CHORUS Distributed Operating Systems

M. Rozier, V. Abrossimov, F. Armand,
I. Boule, M. Gien, M. Guillemont,
F. Herrmann, C. Kaiser, S. Langlois,
P. Léonard, and W. Neuhauser
Chorus systèmes

CHORUS: in ancient Greek drama, a company of performers providing explanation and elaboration of the main action. (Webster's New World Dictionary)

ABSTRACT: The CHORUS technology has been designed for building "new generations" of open, distributed, scalable Operating Systems. CHORUS has the following main characteristics:

- a communication-based technology, relying on a minimal Nucleus integrating distributed processing and communication at the lowest level, and providing generic services used by a set of subsystem servers to provide extended standard operating system interfaces (a UNIX interface has been developed, others such as OS/2 and object oriented systems are envisaged);

- real-time services provided by the real-time Nucleus, and accessible by "system programmers" at the different system levels,
- a modular architecture providing scalability, and allowing in particular dynamic configuration of the system and its applications over a wide range of hardware and network configurations, including parallel and multiprocessor systems.

CHORUS-V3 is the current version of the CHORUS Distributed Operating System, developed by Chorus systèmes. Earlier versions were studied and implemented within the Chorus research project at INRIA between 1979 and 1986.

This paper presents the CHORUS architecture and the facilities provided by the CHORUS-V3 Nucleus. It describes the UNIX subsystem built with the CHORUS technology that provides:

- binary compatibility with UNIX,
- extended UNIX services supporting distributed applications (network IPC, distributed virtual memory), light-weight processes, and real-time facilities.

1. *Introduction*

The evolution of computer applications has led to the design of large distributed systems for which the requirements for efficiency and availability have increased, as has the need for higher level tools to help in their construction, operation and administration.

This evolution introduces requirements for new system structures which are difficult to fulfill merely by extending current monolithic operating systems into networks of cooperating systems. This has led to a new generation of *distributed* operating systems.

- Separate applications running on different machines, from different suppliers, supporting different operating systems, and written in a variety of programming languages need to be tightly coupled and logically integrated. The loose coupling provided by current computer networking is insufficient. The requirement is for tighter logical coupling.
- Applications often evolve by growing in size. Typically this leads to distributing programs on several machines, to grouping several geographically distributed sets of files into a unique logical one, to upgrading hardware and software to take advantage of the latest technologies, newer releases, etc. The requirement is for a gradual on-line evolution.
- Applications grow in complexity and get more and more difficult to master, i.e., to specify, to debug, to tune. The requirement is for a clear, logical architecture, which allows the mapping of the modularity of the application onto the operational system and to hide distribution when it does not directly reflect the distributed nature of organizations.

These structural properties can best be accomplished through a set of unified, coherent and standard basic concepts and structures providing a rigorous framework adapted to constructing distributed operating systems.

The CHORUS architecture has been designed to meet these requirements. Its foundation is a generic Nucleus running on each machine; communication and distribution are managed at the lowest level by this Nucleus; customary operating systems are build as subsystems on top of the generic Nucleus using its basic services; user application programs run in the context of these operating systems. CHORUS provides the generic Nucleus and a set of servers implementing generic operating system services, which are used to build complete host operating systems. The generic CHORUS Nucleus implements the real-time services required by real-time users. Although it is not dedicated to a particular system, CHORUS provides also a standard UNIX subsystem that can execute UNIX programs with a distributed architecture, as a direct result of the CHORUS technology.

This paper focuses on the CHORUS architecture, the facilities provided by its Nucleus, and the – distributed – UNIX subsystem implementation. Extensions to UNIX services concerning real-time, multi-thread processes, distributed applications and servers are outlined.

The CHORUS history and its transition from research to industry is summarized in section 2. Section 3 introduces the key concepts of the CHORUS architecture and the facilities provided by the CHORUS Nucleus. Section 4 explains how the "old" UNIX kernel has been adjusted to state-of-the-art operating system technology while preserving its semantics, and gives examples of how its services can then be easily extended to handle distribution. Section 5 gives some implementation considerations and concluding remarks.

Comments about some of the important design choices, often related to previous experience, are given in small paragraphs entitled *"RATIONALE."*

2. Background and Related Work

"Chorus" was a research project on Distributed Systems at INRIA[1] in France from 1979 to 1986. Three iterations were developed, referred to as CHORUS-V0, CHORUS-V1, and CHORUS-V2, all based on a communications-oriented kernel [Zimmermann et al. 1981; Guillemont 1982(2); Zimmermann et al. 1984; Rozier & Legatheaux-Martins 1987]. The basic concept for handling distributed computing within CHORUS, for system as well as application services, is for a "Nucleus" to manage the exchange of "Messages" between "Ports" attached to "Actors."

While early versions of CHORUS had a custom interface, CHORUS-V2 [Armand et al. 1986] was compatible with UNIX System V, and had been used as a basis for supporting half a dozen research distributed applications. CHORUS-V3 is the current version, developed by Chorus systêmes. It builds on previous CHORUS experience [Rosier & Legatheaux-Martins 1987] and integrates many concepts from state-of-the-art distributed systems

1. INRIA: Institut National de Recherche en Informatique et Automatique.

developed in several research projects, while taking into account constraints of the industrial environment.

The CHORUS-V3 message-passing Nucleus compares to the V-system [Cheriton 1988(1)] of Stanford University, distributed virtual memory and "threads" are similar to that of Mach [Accetta et al. 1986] of Carnegie Mellon University, network addressing incorporates ideas from Amoeba [Mullender et al. 1987] of the University of Amsterdam, and uniform file naming is based on a scheme similar to the one used in Bell Laboratories' 9th Edition UNIX [Presotto 1986; Weinberger 1986].

This technology has been used to implement a distributed UNIX system [Herrmann et al 1988] as a set of servers using the generic services provided by the CHORUS *Nucleus*.

2.1 Early Research

The Chorus project at INRIA was initiated with a combined experience from previous research in packet switching Computer Networks – Cyclades [Pouzin et al. 1982] – and time sharing Operating Systems – Esope [Bétourné et al. 1970]. The idea was to bring distributed control techniques originated in the context of packet switching networks into distributed operating systems.

In 1979 INRIA also launched another project, Sol, to reimplement a complete UNIX environment on French micro and mini computers [Gien 1983]. The Sol team joined Chorus in 1984, bringing their UNIX expertise to the project.

2.2 CHORUS-V0 (1980-1982)

CHORUS-V0 experimented with three main concepts:

- A distributed application which was an ensemble of independent actors communicating exclusively by exchange of messages through ports or groups of ports; port management and naming was designed so as to allow port migration and dynamic reconfiguration of applications.
- The operation of an actor which was an alternate sequence of indivisible execution phases, called processing-steps, and

of communication phases; it provided a message-driven automaton style of processing.

- The operating system was built as a small nucleus, simple and reliable, replicated on each site and complemented by distributed system actors, in charge of ports, actors, files, terminal and network management.

These original design choices were revealed to be sound and were maintained in subsequent versions.

These CHORUS concepts have been applied in particular for fault tolerance: the "coupled actors" scheme [Banino & Fabre 1982] provided a basis for non-stop services.

CHORUS-V0 was implemented on Intel 8086 machines, interconnected by a 50 Kb/s ring network (Danube). The prototype was written in UCSD Pascal and the code was interpreted. It was running by mid-1982.

2.3 CHORUS-V1 (1982-1984)

This version moved CHORUS from a prototype to a real system. The sites were SM90 multi-processor micro-computers – based on Motorola 68000 and later 68020 – interconnected by a 10 Mb/s Ethernet. In a multi-processor configuration, one processor ran UNIX, as a development system and for managing the disk; other processors (up to seven) ran CHORUS, one of them interfacing to the network. The Pascal code was compiled.

The main focus of this version was experimentation with a native implementation of CHORUS on multi-processor architecture.

The design had few changes from CHORUS-V0, namely

- Structured messages were introduced to allow embedding protocols and migrating their contexts.
- The concept of an activity message, transporting the context of embedded computations and the graph of future distributed computation, was experimented on for a fault tolerant application [Banino et al. 1985(1)].

CHORUS-V1 was running in mid-1984. It was distributed to a dozen places, some of which still use it in 1988. It was documented for these users.

2.4 CHORUS-V2 (1984-1986)

Adopting UNIX forced the CHORUS interface to be recast and the system actors to be changed. The Nucleus, on the other hand, did not change a great deal. The UNIX subsystem was developed partly from results of the Sol project (File Manager) and partly from scratch (Process Manager). Concepts such as ports, messages, processing steps, and remote procedure calls were revisited in order to be closer to UNIX semantics and to allow a protection scheme à la UNIX. The UNIX interface was extended to support distributed applications (distant fork, distributed signals, distributed files).

CHORUS-V2 was an opportunity to reconsider the whole UNIX kernel architecture with two objectives:

1. Modularity: all UNIX services were split into several independent actors. This implied splitting UNIX kernel data into several independent CHORUS actors along with cooperation protocols between these actors.
2. Distribution: objects managed by system actors (files, processes) could be distributed; services offered by system actors could also be distributed (e.g. distant fork, remote file access); this implied new protocols, naming, localization, etc.; the designation and naming levels for distributed objects, groups and the communication protocols were redesigned.

A distributed file system was implemented. A distributed shell for UNIX was also developed.

All this work was an irreplaceable exercise for CHORUS-V3: CHORUS-V2 may be considered as the draft of the current version.

CHORUS-V2 was running at the end of 1986. It has been documented and used by research groups outside the Chorus project.

2.5 *CHORUS-V3 (1987-)*

The objectives of this current version are to provide an industrial product integrating all positive aspects of the previous experiences and research versions of CHORUS and of other systems along with several new significant features. CHORUS-V3 is described in the rest of the paper.

2.6 *Appraisal of Four CHORUS Versions*

The first lesson which can be pulled out from the CHORUS story is that several steps and successive whole redesigns and implementations of the same basic concepts provide an exceptional opportunity for refining, maturing and validating initial intuitions: think about UNIX!

On the technical side, the basic modular structure, kernel, and system actors never really changed; some concepts also resisted all versions: ports, port groups, messages.

However, the style of communication (IPC) evolved in each version: naming and protection of ports experimented with local names, global names, protection identifiers. The protocols which were purely asynchronous at the beginning moved by steps to synchronous communications and led finally to synchronous RPC. Consequently, structured messages were no longer useful and processing steps within an actor were in contradiction with the extent of the RPC.

Actors evolved from a purely sequential automaton with processing steps to a real-time multi-thread virtual machine, which is now used for resource allocation and as an addressing space.

Protection and fault tolerance are still open questions since UNIX leaves few choices and because earlier experiments were not convincing as to the value of implementing specific mechanisms inside the kernel (e.g., reliable broadcast, atomic transactions, commit, redundancy).

Early versions of CHORUS handled fixed memory spaces, with possibility to use memory management units for relocation. This evolved to dynamic virtual memory systems with demand paging, mapped into distributed and sharable segments.

Finally, although Pascal did not cause any major problem as an implementation language, it has been replaced by C++ which can rely on the wider audience that C has now in the industry. C++ also brings facilities (classes, inheritance, tight coupling with C) that have been quite useful as a system language.

Since the beginning of the project, most design concepts and experiments have been reported. A summary of these publications is given in §8.

3. CHORUS Concepts and Facilities

3.1 The CHORUS Architecture

3.1.1 Overall Organization

A CHORUS System is composed of a small-sized **Nucleus** and a number of **System Servers**. Those servers cooperate in the context of **Subsystems** (e.g., UNIX) providing a coherent set of services and interfaces to their "users" (Figure 1).

RATIONALE. This overall organization is a logical view of an open operating system. It can be mapped on a centralized as well as on a distributed configuration. At this level, distribution is hidden.

The choice has been to build a two level logical structure, with a "generic nucleus" at the lowest level, and almost autonomous "subsystems" providing applications with usual operating system services.

Therefore the CHORUS Nucleus has not been built as the core of a specific operating system (e.g., UNIX), rather it provides generic tools designed to support a variety of host subsystems, which can co-exist on top of the Nucleus.

This structure allows support of application programs which already run on existing (usually centralized) operating systems, by reproducing those existing operating system interfaces within a given subsystem. This approach is illustrated with UNIX in this paper.

Note also the now classic idea of separating the functions of an operating system into groups of services provided by

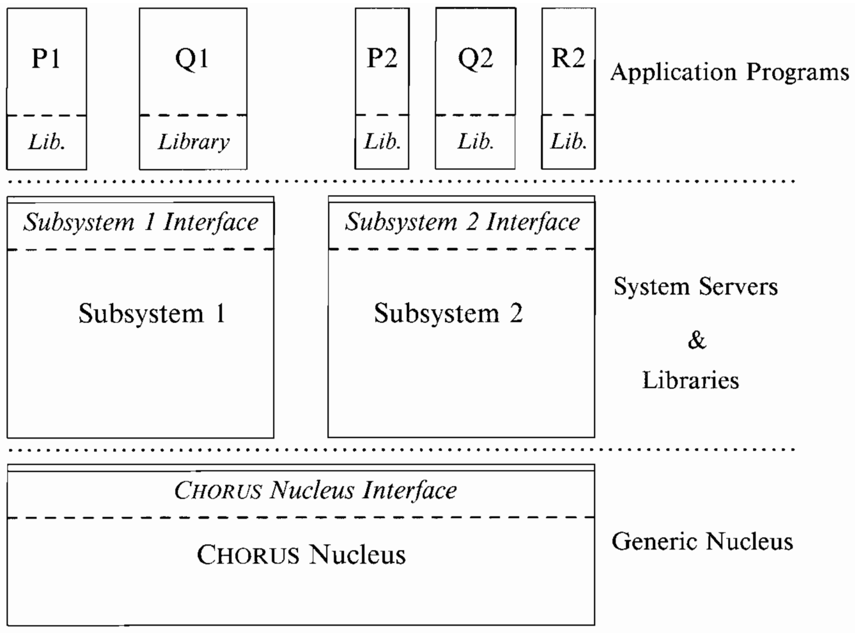


Figure 1: The CHORUS Architecture

autonomous servers inside subsystems. In monolithic systems, these functions are usually part of the "kernel." This separation of functions increases modularity and therefore portability and scalability of the overall system.

3.1.1.1 THE CHORUS NUCLEUS The CHORUS Nucleus (Figure 2) plays a double role:

1. Local services:
   It manages, at the lowest level, the local physical computing resources of a "computer," called a **Site**, by means of three clearly identified components:
   - allocation of local processor(s) is controlled by a **Real-time multi-tasking Executive**. This executive provides fine grain synchronization and priority-based preemptive scheduling,
   - local memory is managed by the **Virtual Memory Manager** controlling memory space allocation and structuring virtual memory address spaces,

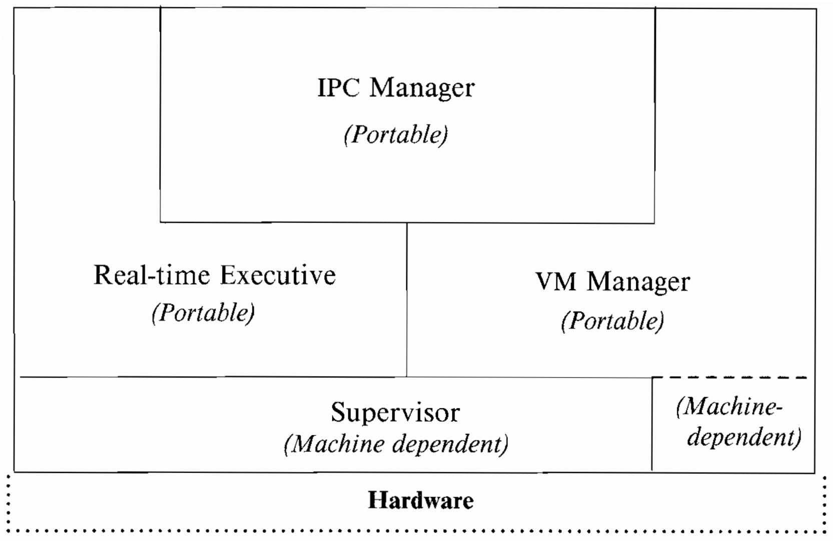


Figure 2: The CHORUS Nucleus

- external events – interrupts, traps, exceptions – are dispatched by the **Supervisor**.

2. Global services:
   The **IPC Manager** provides the communication service, delivering messages regardless of the location of their destination within a CHORUS distributed system. It supports **RPC** (Remote Procedure Call) facilities and **asynchronous** message exchange, and implements multicast as well as functional addressing. It may rely on external system servers (i.e., Network Managers) to operate all kinds of network protocols.

RATIONALE. Surprisingly, the structure of the Nucleus is also logical, and distribution is almost hidden. Local services deal with local resources and can be mostly managed with local information only. Global services involve cooperation between nuclei to cope with distribution.

In CHORUS-V3 it has been decided for efficiency reasons experienced in CHORUS-V2, to include in the nucleus some functions which could have been provided by system servers:

actor and port management (creation, destruction, localization), name management, RPC management.

The "standard" CHORUS IPC is the only means – or "tool" – used to communicate between managers of different sites; they all use it rather than dedicated protocols – for example, Virtual Memory managers requesting a remote segment to service a page fault.

The nucleus has also been designed to be highly portable, even if this prevents using some specific features of the underlying hardware. Experience with porting the nucleus to half a dozen of different Memory Management Units (MMU's) on three chip sets has shown the validity of such a choice.

3.1.1.2 THE SUBSYSTEMS System servers implement high-level system services, and cooperate to provide a coherent operating system interface. They communicate via the Inter-Process Communication facility (IPC) provided by the CHORUS Nucleus.

3.1.1.3 SYSTEM INTERFACES A CHORUS system exhibits several levels of interface (Figure 1):

- **Nucleus Interface**: The Nucleus interface is composed of a set of procedures providing access to the services of the Nucleus. If the Nucleus cannot render the service directly, it communicates with a distant Nucleus via the IPC.
- **Subsystem Interface**: This interface is composed of a set of procedures accessing the Nucleus interface, and some Subsystem specific protected data. If a service cannot be rendered directly from this information, these procedures "call" (RPC) the services provided by System Servers.

The Nucleus and Subsystem interfaces are enriched by libraries. Such libraries permit the definition of programming language specific access to System functionalities. These libraries (e.g., the "C" library) are made up of functions linked into and executed in the context of user programs.

3.1.2 Basic Abstractions Implemented by the CHORUS Nucleus

The basic abstractions implemented and managed by the CHORUS Nucleus are:

Actor	unit of resource collection, and memory address space
Thread	unit of sequential execution
Message	unit of communication
Port, Port Groups	unit of addressing and (re)configuration basis
Unique Identifier (UI)	global name
Region	unit of structuring of an Actor address space

These abstractions (Figure 3) correspond to object classes which are private to the CHORUS Nucleus: both the object representation and the operations on the objects are managed by the Nucleus. Those basic abstractions are object classes to which the services invoked at the Nucleus interface are related.

Three other abstractions are also managed both by the CHORUS Nucleus and Subsystem Actors:

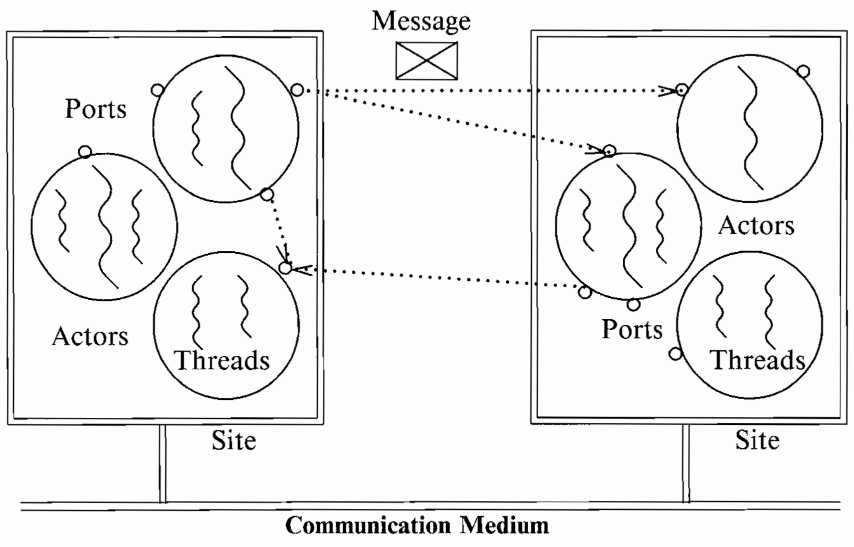


Figure 3: CHORUS Main Abstractions

Segment	unit of data encapsulation
Capability	unit of data access control
Protection Identifier	unit of authentication

RATIONALE. Each of the above abstractions plays a specific role in the System.

An *Actor* encapsulates a set of resources:

- a virtual memory context divided into *Regions*, coupled with local or distant segments,
- a communication context, composed of a set of ports,
- an execution context, composed of a set of threads.

A *Thread* is the grain of execution and corresponds to the usual notion of a process or task. A thread is tied to one and only one actor, sharing the actor's resources with the other threads of that actor.

Messages are byte strings addressed to ports.

Upon creation, a *Port* is attached to one actor, allowing (the threads of) that actor to receive messages on that port. Ports can migrate from one actor to another. Any thread knowing a port can send messages to it.

Ports can be grouped dynamically into *Port Groups* providing multicast or functional addressing facilities.

Actors, ports and port groups receive *Unique Identifiers (UI)* which are global (location independent), unique in space and in time.

Segments are collections of data located anywhere in the system and referred to independently of the type of device used to store them. Segments are managed by System Servers, defining the way they are designated and handling their storage.

Two mechanisms are provided for building access control mechanisms and authentication:

Resources (e.g., segments) can be identified within their servers by a *key* which is server dependent. Since keys have no meaning outside a server they are associated with the port UI of the server to form a (global) *Capability.*

Actors and ports receive *Protection Identifiers* which the nuclei use to stamp all the messages sent and that receiving actors use for authentication.

3.2 Active Entities

3.2.1 Physical Support: Sites

The physical support of a CHORUS system is composed of an ensemble of **sites** ("machines" or "boards"), interconnected by a communication **network** (or Bus). A *site* is a grouping of tightly coupled physical resources: one or more processors, central memory, and attached I/O devices. There is one CHORUS Nucleus per site.

RATIONALE. A site is not a basic CHORUS abstraction (neither are devices). Site management is performed by site servers, i.e., system administrators, and the site abstraction is implemented by these servers.

3.2.2 Virtual Machines: Actors

The **actor** is the logical "unit of distribution" and of collection of resources in a CHORUS system. An *actor* defines a protected (paged) address space supporting the execution of threads (light-weight processes or tasks), that share the address space of the actor. An address space is split into a "user" address space and a "system" address space. On a given site, each actor's "system" address space is identical and its access is restricted to privileged levels of execution (Figure 4).

A given site may support many simultaneous actors. Since each has its own "user" address space, actors define protected "virtual machines" to the user.

Any given actor is tied to one site and the threads supported by any given actor are always executed on the site to which that actor is tied. The physical memory used by the code and data of a thread is always that of the actor's site. Actors (and threads) cannot migrate from one site to another.

RATIONALE. Because each actor is tied to one site, the state of the actor (i.e., its contexts) is precisely defined – there is no uncertainty due to distribution since it depends only on the status of its supporting site. The state of an actor can then be known rapidly and decisions can be taken easily. The crash of a site leads to the complete crash of its actors – there is no actor partially crashed.

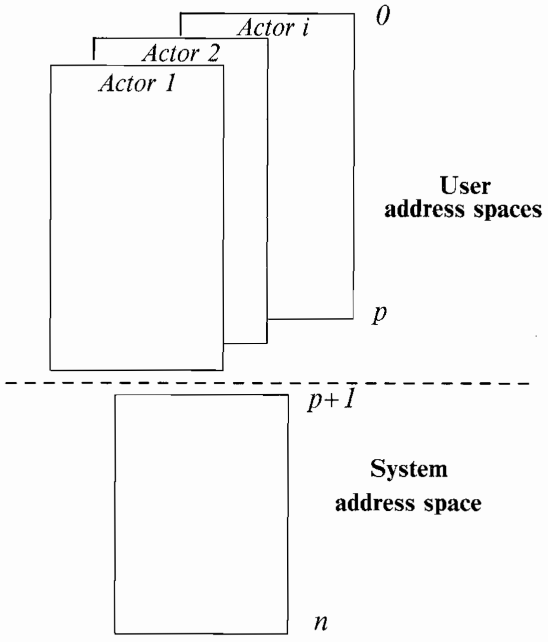


Figure 4: Actor Address Spaces

Actors are designated by capabilities built from a UI, i.e. the UI of the actor's default port and a manipulation key. The knowledge of the capability of an actor yields all of the rights on that actor (creating ports, threads and regions in the actor, destroying it, etc.). By default, only the creator of an actor knows the capability of the created actor, however the creator can transmit it to others.

The resources held by an actor (the ports that are attached to the actor, the threads, the memory regions) are designated within the actor's context with Contextual Identifiers (i.e., Local Descriptors). The scope of such identifiers is limited to the specific actor which uses the resource.

3.2.3 Processes: Threads

The **thread** is the "unit of execution" in the CHORUS system. A *thread* is a sequential flow of control and is characterized by a thread **context** corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.).

A thread is always tied to one and only one actor, which defines the address space in which the thread can operate. The actor thus constitutes the execution environment of the thread. Within the actor, many threads can be created and can run in parallel. These threads share the resources (memory, ports, etc.) of that actor and of that actor only. When a site supports multiple processors, the threads of an actor can be made to run in parallel on those different processors.

Threads are scheduled as independent entities. The basic scheme is a preemptive priority based scheduling, but the Nucleus implements also time slicing and priority degradation on a *per thread basis*. This allows for example real-time applications and multi-user interactive environments to be supported by the same Nucleus according to their respective needs and constraints.

Threads communicate and synchronize by exchanging messages using the CHORUS IPC (see §3.3), even if they are located on the same site. However, as threads of an actor share the same address space, communication and synchronization mechanisms based on shared memory can also be used inside one actor. In most cases, when the machine instruction set allows it, the implementation of such synchronization tools avoids invoking the nucleus.

RATIONALE. Why threads?

- Because one actor corresponds to one virtual address space and is tied to one site, threads allow multiple processes on a site corresponding to a machine with no virtual memory (i.e., which provides only one addressing space, such as a Transputer).
- Threads provide a powerful tool for programming I/O drivers. Those are bound to interrupts and associating one thread to each I/O stream simplifies driver programming.
- Threads allow multi-programming servers, providing a good match to "client-server" style of programming.
- Threads allow using multiple processors within one actor, e.g., on a shared memory symmetric multi-processor site.

- Threads are light-weight processes, whose context switching is far less expensive than an actor context switch.

3.2.4 Actors and Threads Nucleus Interface

The Nucleus interface for actor and thread management is summarized in Table 1:

`actorCreate`	*Create an actor*
`actorDelete`	*Delete an actor*
`actorStop`	*Stop the actor's threads*
`actorStart`	*Restart the actor's threads*
`actorSetPar`	*Set actor parameters*
`threadCreate`	*Create a thread*
`threadDelete`	*Delete a thread*
`threadStop`	*Stop a thread*
`threadStart`	*Restart a thread*
`threadSetPar`	*Set thread parameters*

Table 1: Actors and Threads Services

3.3 Communication Entities

3.3.1 Overview

Threads synchronize and communicate using a single basic mechanism: exchange of **messages** via message queues called **Ports**.

Inside an actor, ports are locally used as message semaphores. More generally, unique and global names (UI's) may be given to ports, allowing message communications to cross the actor's boundaries. This facility, known as **IPC** (Inter-Process Communication facility), allows any thread to communicate and to synchronize with any other thread on any site.

The main characteristic of the CHORUS IPC is its transparency *vis-à-vis* the localization of threads: communication is expressed through a uniform interface (ports), regardless of whether the communication is between two threads in a single actor, between two threads in two different actors on the same site, or between two threads in two different actors on two different sites. Messages are transferred from a sending port to a receiving port.

3.3.2 Messages

A message is basically a contiguous byte string, logically copied from the sender address space to the receiver(s) address space(s). Using a coupling between virtual memory management and IPC, large messages may be transferred efficiently by deferred copying (copy on write), or even by simply moving page descriptors (on a given site).

RATIONALE. Why messages rather than shared memory?

- Messages make the exchange of information explicit, thus clarifying all actions.
- Messages make debugging of a distributed application easier, especially when using RPC which involves sequential processing steps in different actors.
- Messages are easier to manage than shared memory in a heterogeneous environment.
- The state of an actor can be known more precisely (before a message transmission, after receiving a message, etc.).
- The cost of information exchange is better isolated and evaluated when it is done through messages – since there are explicit calls to the nucleus – than the cost of memory accesses – which depend on traffic on the bus, memory contention, memory locking, etc. The grain of information exchange is bigger, better defined, and its cost better known.
- Performance of local communications are still preserved by implementation hints and local optimizations (see §5).

3.3.3 Ports

Messages are not addressed directly to threads (nor actors), but to intermediate entities called **ports**. The notion of a port provides the necessary decoupling between the interface of a service and its implementation. In particular, it provides the basis for dynamic reconfiguration (see §3.4.4).

A port represents both:

- a *resource* (essentially a message queue holding the messages received by the port but not yet consumed by the receiving threads),
- an *address* to which messages can be sent.

When created, a port is *attached* to one actor. The threads of this actor (and only them) may receive messages on the port. A port can only be attached to a single actor at a time, but it can be "used" by different threads within that actor.

A port can be successively attached to different actors: i.e. a port can *migrate* from one actor to another. This migration can be applied also to the messages already received by the port.

RATIONALE. Why Ports?

Decoupling communication from execution, a Port is a functional name for receiving messages:

- one actor may have several ports and therefore communication can be multiplexed,
- a port can be used successively by several actors (actors grouped, and functionally equivalent),
- multiple threads may share a single port, providing cheap expansion of server performance on multiprocessor machines,
- the notion of "port" provides the basis for dynamic reconfiguration: the extra level of indirection (the ports) between any two communicating threads means that the threads supplying a given service can be changed from a thread of one actor to a thread of another actor. This is done by changing the attachment of the appropriate port from the first thread's actor to the new thread's actor (see §3.4.4).

When a port is considered as a resource – for receiving messages – threads access it by means of a local contextual identifier – i.e., a port descriptor – identifying the port within the actor which the port is attached to.

When a port is considered as a destination address for the IPC, it is designated by a UI. A port UI is generated on port creation.

When the port is destroyed, its UI will no longer be used. The knowledge of a port UI gives the right to send messages on that port. Port UI's can be freely transmitted between threads (e.g. in messages).

Messages carry the UI of the port – or port group – they are sent to.

RATIONALE. In the successive versions of CHORUS, naming of ports has changed a number of times:

- In CHORUS-V1, small UIs were adopted as the sole naming space; this proved simple and easy to use, but the lack of protection was an issue for a multi-user environment.
- In CHORUS-V2, UIs were used only by the nucleus and system actors; UNIX processes used contextual identifiers, modeled on file descriptors; protection was insured and port inheritance on *fork* and *exec* was implemented. On the other hand, two main drawbacks were revealed: port inheritance was hard to understand and, more important, port name transmission required specific mechanisms.
- The new scheme adopted in CHORUS-V3 combines advantages of both previous versions. In brief:
    1. Ports are named by global names at user level: name transmission in messages is obvious.
    2. Within an actor, ports attached to the actor are named (in system calls) by local contextual identifiers: this simplifies the user interface, allows controlling the usage of these ports (actually the resources attached to them), and provides performance advantages.
    3. Finally, UIs are protected due to their random and sparse generation in a very large space (128 bits).

3.3.4 Port Groups

Ports can be assembled into **Port Groups** (see Figure 5). The notion of *group* extends port-to-port message passing between threads:

- Asking for a service may not only be done directly from one thread to another thread – via a port. It may also be done by "multicast": from one thread to an *entire group of threads* – via a group of ports.

- Functional access to a service can be selected from among a group of – equivalent – services.

A group of ports is essentially a UI (usable for posting messages). A group exists as long as it has a UI, i.e., groups may be empty. Therefore group UIs may be allocated statically and kept over a "long" period of time. A group is made by creating an empty group and by dynamically inserting ports into the group. A port can be removed from a group. A port can be a member of several groups.

The port group notion provides the basis for stable service naming and reconfigurations – a port of a site that failed, or is overloaded, or is being repaired, may be replaced by another one of the same group, used as a back-up (see §3.4.4).

RATIONALE. This functionality can be used to provide dynamic linking: a subsystem defines names of port groups, declared at system generation time. Port names, which are created dynamically at boot time of every site, are dynamically inserted into the port groups, linking new port names with fixed port groups names. Programs can be written assuming fixed port group names and need not be modified when the site configurations change.

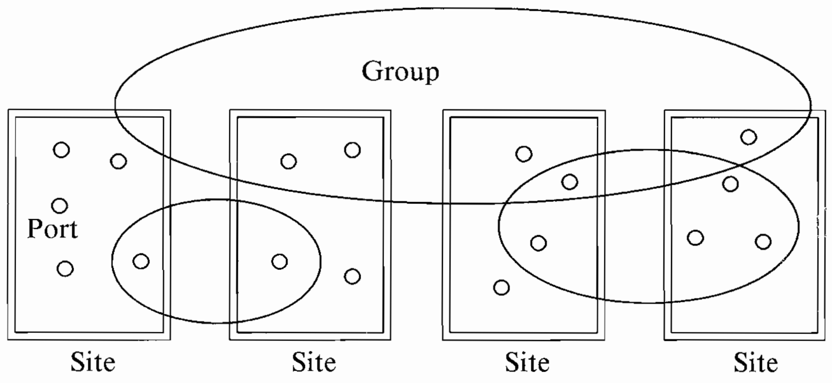


Figure 5: Port Groups

3.3.5 Communication Semantics

The CHORUS Inter-Process Communication (IPC) permits threads to exchange messages in either **asynchronous** mode or in **demand/response** (i.e. Remote Procedure Call or **RPC**) mode.

- **Asynchronous mode**: The emitter of an asynchronous message is blocked only during the time of local processing of the message by the system. The system does not guarantee that the message has been actually received by the destination port or site. When the destination port is not present, the sender is not notified, and the message is destroyed.
- **RPC mode**: The RPC protocol permits the construction of services with a **client-server** model: a demand/response protocol with management of **transactions**. RPC guarantees that the response received by a client is definitely that of the server and corresponds effectively to the request (and not to a former request to which the response would have been lost); RPC also permits a client to know if his request has been received by the server, if the server has crashed before emitting a response, or if the communication path broke.

RATIONALE. Asynchronous IPC and RPC are the only communication services provided by the CHORUS nucleus. The nucleus does not provide "flow control" protocols. RPC is a simple concept, easy to understand, present in language constructs, easy to handle in case of errors or crashes. Flow control would be costly if provided by the nucleus, there are no real standards and needs vary among applications. The asynchronous IPC service is basic enough to allow building more sophisticated protocols within subsystems, and reduces network traffic in the successful cases yielding higher performance and better scaling to large or busy networks.

When messages are sent to port groups, several addressing modes are provided:

- broadcast to all ports in the group,[2]

2. Broadcast mode is not currently applicable to RPC.

- send to any one port of the group,
- send to one port of the group, located on a given site,
- send to one port of the group, located on the same site as a given UI.

3.3.6 Communication Nucleus Interface

The Nucleus interface for communications is summarized in Table 2:

`portCreate`	*Create a port*
`portDelete`	*Delete a port*
`portMigrate`	*Migrate a port*
`grpAllocate`	*Allocate a group name*
`grpPortInsert`	*Insert a port in a group*
`grpPortRemove`	*Remove a port from a group*
`ipcSend`	*Send an asynchronous message*
`ipcCall`	*Send a RPC request and wait for a reply*
`ipcReceive`	*Receive a message*
`ipcReply`	*Reply a message to its original sender*
`ipcForward`	*Forward a message*
`ipcSysInfo`	*Get sender identifiers*

Table 2: Port, Group and Message Services

3.4 Naming and Addressing

3.4.1 Unique Identifiers (UI)

Actors, segments, and IPC addresses (ports and groups) are designated in a *global* fashion with **Unique Identifiers**: the scope of their names is universal and the names are unique in a CHORUS distributed system.

RATIONALE. Global names can be easily transmitted (in particular within messages). They also make the construction of symbolic name servers easier.

Naming Domains: interconnecting CHORUS distributed systems leads to defining naming domains – one domain per CHORUS system. Domains characterize distinct administration prerogatives. A standard structure for UIs – with a part devoted to a domain name – and inter-domain gateways – name servers – allow domain interconnection.

The CHORUS Nucleus implements a localization service, allowing "users" (actors) to use these names without knowledge of the locality of the actual entities.

The *global names* are constructed from *Unique Identifiers.* A UI is unique in space – a sole entity of a CHORUS distributed system can possess this UI at a given instant – and in time – during the lifetime of the system, a given UI will never be used to designate two different entities. A UI is a 128-bit structure. Its uniqueness is assured by classical construction methods of concatenation of a unique – creation – site number and a local (random) stamp.

The localization of a UI is done in a usual way [Legatheaux-Martins & Berbers 1988] using several hints for finding the current residence site when the creation site only is directly given in the UI (see §3.6.1.2).

RATIONALE. Port, group and actor system names (UIs) are directly used by the Nucleus "users" (the actors), and the Nucleus does not control their transmission. It is the responsibility of the *subsystems* to hide these names or to make them visible.

However, the way these names are built offers a cheap level of protection that is suitable for most circumstances. In fact, these names are taken, randomly, from a *large sparse space* (128 bit strings). A user attempting to randomly generate such a name has virtually no chance of finding a valid name during the lifetime of the system.

3.4.2 Capabilities

Some objects are not directly implemented by the Nucleus, but by external services (e.g., segments). These objects are named via global names which hold some protection information, called **capabilities**. A *capability* is made of a UI (the UI of a port of the server managing the object) and a local identifier of the object within the server, called a **key** (Figure 6). This *key* identifies the object and holds the corresponding access control information. The structure and semantics of the keys are defined by their servers.

UI of the Server's Port (128 bits)
Reference of the Resource (Key), within the Server (64 bits)

Figure 6: Structure of a Capability

3.4.3 Port and Port Groups names

Group names play an important role in naming services. Group names are *stable names* for *non-stable entities* (ports): the name of a group can be *rebound* to different entities. This allows, for example, the binding of names to system *services,* rather than the binding of names directly to *servers* providing the services – as a basis for allowing dynamic reconfiguration of services.

For this facility to be secure, the Nucleus must control the operations which associate the port names with group names (i.e., the insertion/removal of ports into/from groups). For that purpose, the Nucleus associates to each group name a **group manipulation key**,[3] required for port insertion and removal. The creator of a group receives the key to the group and may freely transmit the key.

The name and the key are related as follows:

$$name = f(key)$$

where f is a non-invertible function known by every Nucleus.

In brief:

1. For a port:
   - knowledge of the name is equivalent to the emission right (protected by the port name generation);
   - possession of the port is equivalent to the reception right (protected by the impossibility to share ports).
2. For groups:
   - knowledge of the name is equivalent to the emission right (protected by the group name generation);
   - knowledge of the key is equivalent to the update right (protected by the impossibility to discover the key from the name).

3. In fact, the group UI and the group key form a capability.

3.4.4 Reconfiguring a Service

The notion of "port" as an indirection between communicating threads allows one to dynamically modify the implementation of a service within an actor (e.g. add new server threads during "rush hours").

Moreover, the Nucleus allows the dynamic reconfiguration of services between actors by permitting the **migration** of ports. This reconfiguration mechanism requires that the two servers involved in the reconfiguration be active at the same time (Figure 7).

Finally, it also offers some mechanisms permitting one to manage the stability of the system, even in the presence of transitory failures of servers. The notion of port groups is used to establish the stability of server addresses.

Recall that:

- A group collects several ports together.
- A server that possesses the name of a group and its manipulation key can insert new ports into the group, replacing the ports that were attached to servers that have terminated.

A client that *references a group UI* (rather than directly referencing the port attached to a server) can continue to obtain

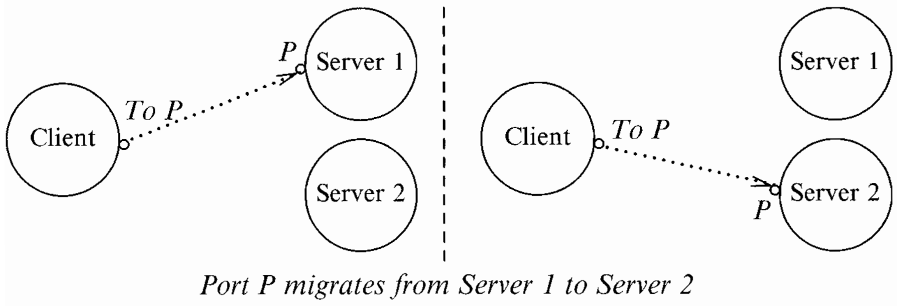


Figure 7: Reconfiguration Using Port Migration

Ports can migrate from one actor to another. While *Client* continues communicating with port *P*, the port can be moved from *Server 1* to *Server 2*.

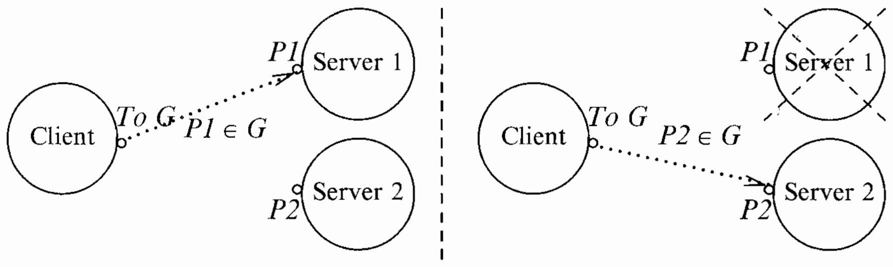
This allows, for example, the updating of a server with a new version or the replacement of one server with a faster one located on another site.

the needed services once the terminated port has been replaced in the group (Figure 8).

In other words, the lifetime of a group of ports is unlimited because groups continue to exist even when ports within the group have terminated.

Thus clients can have stable service as long as their requests for services are made by emission of a message towards a group.

> *RATIONALE.* The coherence of UI space implies that the migration of a port both removes the port from its old site and installs it on its new site: the two actors must be present simultaneously; port migration permits *cold reconfiguration.*
>
> On the other hand, failures imply *hot reconfiguration*: port migration is impossible if one site is not accessible. Group addressing provides the indirection allowing such reconfigurations: a group lifetime is logically infinite as the validity of its update (its coherence) may be checked on any site, even if the group is not yet known by the Nucleus.



P2 has replaced P1 in group G

Figure 8: Reconfiguration Using Groups

Using groups allows a more general reconfiguration facility than is available with port migration.

Client addresses its communications to group *G* instead of directly to port *P1*. The extra level of indirection allows the replacement of *Server 1*, that may have ceased to function, with *Server 2* even though the two servers have their own ports.

3.4.5 *Authentication*

The CHORUS Nucleus provides the ability to protect objects managed by the subsystem servers (e.g., files). As these servers are always invoked via IPC, the IPC provides the support for authentication policies. For that purpose, the Nucleus offers the notion of **Protection Identifier (PI)** and a mechanism for **message-stamping**.

The Nucleus provides a Protection Identifier to each actor and to each port. The structure of these identifiers is fixed but the Nucleus does not associate any semantics to their values, except that it recognizes a special value corresponding to the **super-user** which is allowed to modify the Protection Identifiers.

Upon creation, an actor (a port) receives the same Protection Identifier as the actor which created it. Protection relies on the fact that only the super-user actor can change the Protection Identifier of any actor or port.

Each message sent is stamped by the Nucleus with the Protection Identifiers of its source actor and port. These values can be read but not modified by the receiver of the message, which can apply its own authentication policies.

3.5 *Virtual Memory Management*

3.5.1 *Segments*

The unit of representation of information in the system is the **segment**. Segments are generally located in secondary storage (e.g. files or "swap areas"). Segments are managed by system actors called **segment servers** or **Mappers**. The representation of a segment, its capabilities, access policies, protection, and consistency are defined and maintained by these servers.

RATIONALE. Segments names are global – UI of segment server + local reference – which provides a unique designation mechanism for segments.

Grouping management of segment naming with the management of segments on secondary storage within unique "segment servers" is an implementation choice, not inherent to the CHORUS architecture. Name resolution could be provided by independent name servers.

CHORUS provides a distributed virtual memory management service allowing threads to access segments concurrently.

3.5.2 Mapped segments: Regions

The actor address space is divided into **regions**. A region of an actor maps a portion of a segment at a given virtual address with associated given access rights (read, write, execute per privilege level) (Figure 9). Every reference to an address within a region behaves as a reference to the mapped segment, controlled by the associated access rights.

Threads can create, destroy, and change access rights of the regions of its own actor "user" address space as well as of other actors' "user" address spaces. Note that a thread cannot manipulate the "user" address space of another actor without knowing the UI of the actor.

The "system" address space can be manipulated only by super-user threads.

RATIONALE. Allowing actors to create regions in the "system" address space, shared by all address spaces on a site, is a way to avoid the overhead of an address space context switch.

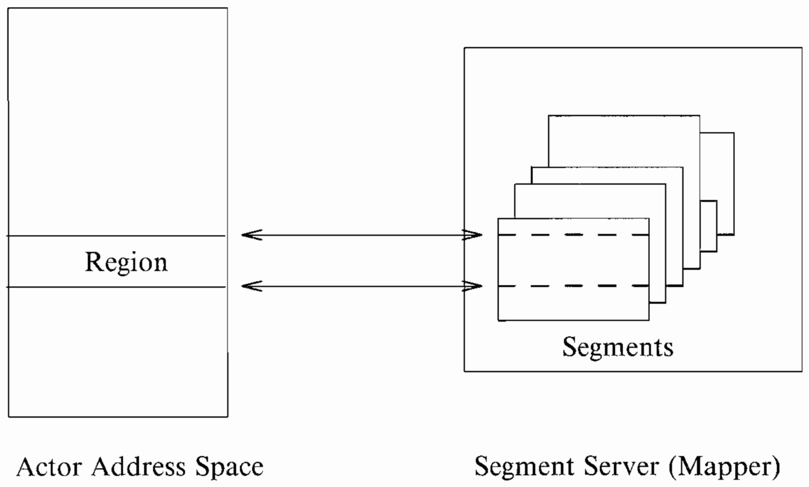


Figure 9: Regions and Segments

In particular the CHORUS system uses this functionality in the following cases:

- IPC between subsystem actors, to avoid re-copying messages between actors of the same site (they are just remapped),
- Interrupt handlers.

3.5.3 Segment Representation in the Nucleus: Local Cache

For each accessed segment on its site, the nucleus encapsulates in a per segment **local cache** the physical memory pages holding portions of segment data. Page faults generated during access to portions of a segment which are not accessible are handled by the Nucleus. In order to resolve these exceptions, the Nucleus may invoke the segment's mapper and fills the *local cache* with the data received from that mapper (Figure 10).

RATIONALE. "On-demand page loading" techniques have been chosen in order to make it possible to access very large segments. Another approach, based on "whole segment

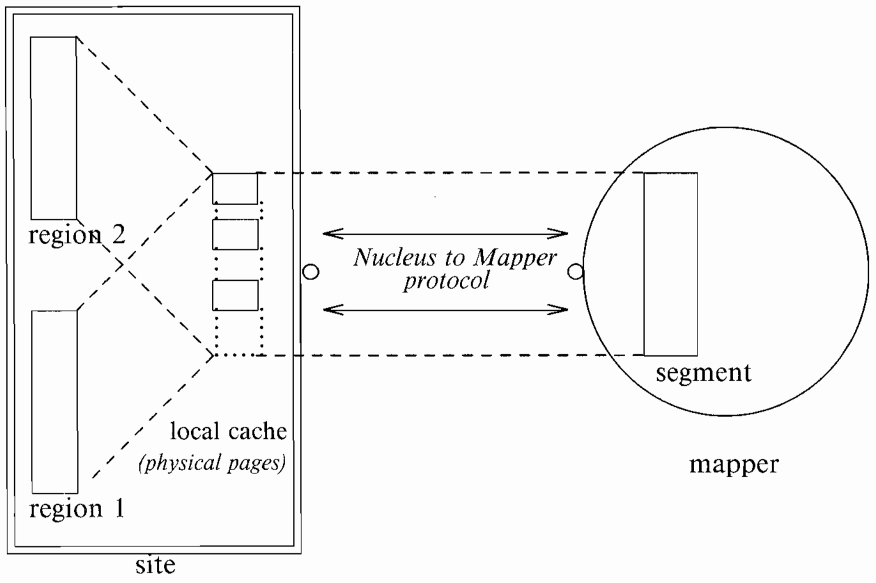


Figure 10: Local Cache

loading" can be found in [Tanenbaum et al. 1986], but this assumes that segments are relatively small and requires big amounts of physical memory.

The consistency of a segment shared among regions belonging to *actors of the same site* is guaranteed by the unicity of the segment local cache in physical memory.

When a segment is shared among *actors of different sites*, there is one segment representation (local cache) per site and Mappers are then in charge of maintaining the consistency of these distributed caches (Figure 11). Algorithms for dealing with problems of coherency of shared memory are proposed in [Li 1986].

A standard Nucleus to Mapper protocol, based on the CHORUS IPC, has been defined for managing local caches:

- on demand paging,
- to flush pages – invalidate them – for swap out and cache consistency,
- to destroy a local cache.

3.5.4 Explicit Access to a Segment

The CHORUS virtual memory management allows also explicit access to (i.e., copy of) segments without mapping them into an address space. This kind of access to a segment uses the same local cache mechanism as described above. Segment consistency is thus guaranteed during concurrent accesses on a given site, whether they are explicit or mapped. Note that mappers do not distinguish between these two kinds of access modes.

The same cache management mechanism is used for segments representing program text and data, mapped files and files accessed by conventional read/write instructions.

RATIONALE. A unique cache management optimizes physical memory allocation and avoids consistency problems between virtual memory and file system caches [Cheriton 1988(2)].

An approach using two different caches is described in [Nelson et al. 1988].

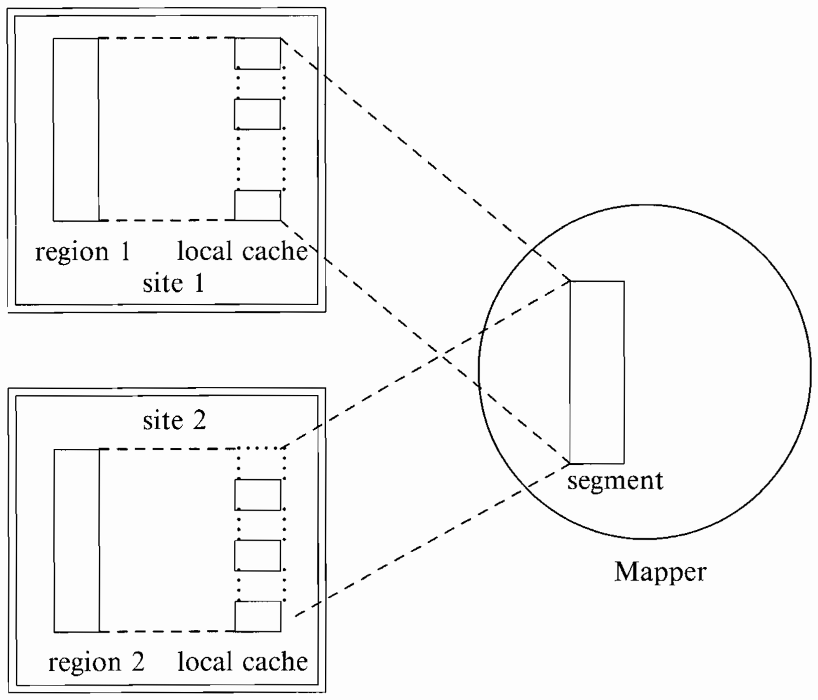


Figure 11: Distributed Local Caches

3.5.5 Deferred Copy Techniques

There are two main circumstances where deferred copy techniques are useful:

1. creation of a new segment as a copy of another one (e.g., UNIX *fork()*, object version management),
2. copy of a portion of data between two existing segments (e.g., IPC, I/O operations).

CHORUS uses two different techniques in such circumstances:

1. history object techniques, similar to shadow object techniques of Mach [Rashid et al. 1987] for initializing large objects,
2. per-virtual-page based techniques [Gingell et al. 1987; Moran 1988] to copy small amounts of data.

3.5.6 Virtual Memory Management Nucleus Interface

The Nucleus interface for memory management is summarized in Table 3:

Regions	
`vmMap`	*Map a segment into a region*
`vmAllocate`	*Initialize a new segment and map it into a region*
`vmFree`	*Delete a region*
`vmReMap`	*Change a region's parameters*
`vmStatus`	*Get status information*

Segments and Local Caches	
`vmOpen`	*Get access to a segment*
`vmClose`	*Release segment access*
`vmInval`	*Invalidate a segment's local cache*
`vmFlush`	*Force updating a segment*
`vmCopy`	*Data transfer between two segments*

Table 3: Virtual Memory Services

3.6 Communications Support

The Nucleus handles message passing between actors executing on the local site. Fully distributed facilities are achieved in cooperation between the Nucleus and the Network Manager. Usually actors don't have to know the site location of ports they want to send messages to.

The first role of the Network Manager is to hide the scattering of ports, and communicating actors, across the network: it helps the Nucleus in conveying transparently messages between actors running on different sites.

Its second function is to act as a "communication channel" server to Actors which do know about network organization and communications facilities. In this case it offers to these kind of actors an access method to network services.

The following sections describe the Network Manager functions.

3.6.1 *Remote IPC and RPC Support*

During their lifetime, ports can be attached to different Actors, one after another. The Network Manager, cooperating with the Nucleus, is in charge of fully hiding the location of a RPC or IPC remote destination port from the point of view of the sending side.

For doing so, the Network Manager implements a set of protocols, with different functionalities.

Two kinds of protocols are needed: the first one deals with CHORUS specific features, such as port localization, remote host failure handling, etc. The second one is responsible for data transmission between sites; this last family is independent of any system specificity.

To enforce portability of the actor code, the Network Manager is designed as three distinct modules each of which makes very few assumptions about the other two:

1. the High Interface implements system specific protocols,
2. the Communication Core gathers various sets of standard protocols and services,
3. the Low Interface deals with network drivers and low level functions.

The following subsection summarizes the transmission protocols. CHORUS specific protocols are then outlined.

3.6.1.1 DATA TRANSMISSION PROTOCOLS Concerning data transmission, the Network Manager currently sticks to current international standards; two protocol families are implemented: the OSI protocols and the Internet family. As the Network Manager needs only one means to carry data from one place on the network to another, it uses only protocols up to Transport level for this particular function.

The current version uses OSI protocols to support network-wide IPCs and RPCs.

RATIONALE. The OSI choice results from the CHORUS philosophy to follow existing standards whenever they can be applied. However, such a choice can be complemented or changed according to the characteristics of the supporting network, application needs, etc., and IPC and RPC can use any protocol implemented in the Network Manager Communication Core, as long as it provides reliability and data ordering.

ISO Transport Protocol has been complemented by TCP and even UDP.[4]

3.6.1.2 SYSTEM SPECIFIC PROTOCOL These protocols are: the localization protocol, the connection management protocol if required, the RPC protocol and the interface with the Nucleus.

The localization protocol is in charge of finding the network host on which the destination port of a message lies, when the Nucleus passes a message it cannot deliver itself to the Network Manager. To do so, the Network Manager manages a cache of known ports and groups. When a site (i.e., a remote Network Manager) sends back a negative acknowledgement on reception of a message caused by the destination port migration, crash, or movement off network domain, the local Network Manager enters a search phase: it consists of a simple query protocol which uses the network broadcast facilities if they are supported by the underlying medium.[5] The idea of Network Manager operations is based on the assumption that, while a port or group is not in the localization cache, it is supposed to be still on the site where it was created.

The High Interface also implements an error handling protocol. For example, this is used to notify the RPC thread when a host is unreachable for some reason (network congestion, remote site failure, etc.) and the request cannot be satisfied.

3.6.2 The Network Manager as a Communication Channel Server

Some actors may know of the network organization and thus, want to use directly communication services provided by the protocols implemented in the Communication Core, knowing the semantics of this particular service. This is the case of the 4.3BSD-like Socket Server. The Network Manager also provides access to this kind of actors, through a Generic Connection Access

4. In the latter case, the service offered needs to be upgraded to provide reliability and data sequencing.
5. If "hardware" broadcast is not supported (e.g on top of connection-oriented communication services), broadcast can be simulated on top of the current service, at the price of more overhead.

Interface. In this case, the Network Manager will manage a communication channel to the actor's peer using the specified service. This is a lightweight interface, as the Network Manager assumes that the local actor knows perfectly well the semantics of the service it uses: its role is limited to network resources management and protocol-specific packet formatting.

3.7 Hardware Events and Exception Handling: the Supervisor

The CHORUS Nucleus is intended to support various subsystems (with various device handling strategies) and real-time applications. Giving system programmers direct access to exception handling and low-level I/O provides the required flexibility in handling hardware events.

A dedicated Nucleus component, the **supervisor**, provides an interface allowing "user" handler routines to be connected to interrupt levels.

When an interrupt occurs, the supervisor:

- saves the context interrupted,
- sequentially calls the (priority-ordered) routines attached to the the corresponding level (the handlers being able to force breaking the sequence),
- initiates rescheduling if necessary.

Similarly, events such as software traps or exceptions may also be directly processed by actor handlers. This allows subsystem managers to provide efficient and protected subsystem interfaces. This facility is used in particular by the UNIX process manager described in §4.2.2.

Note that this connection is dynamic, therefore I/O driver actors or subsystem managers may be inserted or removed dynamically, while the system is running.

3.8 Subsystems

A subsystem is an operating system built on top of a CHORUS Nucleus. The user of a subsystem has generally no direct access to the Nucleus interface. Using the abstractions offered by the

Nucleus, subsystems implement their own process semantics, their own protection mechanisms, etc.

A subsystem is made of:

- a set of subsystem actors (e.g., file managers, name managers, etc.),
- an integrated subsystem interface (the subsystem system calls).

The CHORUS Nucleus offers *system programmers* the means to construct **protected** subsystem interfaces. A protected subsystem interface is built by connecting its interface routines behind "system traps." The corresponding code and data structures are loaded into *system space* and, because they are hidden behind traps, they are accessible *only* via the subsystem interface (Figure 12).

RATIONALE. The level of protection that an operating system must offer depends highly on the class of applications it is intended to support. High levels of protection are often desirable, but they may be considered too expensive for certain classes of applications – e.g., real-time systems.

The CHORUS Nucleus itself does not provide a high level of protection. However it offers a basic level of protection and tools for subsystems to enforce higher levels of protection.

4. UNIX as a CHORUS Subsystem

4.1 Overview

The first Subsystem implemented within the framework of the CHORUS Architecture has been a UNIX Subsystem. The facilities provided by the CHORUS Nucleus have allowed designing coherent extensions of UNIX for distributed computing.

The implementation of the abstractions of this UNIX extended interface are described in the following sections. Some of the abstractions are already implemented by the CHORUS Nucleus and are provided by CHORUS Nucleus calls. Others are implemented in terms of CHORUS actors. The main design decisions are given and the general architecture is presented.

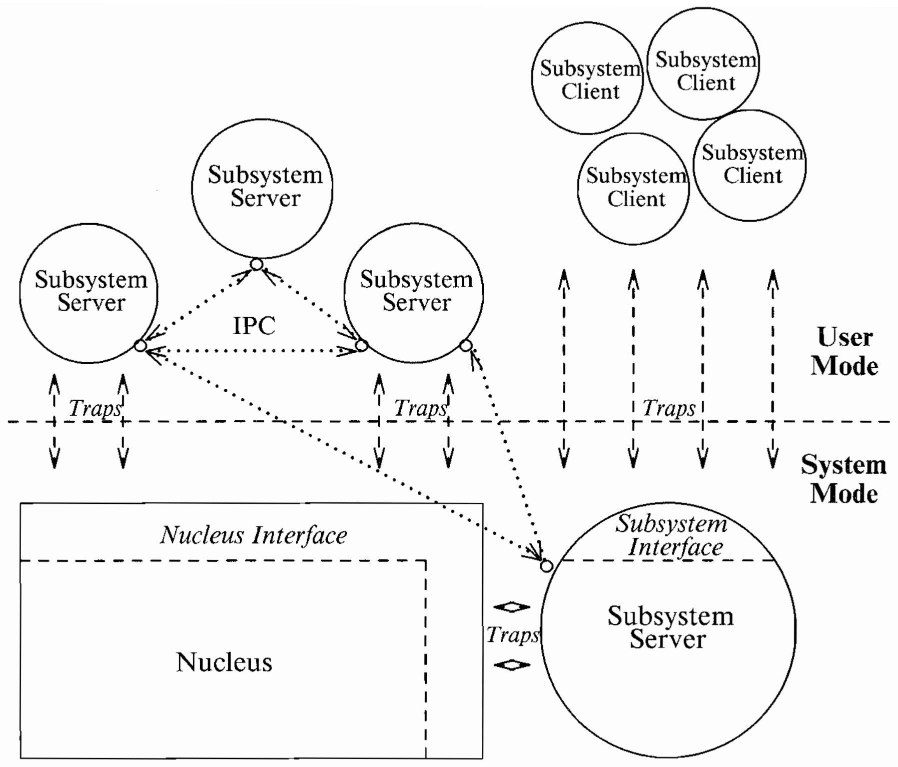


Figure 12: Structure of a Subsystem

Several *user actors* (subsystem clients) are shown using a *protected subsystem*. The subsystem interface is "protected" from its users by being placed in the "system" address space.

The bulk of the subsystem is located in separate user spaces, as *system actors*: no subsystem code or data can be manipulated directly by the "users." Communication between the subsystem interface routines and the subsystem actors is via the IPC. The subsystem actors directly call the protected Nucleus.

Some implementation choices are explained in more detail, the emphasis being on solving problems which arise when introducing distributed processing into a UNIX system.

4.1.1 Objectives

The CHORUS technology applied to UNIX covers a number of well recognized limitations of current "traditional" UNIX implementations. It has been applied with the following general objectives.

Modularity To implement UNIX services as a collection of servers, so that some may be present only on some sites (such as File Managers, Device Managers), and when possible build them in such a way that they can be dynamically (without stopping the system) plugged into/out of the system when needed. This true modularity will allow simpler modifications and maintenance because the system is built of small pieces with well-known interactions.

Openness and Expandability To permit application developers to implement their own servers (e.g., Time Manager, Window Manager, fault-tolerant File Manager) and to integrate them dynamically into the UNIX Subsystem.

Extending UNIX Functionalities Towards

- Real-Time:
  To extend UNIX with the real-time facilities provided by the low-level CHORUS Real-Time Executive.
- Distribution:
  To operate UNIX in a distributed environment with no limitations on the types of resources shared.
- Multiplexed Processes:
  To extend UNIX services with services provided by the underlying CHORUS Nucleus, e.g., multi-threaded UNIX processes.

Orthogonality To keep UNIX specific concepts out of the CHORUS Nucleus. But in turn, to use CHORUS concepts (Actors, Threads, Ports, etc.) to implement UNIX ones outside of the CHORUS Nucleus. This allows other subsystems (OS/2, Object Oriented Systems, etc.) to be implemented also on top of the CHORUS Nucleus without interfering with the particular UNIX philosophy.

Compatibility

- for application programs:
  On a given machine, to be compatible at the executable

code level with a given standard UNIX system (e.g., System V Release 3.2 on a PC AT-386), to ensure complete user software portability.

- for device drivers:
  To be able to adapt a UNIX driver into a UNIX Server on CHORUS with a minimum effort.
- regarding performance:
  To provide the same services about as fast as a given UNIX system on the same machine architecture (i.e., the one chosen for binary compatibility).

4.1.2 Extensions to UNIX Services

Distribution management with basic CHORUS concepts

- The file system is fully distributed and file access is location independent. File trees can be *automatically interconnected* to provide a name space where all files, whether remote or local, are designated with homogeneous and uniform symbolic names, and in fact with no syntactic change from current UNIX.
- Operations on processes (at the *fork/exec* level as well as at the *shell* level) can be executed regardless of the execution site of these processes; on the other hand, the creation of a child process can be forced to occur on any given compatible site.
- The network transparent CHORUS IPC is accessible at the UNIX interface level, thus allowing the easy development of distributed applications within the UNIX environment. Distribution extensions to standard UNIX services are provided in a natural way (in the UNIX sense) so that existing applications may benefit from those extensions even when directly ported onto CHORUS, *without modification or recompilation.* This applies not only to file management but also to process and signal management.

Multiprogramming a UNIX process Multiprogramming within a UNIX process is possible with the concept of **U_thread**. A U_thread can be considered as a lightweight process within a standard UNIX process. It shares all the process' resources and in particular its virtual address space and open files. Each U_thread represents a different locus of control. Thus when a process is created by a *fork*, it starts running with a unique U_thread; the same situation occurs after an *exec*; when a process terminates by *exit*, all U_threads of that process terminate with it.

With each U_thread is associated a list of signal handlers. Depending on their nature, signals are delivered to one of the process U_threads (alarm, exceptions,...) or broadcast to all process U_threads (DEL, user signals,...). The U_thread concept, derived from the CHORUS thread concept, is defined by five system calls (Table 4): *create*, *delete*, *start*, *stop* and *prio* and it has some information attached to it, comprising:

- an identification of the CHORUS thread implementing the U_thread,
- the identification of the owner process,
- a list of associated signal handlers.

`u_threadCreate`	*create a U_thread*
`u_threadDelete`	*delete a U_thread*
`u_threadStop`	*stop a U_thread*
`u_threadStart`	*restart a U_thread*
`u_threadPrio`	*modify U_thread priority*

Table 4: UNIX Threads System Calls

Interprocess communication and U_thread synchronization Interprocess communication and U_thread synchronization rely on the CHORUS IPC functionalities (ports, port groups, messages).

Real-time facilities with priority based scheduling and interrupt handling These facilities result directly from the services provided by the CHORUS Nucleus for real-time handling.

4.2 The UNIX Subsystem Architecture

UNIX functionalities may logically be partitioned into several classes of services according to the different types of resources managed: processes, files, devices, pipes. The design of the structure of the UNIX Subsystem gives emphasis on a clean definition of the interactions between these different classes of services in order to get a true modular structure.

The UNIX Subsystem has been implemented as a set of System Servers, running on top of the CHORUS Nucleus. Each system resource (process, file, etc.) is isolated and managed by a dedicated system server. Interactions between these servers are based on the CHORUS IPC which enforces clean interface definitions.

Several types of servers may be distinguished within a typical UNIX subsystem: Process Managers (PM), File Managers (FM), Pipe Managers (PIM), Device Managers (DM) and User Defined Servers (UDS) (Figure 13).

The following sections describe the general structure of UNIX servers. The role of each server and its relationships with other servers are summarized.

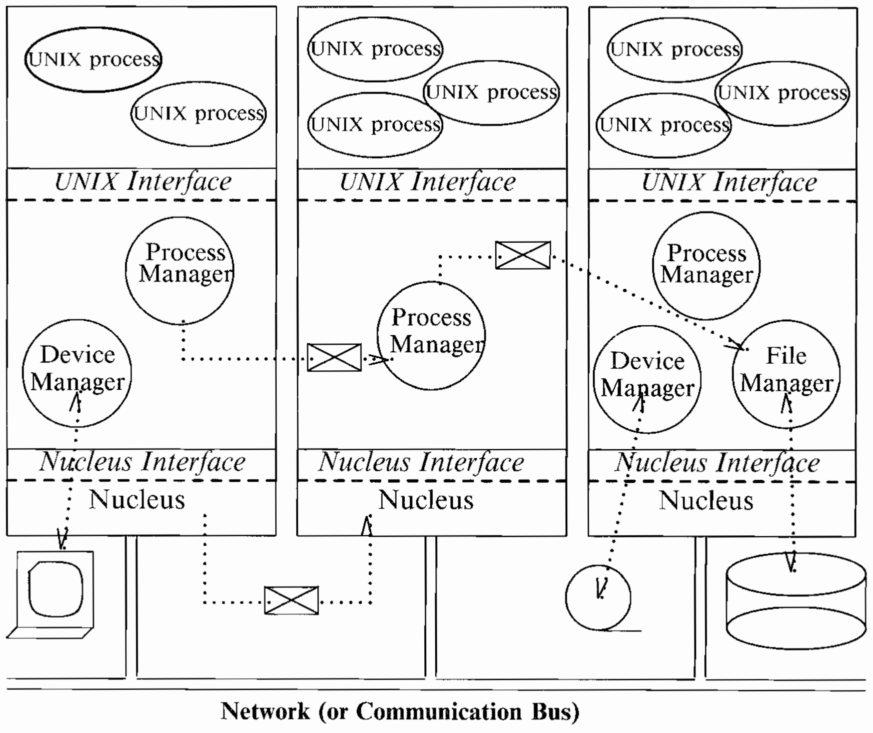


Figure 13: UNIX with the CHORUS Technology

4.2.1 Structure of a UNIX Server

Server = Actor UNIX Servers are implemented as CHORUS Actors. They have their own context (virtual memory regions, ports, etc.) and thus may be debugged as "standard" actors.

Mono or Multi-Threaded Most servers – File Manager, Device Manager, Process Manager – are multi-threaded. Their threads may be executed either in "user mode" or in – protected – "system mode." Each request to a server is processed by one thread of this server which manages the context of the request until the response is complete.

Simpler servers may be mono-threaded (e.g., the Pipe Manager). When a request cannot be served immediately, it is queued by the server, in a private queue. Thus the server is not blocked waiting for an event in order to complete a request. When the server detects that an expected event has occurred, it processes the requests in the queue.

Accessible via Ports or Traps Each server creates one or more ports which clients send requests to. Some of these ports may be inserted into port groups with well-known names. Such port groups can be used to access a service independently of the server which will actually provide it.

In order to provide compatibility with existing UNIX system interfaces, servers may also attach some of their own routines to system traps.

Implementation of a Service A service (e.g., *open(2)*) is realized partly by a "stub" which executes, in "system mode" (behind a trap), in the context of the calling client process. This routine manages the context of the process and when needed, invokes the appropriate Subsystem server via the CHORUS IPC.

Drivers In order to facilitate porting drivers from a UNIX kernel into a CHORUS server, a UNIX kernel emulation library, to be linked with UNIX driver code, has been developed with such functions as *sleep*, *wakeup*, *splx*, *copyin*, *copyout*, etc.

Interrupt handlers are about the only parts which have to be adapted to the CHORUS environment.

4.2.2 Process Manager

The Process Manager maps UNIX process abstractions onto CHORUS concepts (actor, thread, regions, etc.). It implements all of the UNIX process semantics including process creation, context consistency and inheritance, and process signaling.

On each "UNIX site," a Process Manager implements the entry points used by user processes to access UNIX services (other UNIX servers are not accessed directly by user processes). To maintain binary compatibility, these services are accessed through traps. The Process Manager uses the CHORUS Nucleus facility to attach routines to these traps (see §3.7).

Those routines connected to traps either call other Process Manager routines to satisfy requests related to process and signal management (*fork*, *exec*, *kill*, etc.) or invoke via RPC File Managers, Pipe Managers or Device Managers to handle other requests (e.g., *read/write* on files).

Process Managers interact with their environment through clearly defined interfaces:

- Nucleus services are accessed through system calls,
- File Manager, Pipe Manager, and Device Manager services used for process creation and context inheritance are accessed by means of the CHORUS IPC.

Process Managers cooperate to implement remote execution and remote signaling:

- Each Process Manager dedicates a port (the *request* port) to receive remote requests. Those requests are processed by Process Manager threads.
- The request port of each Process Manager (of a given CHORUS domain) is inserted into one port group. Any Process Manager of any given site may thus be reached through one unique functional address: the port group name.

4.2.3 File Manager

There is one File Manager on each site supporting a disk. Diskless stations don't need a local File Manager. File Managers have two main functions:

- to provide UNIX File System management (compatible at disk level),
- to act as a Segment Server (called *Mapper*) to the CHORUS Nucleus, managing segments (i.e., files) mapped into Actors contexts (executable binary files, swap files, etc.).

As UNIX file servers, they process UNIX requests transmitted via IPC (*open(2)*, *stat(2)*, etc.). In addition, File Managers provide some new services needed for process management:

- sharing (on *fork(2)*) and releasing (on *exit(2)*) directories and files between processes,
- associating capabilities to text and data segments of executable files, used to create regions in process contexts.

These two services represent the only interactions between process and file management.

As external Mappers, File Managers implement services required by the CHORUS virtual memory management: swapping pages in/out, creating swap files, etc. They use the standard Mappers Interface services provided by the CHORUS Nucleus to control and to keep consistent the data transferred between CHORUS sites when local virtual memory caches are involved: flush, invalidate pages, etc.

To avoid maintaining unnecessary copies of pages of a given file in physical memory, and to optimize physical memory allocation, CHORUS virtual memory mechanisms are used to implement file system caches and some of the UNIX calls such as *read(2)*, *write(2)*, etc.

Naming Extension The naming facilities provided by the UNIX file system have been extended, to permit the designation of services accessed via Ports.

Symbolic Port Names (new UNIX file type) can be created in the UNIX file tree (Figure 14). They associate a file name to a port Unique Identifier (this is very similar to UNIX device designation). When such a name is found during the analysis of a pathname, the corresponding request is forwarded to the port to which the Unique Identifier is associated – marked with the current status of the analysis (see §4.4.1).

Servers can be designated by such symbolic port names. In particular, this functionality is used to interconnect file systems

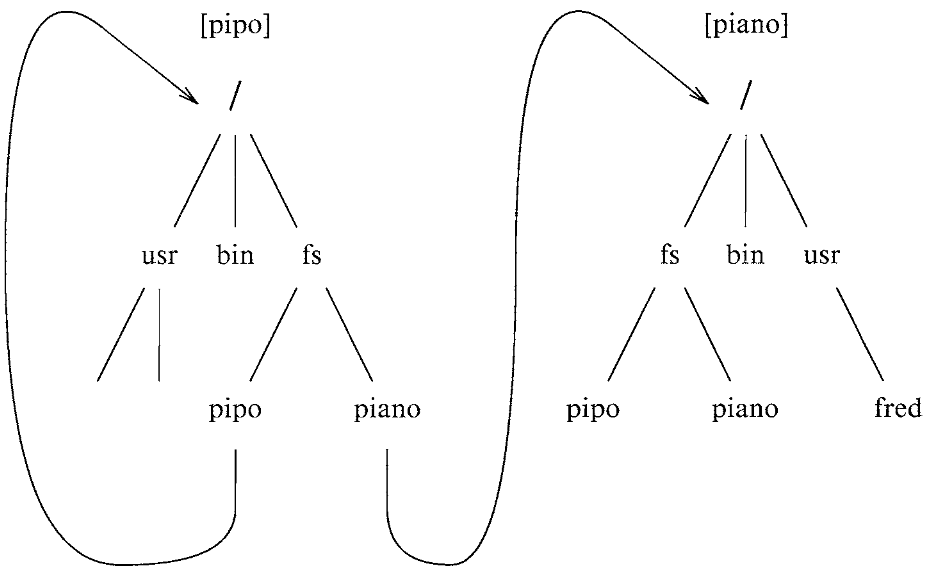


Figure 14: File Trees Interconnection

and provide a global name space. For example, in Figure 14, *pipo* and *piano* are symbolic port names.

Transparent access to remote files is provided through symbolic links (â la BSD). When such a symbolic name is found during the analysis of a pathname, the pathname is transformed into a new one and the analysis is restarted. The File System generates the new pathname by prefixing the not-yet analyzed part of the pathname with the value of the symbolic name – this value is also a pathname.

4.2.4 Pipe Manager

A Pipe Manager implements UNIX pipe management and synchronization. It cooperates with the File Managers to manage named pipes. Pipe management is no longer done by File Managers as in "usual" UNIX kernels. Pipe Managers may be active on every site, thus reducing network traffic when pipes are invoked on diskless stations.

4.2.5 Device Manager

Devices such as ttys and pseudo-ttys, bitmaps, tapes, network interfaces are managed by Device Managers. Needed (respectively not used) drivers can thus be loaded (respectively unloaded) dynamically (i.e., while the system is running). Software configurations can be adjusted to accommodate the use of the system or the local hardware configuration. For example, a bitmap driver might be present on a diskless station; a File Manager might not be.

A CHORUS IPC-based facility is used by these drivers to cooperate with the UNIX servers instead of the original UNIX *cdevsw* mechanism. When starting up, these drivers tell the File Manager which type of devices (i.e., which *major* number) they manage and the name of the port on which they want to receive *open(2)* requests.

4.2.6 User Defined Servers

The homogeneity of server interfaces provided by the CHORUS IPC allows "system users" to develop new servers and to integrate them into the system as user actors. One of the main benefits of this architecture is to provide a powerful and convenient platform for the experimentation of system servers: new file management strategies, fault-tolerant servers, etc., can be developed and tested just like user level utilities, without disturbing a running system.

4.3 Structure of a UNIX Process

CHORUS abstractions are simple and general. They have been designed to fit the needs of various systems which can be built using them. This section describes how the UNIX process concept has been implemented using CHORUS abstractions.

A UNIX process can be viewed as one thread of control executing within one address space. Each UNIX process is therefore implemented as one CHORUS actor. Its UNIX system context is managed by a Process Manager. The actor address space is structured into memory regions for text, data and execution stacks.

In addition, the Process Manager attaches one *control port* and one *control thread* to each actor implementing a UNIX process.

The control port and the control thread are not visible to the user of that process. Control threads executing within process contexts have two main properties:

- they share the process address space and can easily access and modify the core image of the process (stack manipulations on signal reception, text and data access during debugging, etc.).
- they are ready to handle asynchronous events received by the process (mainly signals). These events are implemented as CHORUS messages received on the control port (Figure 15).

Even if standard UNIX processes are mono-thread, the UNIX subsystem on CHORUS has been extended to allow multiprogramming within a process. This facility translates at the UNIX level the CHORUS multi-thread facilities useful for implementing

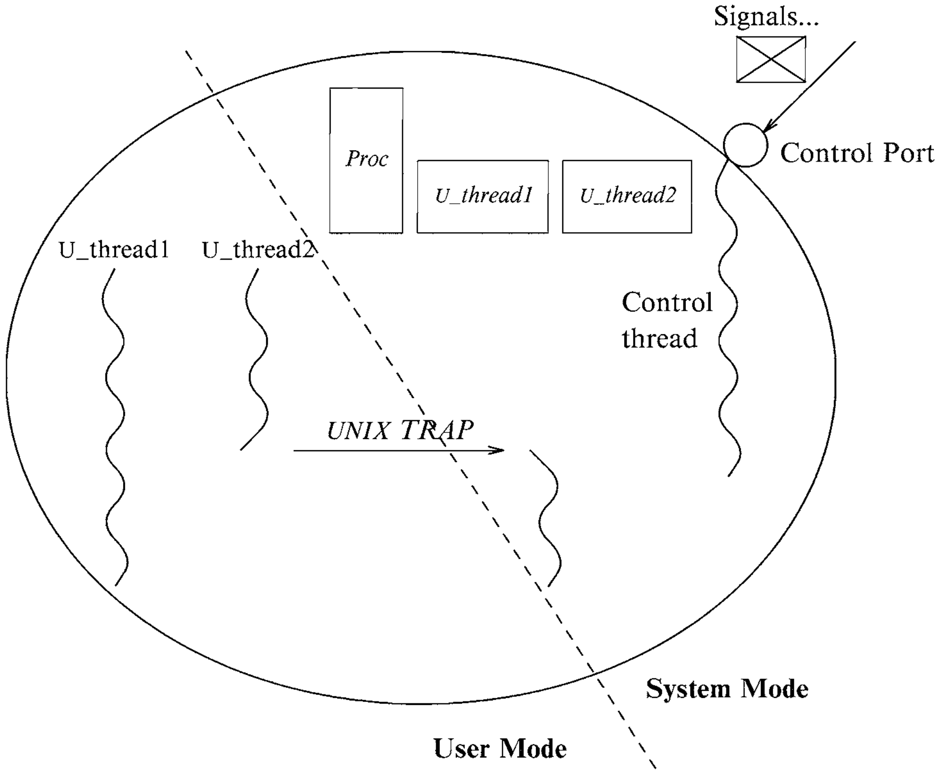


Figure 15: UNIX Process as a CHORUS Actor

multiplexed servers. The semantics of such UNIX threads (called *U_threads*) differ slightly from "pure" CHORUS threads, in particular regarding signal handling, time accounting, system call multiplexing, thus their slightly different name.

Providing multi-threaded processes has impacted the process implementation in three ways:

- signal processing is done on a per thread basis,
- blocking system calls (*pause(2)*, *wait(2)*, *read(2)*, etc.) may be multiplexed within a given process,
- the UNIX system context attached to one process has been split into two system contexts: one process context (*Proc*) (Table 5) and one U_thread context (*U_thread*) (Table 6).

Actor implementing the Process	*actor name, actor priority, etc.*
Unique Identifiers (UI)	*PID, PGRP, PPID, etc.*
Protection Identifiers	*real PID, effective PID, etc.*
Ports	*control port, parent control port, etc.*
Memory Context	*text, data, stack, etc.*
Child Context	*SIGCLD handler, creation site, etc.*
File Context	*root and current directory, open files, etc.*
Time Context	*user time, child time, etc.*
Control Context	*debugger port, control thread descriptor, etc.*
U_threads	*list of process U_threads,*
Semaphore	*for concurrent access to Proc Context.*

Table 5: Process Context

Thread implementing the U_thread	*thread descriptor, priority, etc.*
Owner Process	
Signal Context	*signal handlers, etc.*
System Call Context	*system call arguments, etc.*
Machine execution Context	

Table 6: U_Thread Context

The two system contexts *Proc* and *U_thread* are maintained by the Process Manager present on the current process execution site. These contexts are accessed neither by the CHORUS Nucleus nor by other system servers. On the other hand the UNIX subsystem has no visibility of the internal Nucleus structures associated with actors and threads; the only way to access them is through

Nucleus system calls (this is essential for allowing different subsystems to co-reside on top of the same CHORUS Nucleus).

4.3.1 Process Identifiers

Each process is uniquely designated by a Unique Identifier (UI). This enables Process Managers to take advantage of Nucleus localization facilities when they look for one particular process (signaling, debugging, etc.).

For compatibility reasons, this process UI is not directly used as a process PID – which has a different size – but it is used to generate 32 bits global PIDs. The result is a concatenation of two 16 bit integers:

- the site where the process has been created.
- the value of a per site counter incremented at each process creation.

Process Managers convert PIDs into process UIs (and conversely). They use one or the other depending of the needs (CHORUS IPCs or system call interface).

4.3.2 Process Execution Site

Part of each process context is the child creation site information. Inherited on process creation (*fork*), this information extends standard UNIX *fork* and *exec* operations with distribution facilities. If the child creation site of a process is set to another site than the current execution site, later *fork* and *exec* calls will be applied on this remote site.

4.3.3 Process Environment Known by its Set of Ports

The semantics associated by the CHORUS Nucleus to the port concept – unique and global naming, addressing by IPC with location transparency – make ports extremely useful for system entities designation. The main advantages of using ports are the indirection they provide between the process and its environment, and the robustness against the evolutions of configuration. Port names stored in the process context are always valid either if the process itself migrates to another site (i.e., *exec* on a remote site) or if some of the entities which it is related to migrate.

Used directly or embedded within capabilities, ports constitute the main part of a process environment. Embedded in capabilities, ports are used to designate process resources: open files, segments mapped into process address space (*text*, *data*), etc. But ports are also used directly to address processes.

Resources and capabilities Every resource (managed by a Server) used by a Process is designated internally by a capability: open file, open pipe, open device, current and root directories, text and data segments, etc. Such capabilities may be used to create regions in virtual memory; thus their structure is the one exported by the CHORUS Nucleus.

For example, opening a file associates the capability sent back by the appropriate server to the correct file descriptor. The capability will be built with:

- the port of the server that manages that file,
- the reference of this open file within the server.

Thus, all requests on that open files – *seek(2)*, *close(2)*, etc. – are translated directly into a message and sent directly to the appropriate server. There is no need to locate the server again.

In particular, as the server of a resource is designated by a port, and as the localization of a port is part of the CHORUS IPC, the UNIX subsystem does not have to localize UNIX servers.

Processes and system ports Process Managers attach some "system" ports (to distinguish from the ports created by the process) to each process in order to implement some UNIX services regardless of the process' location:

- Each process is created with a *Control Port* on which the control thread receives messages. The parent process control port UI is also stored in the process context. When a process exits, a message filled with all needed information (exit status, elapsed time, etc.) is sent to the parent's control port. When received by the parent control thread, this information is stored in the parent *Proc* context until the parent process attempts to obtain them – using the *wait(2)* UNIX system call.
- Signals are implemented as messages. Process Managers, acting as intermediaries for process localization, forward these messages to the control port of the target process(es)

(these messages will be processed by the process control thread).

- When a debugging session starts, the debugged process creates a debug port and sends this port name to the debugger. All interactions between the debugger and debugged processes rely on CHORUS IPC between the debug port of the debugged process and the control port of the debugger. Because it is based on the CHORUS IPC, UNIX debugging functionality is distributed – debugged and debugger processes can be on different sites.

4.4 Two Examples

4.4.1 File Access

Current and root directories are represented by capabilities in the UNIX context of a process. When an *open(2)* request is issued, the open routine of the Process Manager looks for a free file descriptor, builds a message containing the pathname of the file to be opened, and sends this message to the port of the server managing the current or the root directory depending on whether the pathname is absolute or relative [1].

Suppose that the pathname of the file is (Figure 14): "*fs/piano/usr/fred/myfile*" and "*piano*" is the symbolic port of a File Manager. This pathname will be sent to the File Manager on which the root directory of the process is located (let's call it "*pipo*"). That File Manager will start the analysis of the pathname, find that "*piano*" is a symbolic port, and forward the message with the rest of the pathname ("*usr/fred/myfile*"), not yet analyzed, to the port whose UI is in the "*piano*" *i-node* [2].

The "*piano*" File Manager will receive the message, complete the analysis of the pathname, open the file, build the associated capability and send it back directly to the client process that issued the *open(2)* request [3].

Afterwards, any subsequent request on that open file (*seek(2)*, *close(2)*, etc.) will be sent directly to the File Manager on "*piano*" thus avoiding any indirection through the *pipo* File Manager responsible for the root directory of the process (Figure 16).

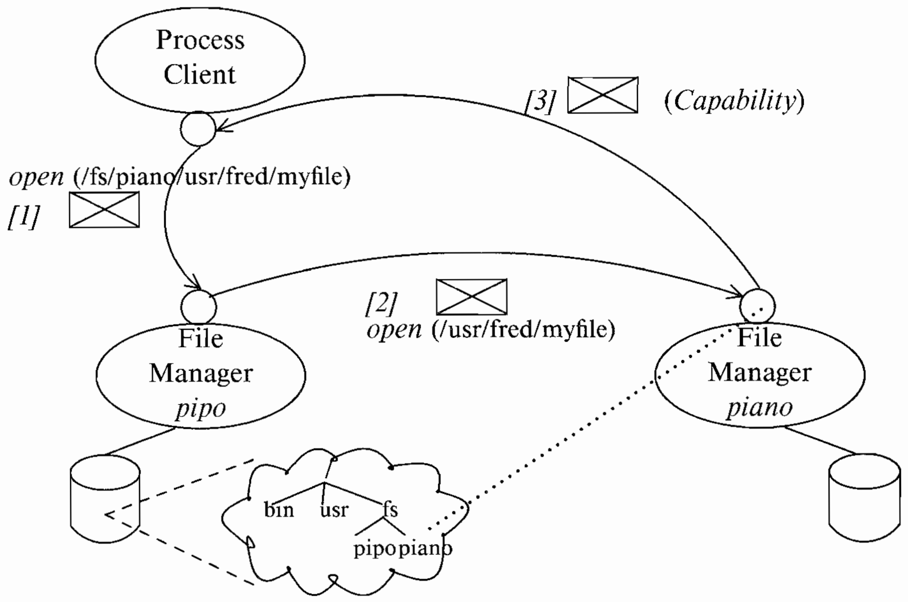


Figure 16: File Access

4.4.2 Remote Exec

This description of the remote *exec* algorithm will illustrate all the interactions between the UNIX subsystem servers and the process control threads. To simplify the description, error cases are not handled in this algorithm (Figure 17).

1. The calling U_thread performs a "Trap" handled by the local Process Manager. The PM will:
    - invoke by RPC *[1]* & *[2]* the File Manager to translate the binary file pathname into two capabilities, used later on to map text and to map data into the process address space. Depending on the pathname, the File Manager of the root directory or of the current directory is invoked.
    - test child execution site (in this example, child execution site is different from current execution site),

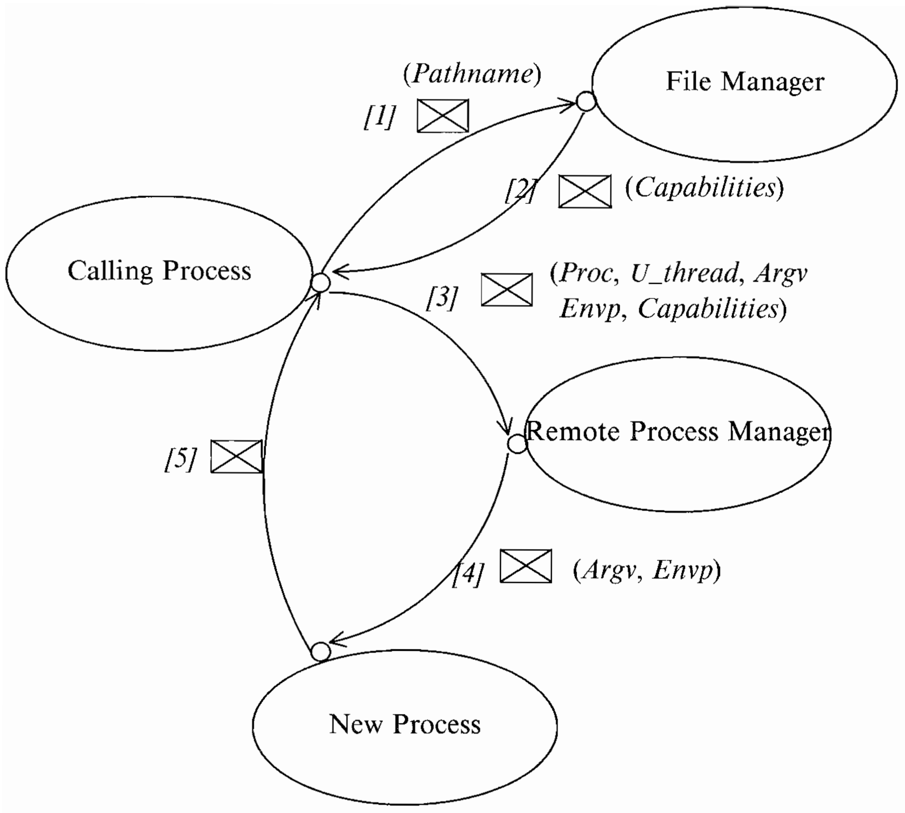


Figure 17: Remote Exec

- prepare a request with:
  - *Proc* and *U_thread* context of calling U_thread,
  - arguments and environments given as *exec* parameters, and
  - all information returned by the File Manager which characterizes the binary file.
- perform an RPC *[3]* to the Process Manager of the target creation site determined above – actually addressed via a Port group.

2. The Process Manager of the remote creation site receives the request and one of its threads processes it. This Process Manager thread initializes a new *Proc* context for the migrating process (much information, such as PIDs, elapsed

time, etc. are just copied from the request message) and creates new CHORUS entities which implement the process: actor, control thread, control port, memory regions, etc.

The rest of the initialization is then done by the control thread of the process which executes in the newly created process context. Also, the request message is forwarded to the control port of the process *[4]*.

3. The process control thread receives the forwarded message and follows the normal UNIX process initialization:
    - arguments and environment are installed in the process address space.
    - close messages are sent to appropriate File Managers, Pipe Managers and Device Managers to close any close-on-exec open files.
    - one U_thread is created which will start executing the new program (even if before *exec* the process was multi-threaded, the new process is always mono-threaded after). The signal context of the created U_thread is set up with signal context of calling U_thread present in the request message.
    - a reply message is sent back to the U_thread which had invoked the *exec(2)* system call and has blocked in the RPC *[5]* awaiting the reply message.
4. The calling thread receives the reply message, frees the *Proc* and *U_thread* contexts of its process and removes the actor implementing the process (on actor destruction, the CHORUS Nucleus removes all CHORUS entities attached to this actor: ports, threads, regions, etc.).

4.5 Other UNIX Extensions

The CHORUS implementation of the UNIX subsystem, has led to several significant extensions which offer, at the UNIX subsystem level, basic CHORUS functionalities.

4.5.1 IPC

UNIX processes running on CHORUS can communicate with other processes – and bare CHORUS actors or entities from other Subsystems – using the CHORUS IPC mechanisms. In particular, processes are able to:

- create and manipulate CHORUS ports,
- send and receive messages,
- issue remote procedure calls.

4.5.2 Real-Time

CHORUS real-time facilities provided by the Nucleus are available at the UNIX subsystem level to privileged applications:

- The ability to dynamically connect handlers to hardware interrupts. This facility is already used by UNIX Device Managers.
- The benefit of the priority based preemptive scheduling provided by the CHORUS Nucleus.

Moreover, two policies for interrupt processing may be used by UNIX servers:

- process entirely the interrupt in its handler or,
- just signal the event to a dedicated thread which will process it, allowing the code to be executed in interrupt handlers to be as short as possible.

This allows UNIX device drivers to have interrupt masked sections that are shorter than in many standard UNIX implementations. Thus, with a little tuning, real-time applications run on UNIX with better response time to external events.

4.5.3 UNIX Commands

Several UNIX commands (such as *ls(1)*, *find(1)*, *ln(1)*, *rm(1)*, etc.) have been extended also to cope with the new "Port" file type. A *shell* has been extended to give access to some CHORUS functionalities:

- remote execution using a "@ *site*" syntax, where *site* may be given as a symbolic or "functional" name (e.g., *lisp*, *laser-printer*), a given machine (or processor) being able to be associated to several of these "names."

- creating symbolic port names,
- sending, receiving messages between *shell*s.

For example, one can synchronize two *shell*s which may be running on different machines as shown in Figure 18.

```
# create a Port on the current
# shell.  give it the symbolic
# name "shport" in the
# ''/usr/fred'' directory:
bind /usr/fred/shport
# wait for message on that port:
receive /usr/fred/shport > msg_in
# reply to the sender:
reply < msg_out
```

```
# send a message
# to the other shell:
send "message" \
     /usr/fred/shport
```

Figure 18: Synchronizing *shell* Scripts by Means of Messages

5. Implementation

CHORUS has been designed to be highly portable on a wide range of modern hardware architectures. It is mostly written in C++ (and in C). The assembler code is less than 5%, in the hardware dependent parts.

CHORUS-V3 has been implemented on several different computer architectures, using different microprocessor families (MC680x0 and Intel 386 in particular), different bus architectures and different memory management units.

The initial UNIX implementation is X/OPEN compatible (System V).

The size of the Nucleus on a MC68020 varies from 29 Kb of code for a reduced version handling linear memory and local IPC only, to 55 Kb of code for a full Nucleus handling distributed virtual memory (for the MC68851/PMMU) and complete IPC.

Code sizes of the UNIX servers are given in Table 7.

Server	Size (Kb)
Network Manager	70
Process Manager	45
File Manager	55
Device Manager	30
Socket Manager	40

Table 7: Size of UNIX servers

Initial performance measurements on a Bull SPS 7/300 computer, based on a MC68020 (with MC68851) processor running at 16 MHz (with 2 wait-states) are given on Tables 8 and 9. Performance measurements of the CHORUS UNIX subsystem are related to SPIX, the Bull System V implementation.

Primitive	Time (μs)
IT processing latency	22
Thread scheduling (in system mode)	45
Thread scheduling (in user mode)	130
Synchronization ($V + P$)	56
send + *receive* (32b in system mode)	314
send + *receive* (32b in user mode)	526
RPC (32b in system mode)	857
RPC (32b in user mode)	1375

Table 8: Initial performance figures of the CHORUS Nucleus

Primitive	CHORUS	SPIX
getpid	77 μs	77 μs
write (1 Kb)	64 Kb/s	55 Kb/s
exec	27 ms	27 ms
fork (8 Kb)	22.2 ms	14.3 ms
fork (4 Mb)	100 ms	143 ms

Table 9: Initial performance figures of the CHORUS UNIX subsystem

Although made on a CHORUS system which is far from being optimized, these figures show a similar level of performance with a traditional UNIX implemented on a standalone machine. CHORUS real-time performance is also quite in line with that of other real-time executives. Complete performance measurements and analysis will be the subject of following papers. Those given here simply show that a true modular system can also exhibit a fairly good level of performance.

6. Conclusion

CHORUS was designed with the intention of being used in "real" life. Thus, the inherent trade-off between performance and richness of the design tended to get resolved in favor of performance.

Making the CHORUS Nucleus *generic* prevented the introduction of "features" with "heavy" semantics and prevented the use of research implementations that were not considered proven.

Features such as highly secure protection against intrusions, application-oriented protocols, and fault tolerant strategies, do not appear in the CHORUS Nucleus. However, CHORUS provides the building blocks to construct these features inside subsystems.

On the other hand, CHORUS provides effective, high performance solutions to some of the issues known to cause difficult problems to system designers:

- Errors and exceptions are posted by the Nucleus to a port chosen by the actor program. This very flexible mechanism allows the "user" actor to apply its own strategy to handling errors and exceptions, and, because of the nature of ports, it applies transparently to distributed systems.
- An actor failure has two kinds of consequences:
    1. at the communication level, attached ports are no longer valid. The Nucleus handles that "normally" when it needs to locate a port which may be present, or absent,
    2. at the service level, the service is no longer available and the subsystem will handle the failure at its own level.

  Similarly, introducing a new site is implicit at the Nucleus level and explicit at the service level. The service level can use the port group facility for managing services dynamically.

- Debugging CHORUS distributed systems is eased by isolating resources within actors and by communicating by means of messages which provide explicit and clear interactions.
- Different types of applications can be supported whether they make high or low demands on the communication bandwidth.
- The CHORUS modular structure was successful in all versions, allowing binary compatibility with UNIX in CHORUS-V3, while keeping the implementation well structured, portable, and efficient.

The experience of four CHORUS versions has clearly strengthened the role of the CHORUS concepts:

- Two concepts provide *stability* in the evolving state of a distributed system and are static, fixed references in a distributed application design:
    1. the *actor* is a local grouping of local entities, behaving as a whole and providing a local and precise state, as long as the actor exists,
    2. the *port group* is a global grouping of distributed entities behaving as a whole, statically defined while its members can evolve on their own.
- Two concepts provide *dynamism* and allow migration and evolution of information as well as computation – i.e., *messages* and *ports*.

The CHORUS technology has been designed for building "new generations" of open, distributed, and scalable Operating Systems. It has the following main characteristics:

- a communication-based technology, relying on a minimum Nucleus integrating distributed processing and communication at the lowest level, and providing generic services used by a set of subsystem servers to provide extended standard operating system interfaces – e.g., UNIX,
- real time services provided by the real-time Nucleus, and accessible by "system programmers" at the different system levels,

- a modular architecture providing scalability, and allowing in particular dynamic configuration of the system and its applications over a wide range of hardware and network configurations, including parallel and multiprocessor systems.

While transparently extending existing applications to run in distributed environments, such operating systems provide new services adapted to develop new applications, that can map better the distributed nature of organizations and therefore meet users' needs in a much better way.

7. *Acknowledgments*

Jean-Pierre Ansart, Philippe Brun, Hugo Coyote, Corinne Delorme, Jean-Jacques Germond, Steve Goldberg, Pierre Lebée, Frédéric Lung, Marc Maathuis, Denis Metral-Charvet, Bruno Pillard, Didier Poirot, Eric Pouyoul, François Saint-Lu and Eric Valette contributed, each with a particular skill, to the CHORUS-V3 implementation on various machine architectures.

Hubert Zimmermann by initiating the Chorus research project at INRIA, encouraging its development, and leading its transformation into an industrial venture made all this possible.

8. *CHORUS Bibliography*

8.1 *CHORUS-V0*

Jean-Serge Banino, Alain Caristan, Marc Guillemont, Gérard Morisset, and Hubert Zimmermann, CHORUS: an Architecture for Distributed Systems, Research Report, INRIA, Rocquencourt, France (November 1980).

Jean-Serge Banino and Jean-Charles Fabre, Distributed Coupled Actors: a CHORUS Proposal for Reliability, page 7 in *IEEE 3rd. International Conference on Distributed Computing Systems Proc.*, Fort Lauderdale, FL (18-22 October 1982).

Marc Guillemont, Intégration du Système Réparti CHORUS dans le Langage de Haut Niveau Pascal, Thèse de Docteur Ingénieur, Université Scientifique et Médicale, Grenoble, France (Mars 1982).

Marc Guillemont, The CHORUS Distributed Operating System: Design and Implementation, pages 207-223 in *ACM International Symposium on Local Computer Networks Proc.*, Florence, Italy (April 1982).

Hubert Zimmermann, Jean-Serge Banino, Alain Caristan, Marc Guillemont, and Gérard Morisset, Basic Concepts for the Support of Distributed Systems: the CHORUS Approach, pages 60-66 in *IEEE 2nd. International Conference on Distributed Computing Systems Proc.*, Versailles, France (April 1981).

8.2 CHORUS-V1

Jean-Serge Banino, Jean-Charles Fabre, Marc Guillemont, Gérard Morisset, and Marc Rozier, Some Fault-Tolerant Aspects of the CHORUS Distributed System, pages 430-437 in *IEEE 5th. International Conference on Distributed Computing Systems Proc.*, Denver, CO (13-17 May 1985).

Jean-Serge Banino, Gérard Morisset, and Marc Rozier, Controlling Distributed Processing with CHORUS Activity Messages in *18th. Hawaii International Conference on System Science*, Hawaii (January 1985).

Jean-Charles Fabre, Un Mécanisme de Tolérance aux Pannes dans l'Architecture Répartie CHORUS, Thèse de Doctorat, Université Paul Sabatier, Toulouse, France (Octobre 1982).

Marc Guillemont, Hubert Zimmermann, Gérard Morisset, and Jean-Serge Banino, CHORUS: une Architecture pour les Systèmes Répartis, Rapport de Recherche, INRIA, Rocquencourt, France (Mars 1984).

Frédéric Herrmann, Décentralisation de Fonctions Système: Application à la Gestion de Fichiers, Thèse de Docteur Ingénieur, Université Paul Sabatier, Toulouse, France (Septembre 1985).

Ahmad Rozz, La Gestion des Fichiers dans le Système Réparti CHORUS, Thèse de Docteur Ingénieur, Université Paul Sabatier, Toulouse, France (Octobre 1985).

Christine Senay, Un Système de Désignation et de Gestion de Portes pour l'Architecture Répartie CHORUS, Thèse de Docteur Ingénieur, C.N.A.M., Paris, France (Décembre 1983).

Hubert Zimmermann, Marc Guillemont, Gérard Morisset, and Jean-Serge Banino, CHORUS: a Communication and Processing Architecture for Distributed Systems, Research Report, INRIA, Rocquencourt, France (September 1984).

8.3 CHORUS-V2

François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, Towards a Distributed UNIX System – the CHORUS Approach, pages 413-431 in *EUUG Autumn '86 Conference Proc.*, Manchester, UK (22-24 September 1986).

Frédéric Herrmann, CHORUS: un Environnement pour le Développement et l'Exécution d'Applications Réparties, *Technique et Science Informatique* **6**(2) pages 162-165 (Mars 1987).

José Legatheaux-Martins, La Désignation et l'Edition de Liens dans les Systèmes d'Exploitation Répartis, Thèse de Doctorat, Université de Rennes-1, Rennes, France (Novembre 1986).

José Legatheaux-Martins and Yolande Berbers, La Désignation dans les Système d'Exploitation Répartis, *Technique et Science Informatique* **7**(4) pages 359-372 (Juillet 1988).

Azzeddine Mzouri, Les Protocoles de Communication dans un Système Réparti, Thèse de Doctorat, Université Paris-Sud, Orsay, France (Janvier 1988).

Mario Papageorgiou, Les Systèmes de Gestion de Fichiers Répartis, Thèse de Doctorat, Université Paris-6, Paris, France (Janvier 1988).

Mario Papageorgiou, Le Système de Gestion de Fichiers Répartis dans CHORUS, *Technique et Science Informatique* **7**(4) pages 373-384 (Juillet 1988).

Marc Rozier, Expression et Réalisation du Controle d'Execution dans un Système Réparti, Thèse de Doctorat, Institut National Polytechnique,, Grenoble, France (Octobre 1986).

Marc Rozier and José Legatheaux-Martins, The CHORUS Distributed Operating System: Some Design Issues, pages 261-287 in *Distributed Operating Systems, Theory and Practice*, ed. Yakup Paker, Jean-Pierre Banâtre and Muslim Bozyigit, Springer Verlag, Berlin (1987).

8.4 CHORUS-V3

Frédéric Herrmann, François Armand, Marc Rozier, Michel Gien, Vadim Abrossimov, Ivan Boule Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, CHORUS, a New Technology for Building UNIX Systems in *EUUG Autumn '88 Conference Proc.*, Cascais, Portugal (3-7 October 1988).

Marc Rozier, and Michel Gien, Resource-level Autonomy in CHORUS in *1988 ACM SIGOPS European Workshop on "Autonomy and Interdependance in Distributed Systems?"*, Cambridge, UK (18-21 September 1988).

References

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, Mach: A New Kernel Foundation for UNIX Development, pages 93-112 in *USENIX Summer '86 Conference Proc.*, Atlanta, GA (9-13 June 1986).

Claude Bétourné, Jacques Boulenger, Jacques Ferrié, Claude Kaiser, Sacha Krakowiak, and Jacques Mossière, Process Management and Resource Sharing in the Multiaccess System ESOPE, *Communications of the ACM* **13**(12) (December 1970).

David Cheriton, The Unified Management of Memory in the V Distributed System, Technical Report, Computer Science, Stanford University, Stanford, CA (1988).

David Cheriton, The V Distributed System, *Communications of the ACM* **31**(3) pages 314-333 (March 1988).

Michel Gien, The SOL Operating System, pages 75-78 in *USENIX Summer '83 Conference*, Toronto, ON (July 1983).

Robert A. Gingell, Joseph P. Moran, and William A. Shannon, Virtual Memory Architecture in SunOS, pages 81-94 in *USENIX Summer '87 Conference*, Phoenix, AR (8-12 June 1987).

José Legatheaux-Martins and Yolande Berbers, La Désignation dans les Systèmes d'Exploitation Répartis, *Technique et Science Informatique* **7**(4) pages 359-372 (Juillet 1988).

Kai Li, Shared Virtual Memory on Loosely Coupled Multiprocessors, Ph.D. Thesis, Yale University, New Haven, CT (September 1986).

Joseph P. Moran, SunOS Virtual Memory Implementation, pages 285-300 in *EUUG Spring '88 Conference*, London, UK (11-15 April 1988).

Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers 1984 - 1987,* CWI Tract No. 41, Amsterdam, Netherlands (1987).

Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions on Computer Systems* **6**(1) pages 134-154 (February 1988).

Louis Pouzin et al., *The CYCLADES Computer Network - Towards Layered Network Architectures,* Elsevier Publishing Company, Inc, New York, NY (1982). ISBN 0-444-86482-2

David L. Presotto, The Eighth Edition UNIX Connection Service, page 10 in *EUUG Spring '86 Conference Proc.*, Florence, Italy (21-24 April 1986).

Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew, Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures, pages 31-39 in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)* (October 1987).

Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse, Using Sparse Capabilities in a Distributed Operating System, pages 558-563 in *IEEE 6th. International Conference on Distributed Computing Systems*, CWI Tract No. 41, Cambridge, MA (19-23 May 1986).

Peter J. Weinberger, The Eighth Edition Remote Filesystem, page 1 in *EUUG Spring '86 Conference*, Florence, Italy (21-24 April 1986).

[*submitted Sept 20, 1988; revised Dec. 1, 1988; accepted Dec. 6, 1988*]