QosOM, we use a new abstraction, namely *partial_read*. A partial read is a read request associated to a *quota*. The *quota* describes the number of disk blocks that the server can fetch during the request. *Partial_read* and *Trace conservation* allow to split the reading of an application buffer into steps. This is useful for the *QoS Manager* to schedule the reading of multiple continuous media as described below.

QoSOM System Model The interaction scheme between the applications, the QoS Manager (QoSMgr) and the QoSOM is shown in Figure 6. The QoSMgr negotiates with the applications in order to set up QoS contracts. It controls the QoSOM function mode. The QoSOM handles two sources of requests: the non-QoS I/O queue, and the QoS scheduled requests stack. When QoSOM is in QoS mode, it serves the QoS stack first, and it serves standard requests only if there is "time" quotas remaining.

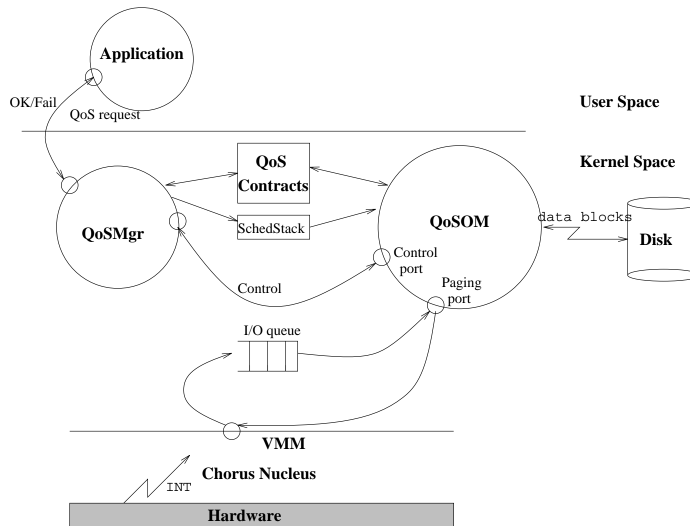

Fig. 6. QoSOM System Model

The QoSMgr achieves the *admission control* on the basis of *resource capacity*. Capacity for disk controllers is expressed by the number of blocks that the controller can fetch per second. Based on the application QoS specifications (rate, buffer size), the QoSMgr computes the *application required capacity*. It grants or denies access to the application, and before granting access, the QoSMgr reserves the necessary cache buffers.

The QoSMgr controls the correct execution of contracts on the basis of

rounds. A *round* is the time interval that is equal to the smallest common multiplier of all contracted stream periods. A new stream can only be started at the beginning of next round. Before each round, the QoSMgr computes the read scheduling for this round and pushes it onto the *SchedStack*. The QoSOM pops scheduled QoS requests from *SchedStack* and executes them.

The QoSMgr computes the scheduling stack using the *partial_read* abstraction. The algorithm, namely *StmSched*, for this computing is relatively simple. However, two interesting results are deduced from it: *(i)* all application data buffers are available before their deadline, and *(ii)* each stream needs two buffers for continuous reading (i.e. one in use by the application and the other in use by QoSOM – this is the minimum buffer number).

Conclusion on QoS We are now achieving the implementation of an experimental version of the QoSOM and the QoSMgr under Chorus/MiX v4 and Chorus Kernel v3 r4.2. The modularity in Chorus/MiX makes it easier to implement new servers and to modify existing ones.

5.2 Fault Tolerance

We address the problem of fault tolerant distributed computing in networks of workstations, where distributed applications consist in several processes running on several workstations in parallel, with communication links between them. Several methods can be used to provide fault tolerance in distributed systems. Using a specialized hardware may be efficient, but such a component cannot be easily added to existing systems. Application-specific methods and atomic transactions require the use of a particular programming model. Active replication is well suited for real time systems but require the use of extra processors. We propose to handle machine failures using checkpointing, message logging and rollback-recovery [Stro85, Borg89, John89, Elno93]. When a machine failure is detected, a replacement machine is found in the network. Practically, a checkpointing algorithm registers from time to time the state of the distributed application, a message logging algorithm saves the history of the execution in stable memory and a recovery algorithm restarts the distributed application from a previous state in case of a machine failure.

UNIX version A fault tolerance fully portable software has been implemented on a network of SUN workstations and demonstrates the intrinsic limitations of monolithic operating systems [Bern94]. The functionalities where implemented in a daemon process running in user space. This daemon acts as an intermediary between the processes for the distributed applications for their communication. When an application process has a message to send to another application process, the message is first passed to the daemon process that performs the appropriate tasks related to fault tolerance (logging, checkpointing, recovery). On the receiving side, messages are buffered by the daemon until being requested by application process. Primitives for communication between remote part of the

distributed application are implemented in a library that is to be linked with programmer's source modules. As could be expected, fault tolerance is expensive because of communication operations. So, we decided to move the whole software in the Chorus UNIX subsystem, i.e. Chorus/MiX. In the following, we study fault tolerance enablers of the Chorus micro-kernel and present the overall architecture of the software.

Prerequisites We found six prerequisites to define what could be a good support for designing and implementing a rollback recovery mechanism. For each of them, we compare the standard (Berkeley) UNIX system - here, called UNIX - with the Chorus micro-kernel. The four first elements are common to migration mechanisms [Alar92].

1. A location-transparent inter-process communication service
   In UNIX, whereas an elementary form of naming service (binding between service names and process ports) is provided, the binding scope is only local to a machine. In Chorus, port' names are global but when a machine failure occurs the localization service broadcasts a request in order to find the new locations of the ports. Because this mechanism is too expensive, we need the concept of reliable ports [Gold90].
2. A global naming scheme for every object handled by the system
   In UNIX and Chorus, most external and internal names are local, particularly process identifiers. Some of these names are very important for interactive and real time applications. Thus, we don't support those applications in a first step.
3. An implementation of services via system server processes remotely accessible
   In UNIX, system calls are procedure calls (without message passing) and local by definition. In Chorus, client processes (either user processes or system processes) can communicate with system server processes via messages on the network.
4. The state of processes easily accessible and grouped in a single place
   In UNIX, the state of processes is disseminated in several parts of the kernel space and most of the information is only relevant to the local machine. In Chorus/MiX, the Process Manager contains the whole state of processes.
5. A reliable and ordered communication service
   The UNIX reliable and ordered communication service is TCP. TCP is also available with Chorus/MiX.
6. A multi-cast communication service
   In UNIX, there is no group functionality, i.e. there are only broadcast messages. The Chorus micro-kernel offers group communication primitives with several addressing mode through the concept of port group.

Therefore, Chorus is a good candidate for the support of a rollback recovery mechanism.

Architecture Now, we present the overall architecture and next show which of the Chorus functionalities are the most useful for the design of each part of the rollback recovery mechanism.

As shown in Figure 7, the fault tolerance software is implemented as a server - the Fault Tolerance Manager, FTM - added to Chorus/MiX. All FTMs have the same role, except one called the controller, which has a complete view of the distributed applications. The controller maintains port groups. Each distributed application is associated to two port groups: the first group is composed of FTMs ports corresponding to the sites where the application is spread, the second group is composed of the control ports of the processes.

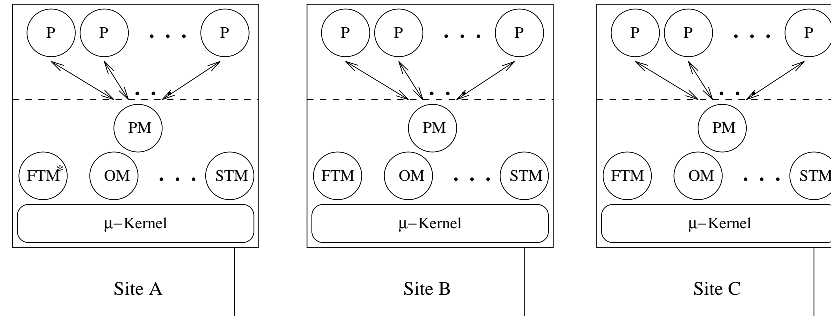

Fig. 7. FTM in Chorus/MiX

As FTMs are parts of the UNIX subsystem, the associated ports are default ports. Groups help in controlling checkpointing, logging and recovery of the distributed applications.

Since processes data structures are managed by the PM, it must checkpoint these structures. Checkpointing is very similar to the one described in [Phil95]. PM transmits the data structures to the FTM, which must end the checkpoint, by saving the process virtual memory and data structures in stable memory. Indeed, inhibition of PM due to checkpointing must be as short as possible, because the PM has to trap every process system calls. Saving of the virtual memory is straightforward because the micro-kernel implements copy-on-write [Abro89] operations. So, we only have to stagger in time the checkpoints by the FTM in order to minimize the storage overhead.

At the beginning of its execution, a process establishes connections with other processes. The STM informs the local FTM, the local FTM constructs a reliable port and transmits it to the controller which saves it in stable memory. The contents of inter-process messages are logged in the volatile memory at the sender site. Stamps (sender identifier + sending sequence number + receiver's identifier + receiving sequence number) are logged in volatile memory at the receiver site, and then regularly transmitted to the controller. Finally, the controller is responsible for saving the stamps in stable memory. The STM captures the

message to be sent and transmits a copy of the message to the local FTM that replies by giving the sending sequence number. At the receiving side, the STM notifies the new arrival to the FTM, the FTM stamps the message and later transmits the stamp to the controller. Therefore, the only modification we add in Chorus/MiX is a dialogue between the FTM and the STM on the sending s and receiving of inter-process messages.

When the STM detects a communication fault, it informs the local FTM, the local FTM informs the controller and the controller broadcasts a warning to all other FTMs. The controller takes the control of the distributed application and is responsible for the recovery. It asks FTMs to reinstall the state of failed processes, then to reconnect failed processes and reconfigure fault-free ones according to reliable ports saved in stable memory, and finally to restart all processes. Thus, FTMs cooperate with PMs and STMs for the recovery.

Conclusion on FT In conclusion, the design of a rollback recovery requires six functionalities. Chorus already implements four of them : system servers remotely accessible via message passing, processes' state grouped in a single place, reliable ordered communication service and multi-cast communication service. To those, we add the concept of reliable ports. More work has to be done in order to support more realistic processes using signals, file and terminal connections. For instance, in [O'Co94], the authors wrote a terminal server to buffer all data sent between actors and their associated terminals.

6 Limitations and extensions

Chorus offers distributed system developers key tools for distribution (modularity of global systems, performing communications, transparent localization of entities), but so far these capabilities have been limited to local area networks.

Capabilities of large scale networks will undoubtly increase significantly in the next years. High speed, up to now reserved for local area networks, will soon be available in wide area networks. A lot of applications which could be used in the context of local area networks only will be usable in wide area networks. These observations lead us to think that microkernel systems must be modified to fit to large scale networks.

In order to make Chorus work upon a large scale network, we propose a localization mechanism which would operate over large scale networks.

We first describe the current localization mechanism in the Chorus microkernel, and then present an extended localization service, and its implementation in Chorus.

6.1 Current localization mechanism in Chorus

Each Chorus resource (port, ports group, site, actor) is identified by a Unique Identifier (UI). An UI is made up with *(i)* the resource's creation site address,

(ii) an identifier of the type of resource, *(iii)* and a unique stamp provided by the creation site's micro-kernel. Some stamps are well known stamps which may be used by any site.

Chorus sites are grouped into domains. IPC (Chorus Inter Process Communication) are implemented above network protocols which provide a broadcast facility. A Chorus domain is typically a local area network, or a multiprocessor computer. Protocols used by Chorus are organized in an oriented graph. The micro-kernel chooses dynamically a way in this graph for each message.

The current localization mechanism of Chorus resources can be broken down into several stages [Syst95a]: given an UI, *(i)* the micro-kernel looks for a resource identified by that UI on the local site, *(ii)* the micro-kernel looks for a localization information in its localization cache where the last localizations it found are stored, *(iii)* the micro-kernel uses some hint such as finding a port on its creation site, *(iiii)* if the hint does not work, the microkernel sends a broadcast message on the domain in order to find the entity.

Localization does not go any further than the limits of the domain.

6.2 An Extended Localization Service

We propose a method for propagating localization searches over a large scale network.

The large scale network is broken down into **domains**. The domains composing the large scale network may be bound by point to point links, or through several Internet routers. All the domains are organized in a **neighborhood graph**. The number of neighbors of each domain and the maximum diameter of the graph have to be limited. The search is propagated step to step into the graph. If the search in the local domain fails, in the next step the domains which are immediate neighbors of the first one take part in the search, next step neighbors of the neighbors are involved – and so on. We estimate the **distance** between two domains by half of the round time trip of a message between them.

Every application fixes freely the scope of its localization searches. For some applications, it is preferable to stay in the local area network because an extended search would be too costly; for other ones, a solution is needed within the limit of a maximum response time; for yet others, only some domains need to be consulted as they know where to find the solution.

Every actor decides on the **visibility** of its ports. A server may restrict its services to sites in its domain, or to sites in specific domains. Other servers may be public, namely they may be used by any domain in the graph. Thus, every server chooses which domains are enabled to use it, by setting the visibility of its ports.

In order to reduce response time, the result of an extended localization search is memorized in a cache. A local cache (present on every site) keeps the information used by local actors. For the domain a **global cache** keeps all the information gathered from outside of the domain. These caches will be managed with a Least Recently Used policy.

When the microkernel of some site is unable to find itself the localization information it needs, it asks an **Extended Localization Server** (*ELS* in the following) for it. There is one ELS on every domain.

A site converses with the ELS of the domain on several occasions : to modify the visibility of its ports, to migrate a port outside of the domain, to invalidate a localization information provided by the ELS, and to initiate an extended localization search.

For an extended search, when the ELS does not find a convenient solution in its global cache, it assigns a **request identifier** to the search, and begins a dialog with other domain ELSs. The waiting site receives the request identifier, and is able to follow the evolution of the search on request.

The dialog between ELSs vary according to the type of search (ports or port groups), and to the scope of the search (list of domains, maximum distance). For ports, the ELS polls directly the creation domain of the port, which is informed in case of migration. For port group with a list of domains, the ELS polls all domains in parallel. For port group with a maximum distance search, the search is propagated step to step, first to immediate neighbors, then possibly to the neighbors of the neighbors, and so on. This search will be stopped on the occurrence of one of three events: *(i)* when enough solutions have been found, or *(ii)* when maximum distance is overrun, or *(iii)* when a domain has already dealt with the same request.

A maximum distance search may become costly in time and in message number when a near solution cannot be found. This drawback may be minimized with a sizable cache which keeps information for a long time, and with efficient stopping tests such as limiting maximum distance.

6.3 Integration of the ELS in Chorus

To integrate the Extended Localization Service into Chorus we need : *(i)* an Extended Localization Library (ELL), which will be linked with applications with extended communications, and *(ii)* the Extended Localization Server (ELS) (see figure 8 an example of extended communication between domain A and domain B).

1. **Application – Library Interface**
   The large majority of communications will be inside a domain. In order to not degrade the performance of local communications, we put extended localization routines in a separate library (outside of the microkernel). Interfaces to extended communications routines will be nearly the same as local ones.
   The Extended library includes other routines in order to : *(i)* modify the visibility of applications ports, *(ii)* modify the limits of a search, *(iii)* anticipate a localization search, and *(iv)* follow the result of an extended search. A localization search is automatically started when a message is sent, but for extended localization it could be interesting to anticipate this search, and to follow the result of the search in order to keep the user waiting.

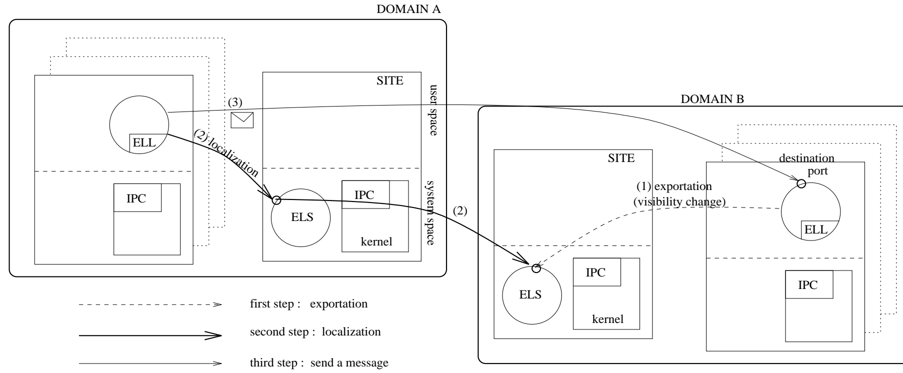

Fig. 8. ELS Architecture

2. **ELS Implementation**
   The heart of the localization service is the ELS. We present here its tasks, and the broad lines of its implementation. The ELS of a domain will be implemented as a **supervisor actor** on one site of this domain. Other sites will contact it through a well known port group. This actor will be made up of several threads corresponding to the different tasks it has to perform.
   (a) **Tasks to be performed by ELSs**
   An ELS has *(i)* to be kept informed of which ports of the domain can be seen from the outside, *(ii)* to localize out of domain entities, *(iii)* to answer other domain requests, *(iv)* to register the ports born in the domain which have migrated outside of it, *(v)* to record the use of the server, to detect occasional anomalies and other servers failures.
   (b) **Memory Resident Information**
   The server needs to maintain a lot of data areas in order to perform its tasks: a global localization cache for the domain, the last requests processed for each domain, the ports visible out of the domain, the ports that have migrated out of the domain, and lastly its neighbors with their current distances. All these table areas should be stored in main memory in order to provide low response times. They must also be saved in permanent memory in order to keep information even if the ELS fails, thus ensuring normal response times just after recovery.
3. **Changes needed in the microkernel**
   The graph of protocols which is implemented in the microkernel will have to be modified in order to cope with messages destinated to other domains: two protocols will be needed *(i)* a router protocol, *(ii)* an intergateway transport protocol [Syst95b].

6.4 Conclusion on limitations

This study leads us to the conclusion that microkernels should be adapted to operate over large scale networks. This adaptation involves an extended local-

ization service. We have presented the design of such a service, and given an implementation scheme for implementation in the Chorus microkernel.

The extended localization service will be useful to applications which need to find dynamically a server in a large scale network, for example in order to locate the nearest instance of a replicated server, or to find a server knowing its domain name only.

In the future we plan to implement the extended localization service into Chorus.

7 Conclusion

This document has presented work done at INT with the Chorus Operating System since 1989. It has listed benefits and limits in using the micro-kernel and subsystem technologies.

The main benefits in using these technologies are the followings:

- As shown by the port on Macintosh and the cohabitation with MacOS, the system allows a great flexibility in using the memory management.
- As shown by the cohabitation version and by the MacOS subsystem, interrupt management can be very versatile too.
- Our research works on Fault Tolerance and Quality of Service, show that it is easier to add services in a subsystem than in a traditional kernel.
- Multiplexing servers is also very easy due to threads of control known at the system level. This is useful to build new subsystem as well as to create new applications.
- IPC mechanism and mini-messages are a good base to build distributed multi threaded servers.

Although there are limits to Chorus, many of which are common to other micro-kernels [Amsa95], these limits have never prevented us from building new applications or adding services to the system. Moreover, the modular design of the micro-kernel allows modifications to the internal mechanisms. In particular, we are adapting Chorus so as it can operate over large scale networks.

We now know much about these technologies and will continue to use them as a basis to experiment our research work.

References

[Abro89] V. Abrossimov and M. Rozier. Generic Virtual Memory Management for Operating System Kernels. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park(SA), December 1989.

[Alar92] E. Alard and G. Bernard. Preemptive Process Migration in Networks of NIX Workstations. In *Proc. 7th International Symposium on Computer and Information Sciences*, Antalya (Turkey), November 1992.

[Amsa95] L. Amsaleg, G. Muller, I. Puaut, and X. Rousset de Pina. Experience with Building Distributed Systems on top of the Mach Microkernel. In *Broadcast, Esprit Research Project 6360, Third Year Report*, July 1995.

[Bac93] C. Bac and E. Garnier. Cohabitation and Cooperation of Chorus and MacOS. In *Proc. USENIX Symposium on Micro-Kernels and Other Kernel Architectures*, San Diego (SA), September 1993.

[Bac94] C. Bac and H.Q. Nguyen. ChorusToolbox : MacOS running on top of Chorus. In *Proc. SUUG '94 Conference*, Moscou (Russie), April 1994.

[Bach86] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall Software Series, Englewood Cliff, New Jersey (SA), 1986.

[Bern94] G. Bernard and D. Conan. Flexible Checkpointing and Efficient Rollback-Recovery for Distributed Computing. In *Proc. SUUG '94 Conference*, Moscou (Russie), April 1994.

[Borg89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance nder NIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[Elno93] E.N. Elnozahy. *Manetho: Fault Tolerance in Distributed Systems Using Rollback-Recovery and Process Replication*. PhD thesis, Rice niversity (SA), October 1993.

[Gold90] A.P. Goldberg, A. Gopal, K. Li, R. Strom, and D.F. Bacon. Transparent Recovery of Mach Applications. In *Proc. 1st USENIX Mach Symposium*, 1990.

[John89] D.B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice niversity (SA), December 1989.

[Nguy95] H.Q. Nguyen, G. Bernard, and D. Belaid. System Support for Distributed Multimedia Applications with Guaranteed Quality of Service. In *Proc. HPN'95, 6th IFIP International Conference on High Performance Networking*, Palma de Mallorca, Balearic Islands (Spain), September 1995.

[O'Co94] M. O'Connor, B. Tangney, V. Cahill, and N. Harris. Micro-kernel Support for Migration. *Distributed Systems Engineering Journal*, 1(4), June 1994.

[Phil95] L. Philippe and G.-R. Perrin. Migration de processus dans Chorus/MiX. *Revue Électronique sur les Réseaux et l'Informatique Répartie*, (1), Avril 1995.

[Rozi88] 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 Journal, The USENIX Association*, 1(4), December 1988.

[Rozi93] M. Rozier. Chorus Kernel v3r4.2 Programmers Reference Manual. Technical Report CS/TR-92-26.1, Chorus Systèmes, Saint-Quentin-en-Yvelines (France), March 1993.

[Stro85] R.E. Strom and S.A. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[Syst95a] Chorus Systèmes. Chorus Kernel v3r5 : Implementation Guide. Technical report, Chorus Systèmes, March 1995.

[Syst95b] Chorus Systèmes. Chorus Kernel v3r5 : Network Artchitecture. Technical report, Chorus Systèmes, March 1995.

[Taco94] C. Taconet and G. Bernard. A Localization Service for Large Scale Distributed Systems based on Microkernel Technology. In *Proc. ROSE'94 Technical Sessions*, Bucharest (Romania), November 1994.

[Walp92] J. Walpole, J. Inouye, and R. Konuru. Modularity and Interfaces in Micro-Kernel Design and Implementation: a Case Study of Chorus on the HP PA-RISC. In *Proc. USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, Seattle (SA), April 1992.

This article was processed using the LaTeX macro package with LLNCS style