CS/TR-90-25.1

Overview of the CHORUS® Distributed Operating Systems

M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont
F. Herrmann, C. Kaiser[*]*, S. Langlois, P. Léonard, W. Neuhauser*

[*] Also at CNAM (Conservatoire National des Arts et Métiers), Paris

Overview of the CHORUS® Distributed Operating Systems

CS/TR-90-25.1

Chorus
SYSTEMES

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 architecture, relying on a minimal Nucleus which integrates distributed processing and communication at the lowest level, and which implements generic services used by a set of subsystem servers to extend standard operating system interfaces. A UNIX subsystem has been developed; other subsystems such as object-oriented systems are planned;
- a real-time Nucleus providing real-time services which are accessible to system programmers;
- 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 also describes the UNIX subsystem built with the CHORUS technology that provides:

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

This paper is a revised and updated version of an article published in *''Computing Systems'', The Journal of the Usenix Association, Volume 1, Number 4.*

1. INTRODUCTION

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

This evolution has introduced the following requirements for new system structures that are difficult to fulfill merely by assembling networks of cooperating systems:

- Separate applications running on different machines, often from different suppliers, using 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. A requirement exists for a higher-level coupling of applications.
- Applications often evolve by growing in size. Typically, this growth leads to distribution of programs to different machines, to treating several geographically distributed sets of files as a unique logical file, and to upgrading hardware and software to take advantage of the latest technologies. A requirement exists for a gradual on-line evolution.
- Applications grow in complexity and become more difficult to understand, specify, debug, and tune. A requirement exists for a straightforward underlying architecture which allows the modularity of the application to be mapped onto the operational system and which conceals unnecessary details of distribution from the application.

These structural properties can best be accomplished through a basic set of unified and coherent concepts which provide a rigorous framework that is well adapted to constructing distributed 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. The generic CHORUS Nucleus implements the real-time services required by real-time applications. Traditional operating systems are built as subsystems on top of the generic Nucleus and use its basic services. User application programs run in the context of these operating systems.

CHORUS provides a UNIX subsystem as one example of a host operating system running on top of the CHORUS Nucleus. UNIX programs can run unmodified under this subsystem, optionally taking advantage of the distributed nature of the CHORUS environment.

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-threaded 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 a traditional UNIX kernel has been merged with state-of-the-art operating system technology while still preserving its original semantics. It also gives examples of how its services can then be easily extended to handle distribution. Section 5 gives some concluding remarks.

Comments about some of the important design choices, often related to previous experience, are given in small paragraphs entitled ‘‘Rationale.’’ For the remainder of this document, terms found in **bold** are concisely defined in the glossary, terms found in `typewriter` refer to precise names of machines or system entities, and terms found in *italics* are used for emphasis.

2. BACKGROUND AND RELATED WORK

CHORUS was a research project on distributed systems at INRIA[1] in France from 1979 to 1986. Three versions were developed, referred to as CHORUS-V0, CHORUS-V1, and CHORUS-V2, based on a communication-oriented kernel[Zimm81, Guil82a, Zimm84, Rozi87]. The basic concept for handling distributed computing within CHORUS, for both system and application services, is that a Nucleus manages the exchange of messages between ports attached to actors.

While early versions of CHORUS had a custom interface, CHORUS–V2[Arma86] was compatible with UNIX System V, and had been used as a basis for supporting half a dozen experimental distributed applications. CHORUS-V3 is the current version developed by Chorus systèmes. It builds on previous CHORUS experience[Rozi87] and integrates many concepts from state-of-the-art distributed systems developed in several research projects, while taking into account constraints of the industrial environment.

The CHORUS-V3 message-passing Nucleus is comparable to the *V-system*[Cher88] of Stanford University, its distributed virtual memory and threads are similar to those of *Mach*[Acce86] of Carnegie Mellon University, its network addressing incorporates ideas from *Amoeba*[Mull87] of the University of Amsterdam, and its uniform file naming is based on a scheme similar to the one used in Bell Laboratories' 9th Edition UNIX[Pres86, Wein86].

This technology has been used to implement a distributed UNIX system[Herr88], 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 the combined experience from previous research done on packet switching computer networks, the Cyclades project[Pouz82], and time sharing operating systems, the Esope project[Bét70]. The goal was to incorporate distributed control techniques, originating from packet switching networks, into distributed operating systems.

In 1979 INRIA launched the Sol project whose goal was to re-implement a complete UNIX environment on French micro and mini computers[Gien83]. 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:

- The operation of an actor, which was an alternating sequence of indivisible execution phases and communication phases. It provided a message-driven automaton style of processing.
- 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 to allow port migration and dynamic reconfiguration of applications.
- 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 concepts proved to be sound and have been maintained in subsequent versions.

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

These CHORUS concepts have been applied in particular to fault tolerance: the ‘‘coupled actors’’ scheme[Bani82] 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 multiprocessor micro-computers – based on Motorola 68000 and later 68020 – interconnected by a 10Mb/s Ethernet. In a multi-processor configuration, one processor ran UNIX as a development system and disk manager, and up to seven other processors ran CHORUS with one CHORUS system controlling 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 a few changes from CHORUS–V0, namely:

- Structured messages were introduced to allow embedding protocols and migrating their contexts.
- The concept of an activity message, which embodies data, a context for embedded computations, and a graph of future computations, was experimented with for a fault tolerant application.[Bani85]

CHORUS–V1 was running in mid-1984. It was distributed to a dozen places, some of which still use the system.

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, changed very little. The UNIX subsystem was developed partly from results of the Sol project (the File Manager), and partly from scratch (the Process Manager). Concepts such as ports, messages, processing steps, and remote procedure calls were revisited. Changes were made to more closely match UNIX semantics and to allow a protection scheme à la UNIX. The UNIX interface was extended to support distributed applications by means of new functionalities such as distant `fork`, distributed signals, and distributed files.

CHORUS–V2 provided an opportunity to reconsider the whole UNIX kernel architecture with respect to the following two concepts:

- Modularity: all UNIX services were split into several independent actors. This division required the design of specific interfaces between UNIX services that had previously relied upon informally sharing global kernel data structures in order to communicate.
- Distribution: objects such as files and processes, managed by system actors, could be distributed within CHORUS, as could be services such as `fork` or `exec`. This required the development of new protocols for distributed services, such as naming and object location.

This work provided invaluable experience when it came to development of CHORUS–V3; CHORUS–V2 may be considered as a draft of the current version.

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

2.5 CHORUS–V3 (1987–present)

The objective of this current version is to provide an industrial product integrating all positive aspects of the previous versions of CHORUS as well as those of other systems. CHORUS also provides several new significant features. CHORUS–V3 is described in the rest of this paper.

2.6 Appraisal of Four CHORUS Versions

The first lesson that can be extracted from the CHORUS work is that successively redesigning and implementing the same basic concepts provides an exceptional opportunity for refining, maturing and validating initial intuitions.

Technically, the successive redesigns had the following effects:

- The basic modular structure of kernel and system actors never really changed and some basic concepts persisted in all versions: ports, port groups, and messages.
- The style of communication (IPC) evolved with each version. The protocols, which were purely asynchronous in the beginning, were augmented with synchronous operations and eventually with synchronous RPC.
- Experiments were performed with naming and protection of ports, including variations of local names, global names and protection identifiers.
- Actors evolved from a purely sequential automaton with processing steps to a real-time multi-threaded virtual machine.
- Protection and fault tolerance are difficult problems and have remained open questions.
- Early versions of CHORUS handled fixed memory spaces, with the possibility of using memory management units for relocation. This evolved into dynamic virtual memory systems with demand paging, mapped into distributed and sharable segments.
- Finally, CHORUS was reimplemented in C++. Although the original implementation language, Pascal, did not cause any major problems, C++ was chosen because of the wide acceptance C has gained within the computing industry. C++ also provides modern object–oriented facilities, such as classes and inheritance, which have been quite useful for system development.

A summary of the publications describing the CHORUS system through its evolution 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 **Nucleus** and a set of system **servers**, which cooperate in the context of **subsystems** (Figure 1).

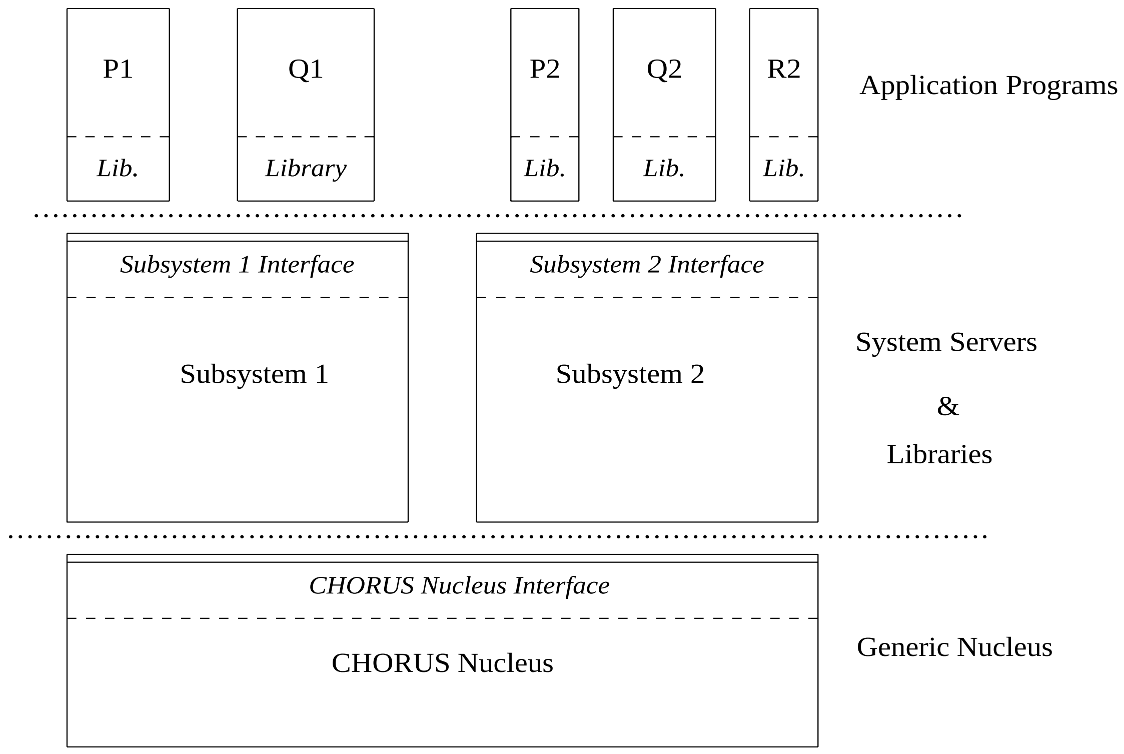

Figure 1. – The CHORUS Architecture

This overall organization provides the basis for an open operating system. It can be mapped onto a centralized as well as a distributed configuration. At this level, distribution is hidden.

The choice was made to build a two-level logical structure, with a generic Nucleus at the lowest level and almost autonomous subsystems providing applications with traditional operating system services.

Therefore, the CHORUS Nucleus is not the core of a specific operating system, rather it provides generic tools designed to support a variety of host subsystems, which can co-exist on top of the Nucleus.

This structure supports application programs, which already run on an existing operating system, by reproducing the operating system's interfaces within a subsystem. An example of this approach is given using a UNIX emulation environment called CHORUS/MiX.

The classic idea of separating the functions of an operating system into groups of services provided by autonomous servers is central to the CHORUS philosophy. In monolithic systems, these functions are usually part of the ‘‘kernel’’. This separation of functions increases modularity, and therefore the portability of the overall system.

3.1.1.1 The CHORUS Nucleus

The CHORUS Nucleus manages, at the lowest level, the local physical resources of a **site**. At the highest level, it provides a location transparent **inter-process communication (IPC)** mechanism. The Nucleus is composed of four major components providing local and global services (Figure 2):

- the **CHORUS supervisor** dispatches interrupts, traps, and exceptions delivered by the hardware;
- the **CHORUS real-time executive** controls the allocation of processors and provides fine-grained synchronization and priority-based preemptive scheduling;
- the **CHORUS virtual memory manager** is responsible for manipulating the virtual memory hardware and local memory resources;
- the **CHORUS inter-process communication manager** provides asynchronous message exchange and **remote procedure call (RPC)** facilities in a location independent fashion.

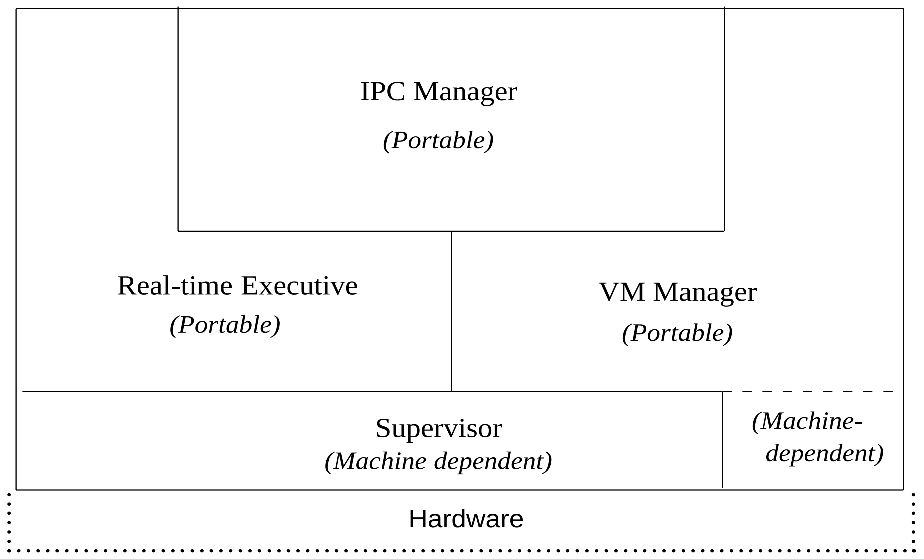

Figure 2. – The CHORUS Nucleus

There are no interdependencies among the four components of the CHORUS Nucleus. As a result, the distribution of services provided by the Nucleus is almost hidden. Local services deal with local resources and can be mostly managed using only local information. Global services involve cooperation between Nuclei to provide distribution.

In CHORUS–V3 it was decided, based on experience with CHORUS-V2 efficiency, to include in the Nucleus some functions that could have been provided by system servers: **actor** and **port** management, name management, and RPC management.

The standard CHORUS IPC mechanism is the primary means used to communicate with managers in a CHORUS system. For example, the virtual memory manager uses CHORUS IPC to request remote data to service a page fault.

The Nucleus was also designed to be highly portable, which, in some instances, may preclude the use of some underlying hardware features. Experience gained from porting the Nucleus to half a dozen different Memory Management Units (MMU's) on three chip sets has validated this choice.

3.1.1.2 The Subsystems

System servers work cooperatively to provide a coherent operating system interface, referred to as a **subsystem**.

3.1.1.3 System Interfaces

A CHORUS system provides different levels of interface (Figure 1).

- The Nucleus interface provides direct access to the low-level services of the CHORUS Nucleus.
- A subsystem interface is implemented by a set of cooperating, trusted servers, and typically represents complex operating system abstractions. Several different subsystems may be resident on a CHORUS system simultaneously, providing a variety of operating system or high-level interfaces to different application procedures.
- User libraries, such as the ''C'' library, further enhance the CHORUS interface by providing commonly used programming facilities.

3.1.2 Basic Abstractions Implemented by the CHORUS Nucleus

The basic abstractions implemented and managed by the CHORUS Nucleus are given in the following table.

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

These abstractions (Figure 3) correspond to object classes that are private to the CHORUS Nucleus. Both the object representation and the operations on the objects are managed by the Nucleus. Three other abstractions, shown in the table below, are cooperatively managed both by the CHORUS Nucleus and subsystem servers.

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

Rationale: Each of the abstractions from the two previous tables plays the role specified below in the CHORUS 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. A thread is tied to exactly one actor and shares its actor's resources with all other threads of that actor.

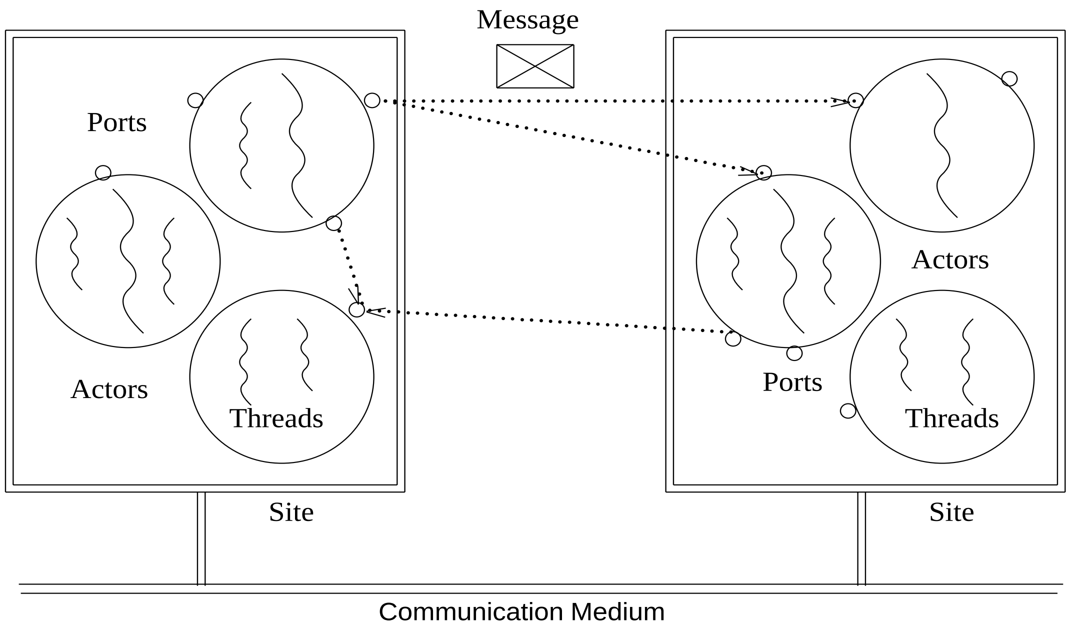

Figure 3. – CHORUS Main Abstractions

A **message** is a byte string which has been addressed to a port.

Upon creation, a port is attached to exactly one actor and allows the threads of that actor to receive messages sent to that port. Ports can migrate from one actor to another. Any thread having knowledge of 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 all have unique identifiers which are global, location independent, and unique in space and in time.

Segments are collections of data, managed by system servers which uniquely define their contents. The location of the data provided for a segment is unrelated to the location of the user of the segment.

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

- Resources can be identified within their servers by a server-dependent **key**. This key, when used in combination with the port UI of the server, forms a **capability** that can be used to securely reference a resource.
- Actors and ports receive **protection identifiers** with which the Nuclei stamp all the messages sent. Actors receiving messages can use these identifiers for authentication.

3.2 Naming Entities

3.2.1 Unique Identifiers

All CHORUS objects such as actors, ports, and segments (described in later sections) are referenced in a global fashion using unique identifiers (UIs).

Rationale: Given that CHORUS is a distributed system, it is important to have a consistent way to name objects within a collection of sites which represents a **CHORUS domain**. The standard structure used for a unique identifier allows distinct administrative entities to be easily interconnected.

The CHORUS Nucleus implements a **Unique Identifier Location Service (UILS**) allowing system entities to use unique identifiers to reference CHORUS objects without knowledge of their current location.

The CHORUS Nucleus guarantees that a UI will exist on at most one site and will never be reused. A UI is a 128 bit structure whose uniqueness is assured by classic construction methods. The location service uses several hints for finding the object represented by its UI. One such hint is the creation site which is embedded within the UI. A UI may be freely transmitted within the CHORUS domain using any desired mechanisms, such as shared memory, messages, or carrier pigeons.

> **Rationale:** The Nucleus does not control the propagation of unique identifiers; it is the responsibility of the individual subsystems. The construction methods used offer a cheap and basic level of protection that is suitable for many circumstances.

3.2.2 Local Identifiers

Local identifiers (LIs) are used within the context of a server to identify resources associated with that server. These identifiers are represented by integers and are generated by the local server. They may be transmitted between entities within the CHORUS domain, but only have meaning when referencing resources of the server that created them. In particular, these identifiers may represent an index into a server table or a pointer to a server data structure.

3.2.3 Capabilities

Some objects are not directly implemented by the CHORUS Nucleus, rather by external services. These objects are named via global names known as **capabilities**. A capability is composed of a unique identifier and a **key**. The UI names the server that manages the object. The key is a server-specific handle which identifies the object and the access rights associated with a request (Figure 4). The structure and semantics of a key are defined by its server.

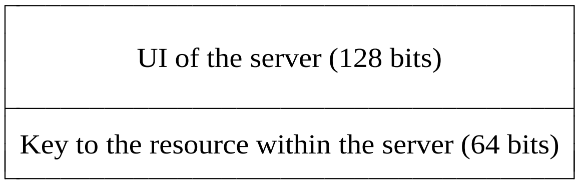
UI of the server (128 bits)
Key to the resource within the server (64 bits)

Figure 4. – Structure of a Capability

3.3 Active Entities

3.3.1 Sites

The physical support of a CHORUS system is composed of an ensemble of **sites**, interconnected by a communication medium.[2] A site is a grouping of tightly-coupled physical resources controlled by a single CHORUS Nucleus. These physical resources include one or more processors, central memory, and attached I/O devices.

2. Examples of suitable communication media include: Ethernet, token rings, hypercubes, and high-speed busses.

3.3.2 Actors

An **actor** is a collection of resources in a CHORUS system. An actor defines a protected address space supporting the execution of threads that share the resources 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 5).

CHORUS defines three types of actors: **user actors**, **system actors**, and **supervisor actors**. These definitions encapsulate the concepts of **trust** and **execution privilege**. An actor is trusted if the Nucleus recognizes its right to perform sensitive Nucleus operations. Execution privilege refers to the ability of an actor to execute privileged instructions and is typically controlled by a hardware status register. User actors are not trusted and unprivileged. System actors are trusted but unprivileged. Supervisor actors are both trusted and privileged. Currently, privileged actors execute within the shared, protected system address space and other actors execute within private user address spaces; this configuration is an implementation detail.

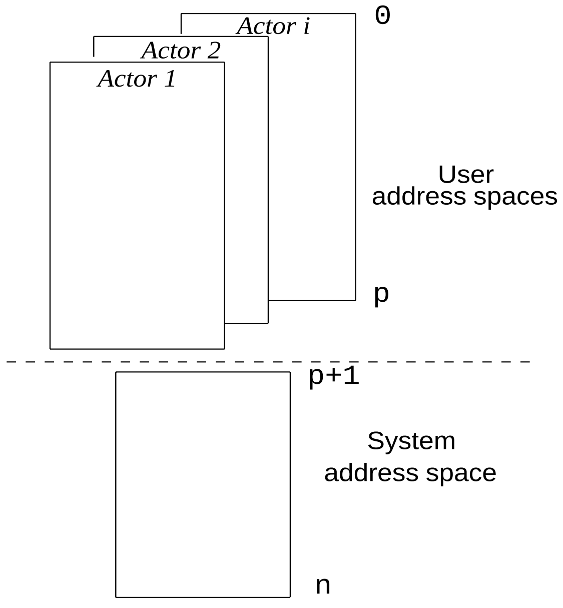

Figure 5. – Actor Address Spaces

A given site may support many actors simultaneously. Since each has its own user address space, actors define protected entities to the user.

Any given actor is tied to one site and its threads are always executed on that site. The physical memory used by the code and data of a thread is always that of the actor's site. Neither actors nor threads can migrate from one site to another.

> **Rationale:** Because each actor is tied to one site, the context of the actor is precisely defined; there is no uncertainty due to distribution because it depends only on the status of its supporting site. The state of an actor can thus be determined easily and context-dependent decisions made quickly. A site failure leads to the complete crash of its actors; no actors partially crash.

Actors are designated by capabilities built from the UI of the actor’s **default port** and a manipulation key. The knowledge of an actor’s capability gives full rights to that actor. By default, only the creator of an actor knows its capability, however the creator can transmit the capability to others.

The resources held by an actor are designated by local identifiers. The scope of these identifiers is limited to the specific actor holding the resource.

3.3.3 Threads

A **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 at any given point during the execution of the thread.

A thread is always tied to exactly one actor. This actor defines the thread’s execution environment. Within the actor, many threads can be created and can run concurrently. When a site supports multiple processors, the threads of an actor can be made to run in parallel on different processors.

Threads are scheduled as independent entities. The basic scheduling algorithm is a preemptive priority-based scheme, but the Nucleus also implements time slicing and priority degradation on a per-thread basis. This combination of strategies allows 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 mechanism (see § 3.4). However, as all threads of an actor share the same address space, communication and synchronization mechanisms based on shared memory can also be used. When the machine instruction set allows it, shared memory synchronization primitives can be constructed to avoid invoking the Nucleus.

Rationale: Why threads?

- Threads execute within an actor corresponding to one virtual address space and tied to one site. Thus, threads provide a mechanism for implementing multiple processes on machines, such as a Transputer, that do not support virtual memory.
- Threads provide a powerful tool for programming I/O drivers. Using threads to implement parallel constructs, such as interrupts and parallel I/O streams, simplifies driver programming.
- Threads allow servers to be multi-programmed, facilitating a concurrent implementation of the ‘‘client-server’’ model of programming.
- Threads allow the use of multiple processors within a single actor on a multi-processor site.
- Threads provide an inexpensive mechanism for achieving concurrency within multi-programs; the cost of a **thread context switch** is far less than the cost of an **actor context switch**.

3.4 Communication Entities

3.4.1 Overview

While threads of a single actor are able to communicate using shared-memory primitives, the basic communication mechanism applicable to the threads of any actor is the exchange of messages via message queues called ports.

Ports provide a communication entity through which threads, potentially within different actors on different sites, can synchronize and exchange information. They are named by unique identifiers thus making them location independent. Messages are records of information which are transferred between threads sending to and receiving on ports.

3.4.2 Messages

A **message** is a contiguous byte string which is logically copied from the sender's address space to the receiver's address space. Using a coupling between virtual memory management and IPC, large messages may be transferred efficiently using copy-on-write techniques, or if possible, by simply moving page descriptors.

Rationale: Why messages rather than shared memory?

- Messages make the exchange of information explicit.
- Messages provide well-defined points for isolating the state of an actor.
- In a heterogeneous environment messages are easier to manage than shared memory.
- Using messages, the grain of information exchange is bigger, is better defined, and its cost can be more accurately calculated.
- The performance of shared memory can be approximated through hints and local optimizations when sending messages locally (see § 5).
- Using current technology, message passing systems increase in scale more easily, and the message passing paradigm can be applied more efficiently in loosely-coupled environments.

3.4.3 Ports

Messages are not addressed directly to threads or actors, rather to intermediate entities called **ports**. The port abstraction provides the necessary decoupling of the interface of a service and its implementation. This decoupling provides the basis for dynamic reconfiguration (see § 3.6.1.)

A port represents both an address to which messages can be sent and an ordered collection of unconsumed messages. When created, a port is attached to a specified actor. Only the threads of this actor may receive messages on that port. A port can be attached to only one actor at a time, but multiple threads within that actor can receive messages on the port.

A port can migrate from one actor to another; this migration may also be applied to the unconsumed messages of the port.

Rationale: Why Ports?

- one actor may have several ports and therefore multiple incoming communication paths.
- multiple threads may share a single port, providing concurrent consumption of data on incoming communication paths.
- the port abstraction provides the basis for dynamic reconfiguration. The extra level of indirection provided between any two communicating threads allows a given service to be migrated transparently from one actor to another. (An example is provided in § 3.6.1).

When a port is created, the Nucleus returns both a local identifier and a unique identifier to name the port. The LI can be used only within the context of the actor to which the port is currently attached. Messages carry the UI of the port or port group to which they are sent.

Rationale: In the successive versions of CHORUS, the naming scheme of ports evolved:

- In CHORUS–V1, small UIs were adopted as the sole naming space. This proved to be 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 local identifiers, modeled on file descriptors, thus insuring protection. Port inheritance on `fork` and `exec` was implemented. This scheme was found to have two main drawbacks: port inheritance was hard to understand and, more importantly, port name transmission required extra mechanisms.
- The scheme adopted in CHORUS–V3 combines the advantages of both previous versions. In CHORUS–V3:

- Ports are named by global names at user level; therefore name transmission in messages is simplified.
- Within an actor, ports attached to it are named in system calls by local identifiers. This naming simplifies the user interface and provides a simple and efficient protection mechanism.

3.4.4 Port Groups

Ports can be assembled into **port groups** (see Figure 6). The port group abstraction extends message-passing semantics between threads by allowing messages to be directed to an entire group of threads and by allowing providers of a service to be selected from among members of a port group.

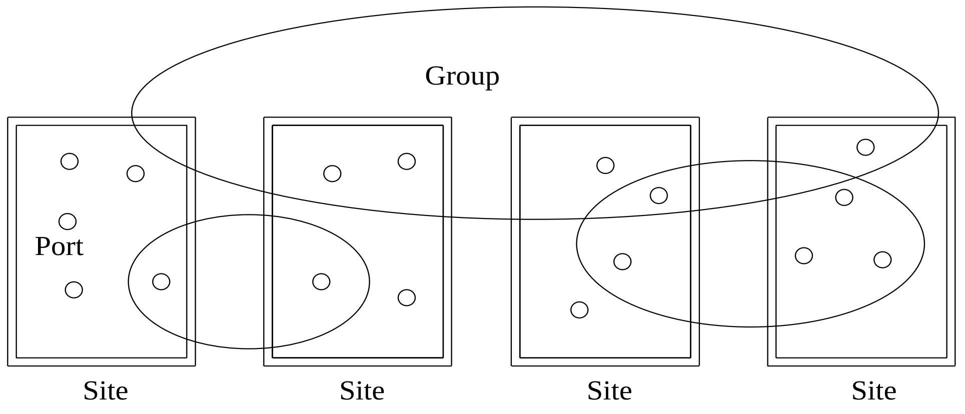

Figure 6. – Port Groups

Port groups are similar to ports in that they are globally named by UIs. When a port group is created it is initially empty; ports can be subsequently added and deleted. Furthermore, port group UIs can be statically allocated and maintained over an extended period. Statically allocated port groups are used by subsystems to provide well-known service names. A port can be a member of several groups.

> **Rationale:** This capability can be used to dynamically bind system servers with their global names. At system compilation time, a subsystem statically defines the names of port groups on which clients will request services. At boot time, subsystems create ports on which they will accept and answer service requests. The subsystems insert these ports into the statically defined port groups. Thus, clients are allowed to be written using these fixed port group names. This extra level of abstraction separates CHORUS service names from the implementors and consumers of that service.

Operations performed on groups, such as the insertion and deletion of ports, must be controlled by the Nucleus to ensure security. The Nucleus provides the creator of a group with its **group manipulation key**. This key must be specified to modify the contents of the group and may be freely transmitted between consenting actors.

The group UI and the group manipulation key are related as follows:[3]

3. The group UI and the group manipulation key combine to form a capability.

$$groupUI = f(key)$$

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

3.4.5 Communication Semantics

The CHORUS **inter-process communication (IPC)** mechanism permits threads to communicate by sending *asynchronous* messages or by placing remote procedure calls (RPC). When sending an asynchronous message, the emitter is blocked only during the local processing of the message. The system does not guarantee that the message will be received by the destination port. When the destination port is not valid, the sender is not notified and the message is discarded.

By contrast, the **RPC** protocol permits the construction of services using the ‘‘client-server’’ model. RPC guarantees that the response received by a client is that of the server and corresponds to the client’s request. RPC also permits a client to know if its request has been received by the server, if the server has crashed before emitting a response, or if the communication path has broken.

> **Rationale:** Asynchronous IPC and synchronous RPC are the only communication services provided by the CHORUS Nucleus. The asynchronous IPC service is basic enough to allow building more sophisticated protocols within subsystems. It reduces network traffic in the successful cases, yielding higher performance and better scaling to large or busy networks. RPC is a simple concept, easy to understand, present in language constructs, and easy to handle in case of errors or crashes. The Nucleus does not provide ‘‘flow control’’ protocols since application requirements vary greatly.

When messages are sent to port groups, several addressing modes are provided:
- broadcast to all ports in the group[4],
- send to any one port in the group;
- send to one port in the group, located on a given site;
- send to one port in the group, located on the same site as a given UI.

3.5 Virtual Memory Management

3.5.1 Segments

The unit of information exchanged between the virtual memory system and data providers is the **segment**. Segments are global and are identified by capabilities. They are generally implemented as secondary storage constructs, such as files or swapping areas. Segments are managed by system actors called mappers. The representation of a segment, its capabilities, access policies, protection and consistency are defined and maintained by these servers.

A segment may be accessed by mapping a portion of it into a region (see § 3.5.2) or by explicitly calling the CHORUS `sgRead` or `sgWrite` system calls. **Mappers** provide a single basic read/write interface, based on CHORUS IPC, which is used regardless of whether the request originates from the Nucleus as a result of a page fault in a mapped region or is the result of an explicit CHORUS system call.

> **Rationale:** On-demand page loading techniques were chosen in order to make it possible to access very large segments. Another approach, based on whole-segment loading can be found in[Tane86]. This technique, however, assumes that segments are relatively small and requires large amounts of physical memory.

4. Broadcast mode is not currently applicable to RPC requests

3.5.2 Regions

The actor address space is divided into **regions**. A region of an actor contains a portion of a segment mapped to a given virtual address with a set of access rights (Figure 7). Every reference to an address within a region behaves as a reference to the mapped segment, controlled by the associated access rights. A thread can create, destroy, and change the access rights of all regions of actors for which it holds capabilities.

Regions within the site-wide system address space can be manipulated only by **trusted threads**. Threads that read, write, and execute regions within the system address space avoid the overhead of an **address space context switch**. The CHORUS system uses this functionality to remap IPC requests between subsystem actors executing on the same site. Remapping IPC requests in this way also avoids the overhead of copying the message. An address space context switch is also avoided when invoking interrupt handlers that execute within the system address space.

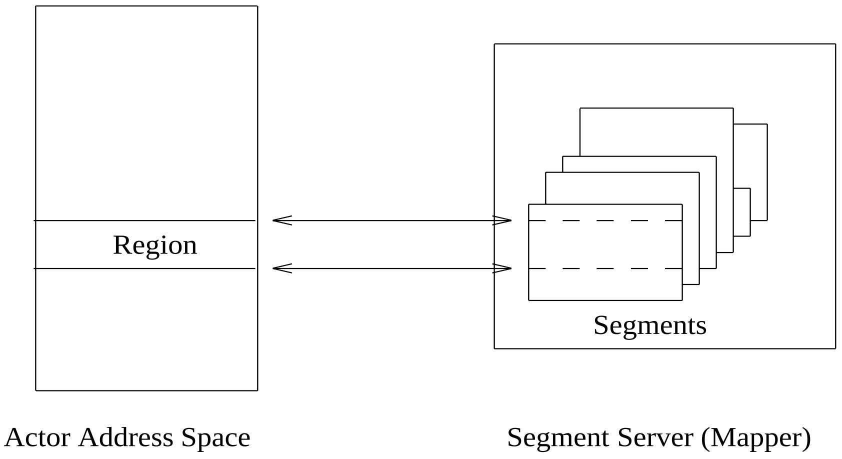

Figure 7. – Regions and Segments

3.5.3 Segment Representation Within the Nucleus

The Nucleus manages a per-segment local cache of physical pages. This cache contains pages obtained from mappers which will be used to fulfill future requests for the same segment data, whether they originate from explicit `sgRead` calls or as a result of page faults. Write updates are made to the local cache and are passed back to the mapper when physical pages are needed by the Nucleus or if the cache is explicitly invalidated (see Figure 8). The use of the local cache for both functions optimizes the use of physical memory within a site and reduces network traffic.

For a given site, the consistency of a segment shared among regions in different actors is guaranteed because they share the same cache in physical memory. When a segment is shared among actors executing on different sites, there is one local cache per site and mappers are required to maintain the consistency of these caches (Figure 9). Algorithms for dealing with the problems of coherency of shared memory are proposed in[Li86] .

A standard Nucleus-to-mapper protocol, which is based on CHORUS IPC, has been defined for managing local caches. The protocol provides mechanisms for demand paging, flushing pages for swapping out and maintaining cache consistency, and annihilating a local cache.

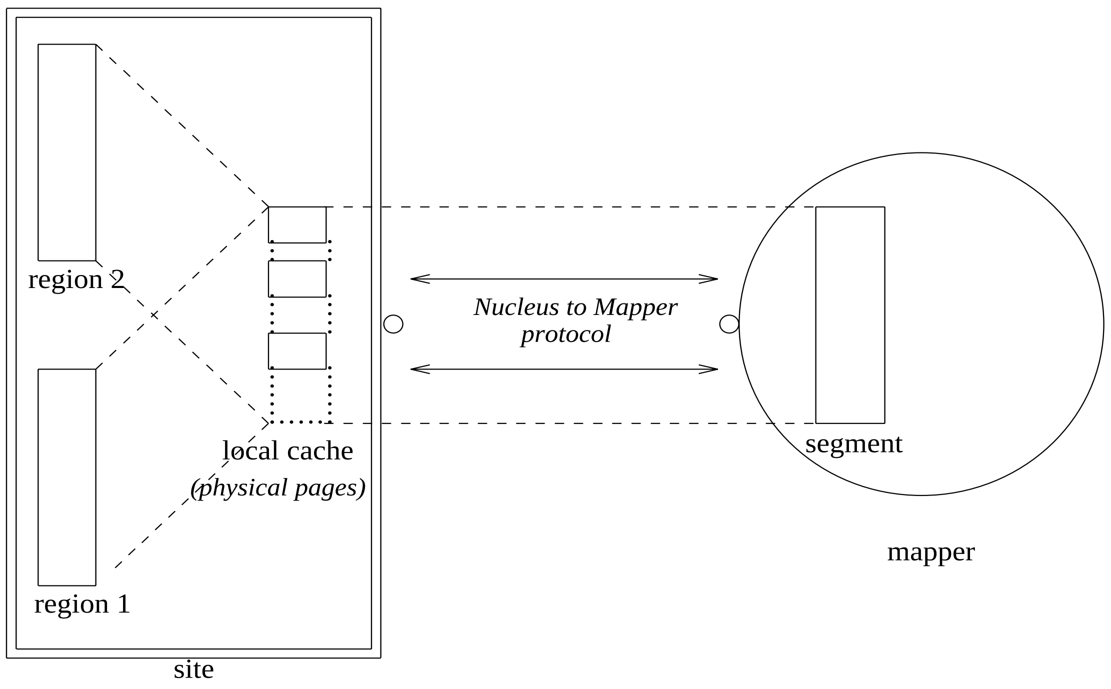

Figure 8. – Local Cache

3.5.4 Deferred Copy Techniques

Deferred copy techniques are a mechanism whereby the Nucleus uses memory management facilities to avoid performing unnecessary copy operations. The CHORUS Nucleus implements the duplication of data using a copy-on-write history technique similar to the shadow-object technique of Mach[Rash87] for large objects. It uses a per-virtual page copy duplication of data similar to [Ging87, Mora88] for small objects.

These techniques vastly improve performance when copying large amounts of data from one actor to another – as is done during a UNIX `fork` operation – and when moving small amounts of data between segments for IPC and I/O operations.

3.6 Communication Support

A variety of communication support facilities is provided by the CHORUS system.

- The CHORUS Nucleus provides a framework within which reliable services may be implemented. The Nucleus allows these services to be dynamically reconfigured to alter server behavior or to recover from server failure.
- The CHORUS Nucleus provides facilities, as an integral part of the CHORUS IPC, for constructing authentication protocols.
- The CHORUS Nucleus handles message passing between actors executing on the same site. The Nucleus cooperates with the **Network Manager** to provide a location transparent facility for exchanging messages with actors executing on different sites.

3.6.1 Reconfiguring a Service

Ports and port groups provide a level of indirection for information sent between communicating threads. By taking advantage of this indirection, the implementation of a service provided by a server may be reconfigured without disturbing the clients of that service.

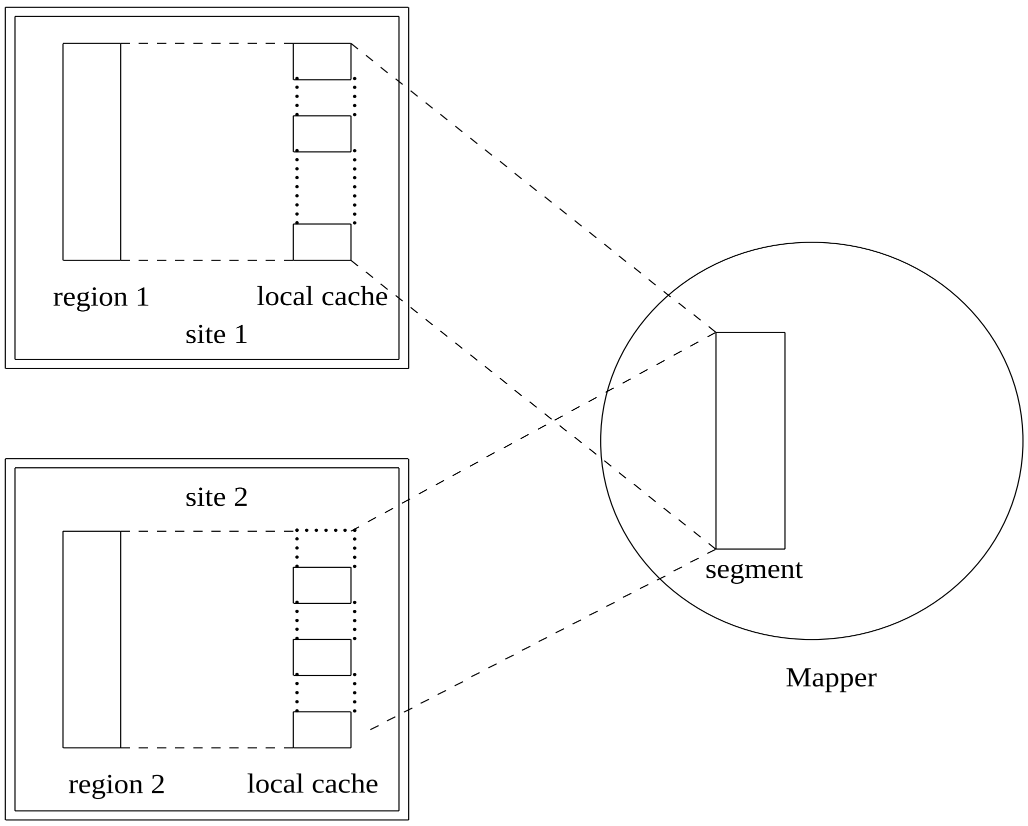

Figure 9. – Distributed Local Caches

3.6.1.1 Reconfiguration Using Port Migration

Figure 10 illustrates the migration of a port from `Server 1` to `Server 2` without interruption of the provided service. Migration allows, for example, a new version of a service to be installed and activated. When a port is migrated, any unconsumed messages associtated with that port can be made to migrate with it. Thus, clients of a service can continue to operate undisturbed during migration; communication is not interrupted. Port migration requires, however, that the servers involved participate actively. In particular if a server crashes, its port will not be migrated.

3.6.1.2 Reconfiguration Using Port Groups

The port group abstraction, on the other hand, offers a framework for passively reconfiguring a service. Messages that are directed towards port groups will be automatically rerouted to a port that is capable of providing the desired service if a member port becomes invalid. This form of reconfiguration may require minimal participation of clients or providers of a service to allow them to recover from temporary disruptions in communication.

Figure 11 illustrates the automatic reconfiguration of a service using port groups. The `client` directs requests to port group `G` whose members include ports `P1` and `P2`. When `Server 1` crashes, requests are rerouted to `Server 2` which continues to provide the service.

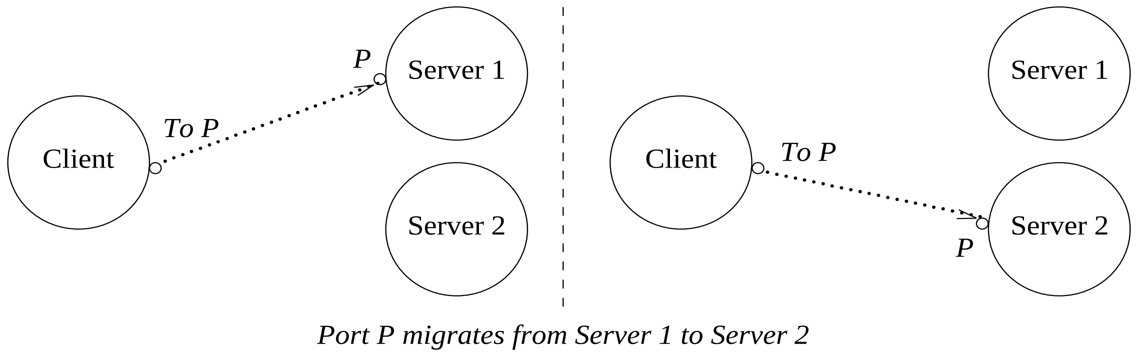

Figure 10. – Reconfiguration Using Port Migration

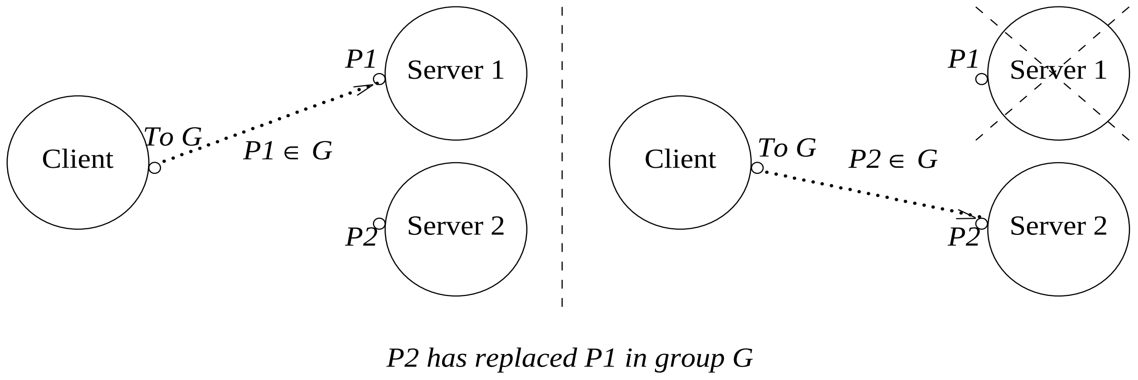

Figure 11. – Reconfiguration Using Groups

3.6.2 Authentication

The CHORUS Nucleus provides the ability to protect objects managed by subsystem servers. Since all services are invoked via the CHORUS IPC mechanism, the support provided is integral to this mechanism.

The Nucleus supports a **protection identifier** for each actor and port. The structure of these identifiers is fixed, but the Nucleus does not associate any semantic value with them. Protection identifiers can be altered only by trusted threads. Upon creation, an actor or port inherits the protection identifier of the actor that created it.

Each message sent by an actor 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 and can be used to securely determine the identity of the requester of a service.

3.6.3 The Network Manager

The **Network Manager** provides high-level CHORUS communication facilities. The Network Manager is constructed from three independent modules, known as the High Interface, the Communication Core and the Low Interface. The High Interface implements CHORUS remote IPC, RPC, and error handling protocols. The Communication Core implements standard communication protocols and services such as the OSI protocols. The Low Interface manages requests to the **Network Device Manager**, which implements network device drivers.

3.6.3.1 Remote IPC and RPC

To implement remote IPC and RPC, the Network Manager uses two protocols. The first provides CHORUS-specific features such as locating distant ports, and remote host failure handling. The second protocol is responsible for data transmission between sites and is operating system independent.

To provide a data transmission facility, the Network Manager follows current international standards. The Network Manager implements the OSI family of protocols through the transport level to support network-wide IPCs and RPCs.

> **Rationale:** The choice to use the OSI protocols results from the desire to use existing standards whenever possible. Such a choice, however, can be complemented or changed according to the specific requirements of the supporting network or applications. The IPC and RPC mechanisms can use any protocol implemented in the Network Manager Communication Core as long as this protocol provides reliability and data ordering.

The Network Manager High Interface also implements an error handling protocol. In the case of site failure, this protocol is used to notify the initiator of an RPC that its request cannot be satisfied.

3.6.3.2 UI Location Service

During their lifetime, ports can be attached, in succession, to different actors. The Network Manager, cooperating with the Nucleus, is responsible for hiding the location of the destination ports used in IPC or RPC requests. The UI Location Service is used by the Network Manager to find the site on which a non-local destination port resides.

To locate non-local destination ports, the Network Manager maintains a cache of known ports and groups. When a remote port becomes invalid, due to port migration, crash, or movement out of the CHORUS domain, the cache entry is invalidated. Upon receipt of a negative acknowledgement, the local Network Manager enters a search phase and broadcasts queries to remote Network Managers. Ports not held in the cache are first assumed to reside on their original creation site.

3.7 Hardware Events and Exception Handling

The CHORUS Nucleus provides simple facilities which allow user actors to implement their own exception handling schemes. In addition, CHORUS supports high-performance event and exception handling for subsystems that wish to directly support hardware devices as well as real-time applications.

3.7.1 Basic Exception Handling

In its simplest form, the CHORUS Nucleus provides mechanisms that allow user or system threads to take control of exceptional events within an actor. A thread may ask to associate a port with exceptions occurring within an actor. When the Nucleus detects an exception and the actor has a port assigned to the task, it sends a message to that port, describing the exceptional condition. Threads listening on this port can then take whatever action is appropriate for the given exception.

If no port has been assigned, exceptions cause the death of the thread generating the exception.

3.7.2 Supervisor Actor Exception and Device Handling

For supervisor actors, CHORUS provides additional facilities whereby system programmers can directly access low-level I/O events and hardware exceptions to meet their needs. The supervisor supports a mechanism for assigning device interrupts from the hardware to user threads.

When an interrupt occurs, the supervisor saves the context of the interrupted thread and sequentially calls a prioritized sequence of user routines associated with the given interrupt. Any of the individual routines may initiate a break in the sequence, if necessary. After the last routine has finished, the supervisor may initiate rescheduling.

The supervisor supports a similar facility, through which actor routines may be associated with hardware traps or exceptions. This facility allows subsystem managers, such as the CHORUS/MiX **Process Manager**, to provide efficient and protected subsystem interfaces.

The routines associated with hardware events are executed by supervisor threads. The association between a hardware event and its handler may be reconfigured at any time. If no association between a hardware exception and a handler is made, the exception is handled as it would be for a standard actor.

3.8 Subsystems

Sets of CHORUS actors that work together to export a unified application programming interface are known as **subsystems**. Subsystems, such as the CHORUS/MiX subsystem, export high-level operating system abstractions, such as process objects, protection models, and data providing objects. They construct these high-level abstractions using the primitives exported by the CHORUS Nucleus.

Subsystems can securely export their abstractions by providing access to them through system traps. Code and data to implement these abstractions may be loaded into system space to improve performance. Subsystem actors communicate with one another by means of CHORUS IPC.

The level of protection required for subsystem servers varies widely from service to service. The CHORUS Nucleus provides basic protection facilities with which subsystems may enforce their own levels of security and protection as appropriate.

In the example illustrated in Figure 12, several user actors are shown accessing facilities in a CHORUS subsystem. The subsystem is protected by means of a system trap interface. A portion of the subsystem is implemented as a system actor, executing in system space, and a portion is implemented as user actors. The subsystem servers communicate using CHORUS IPC.

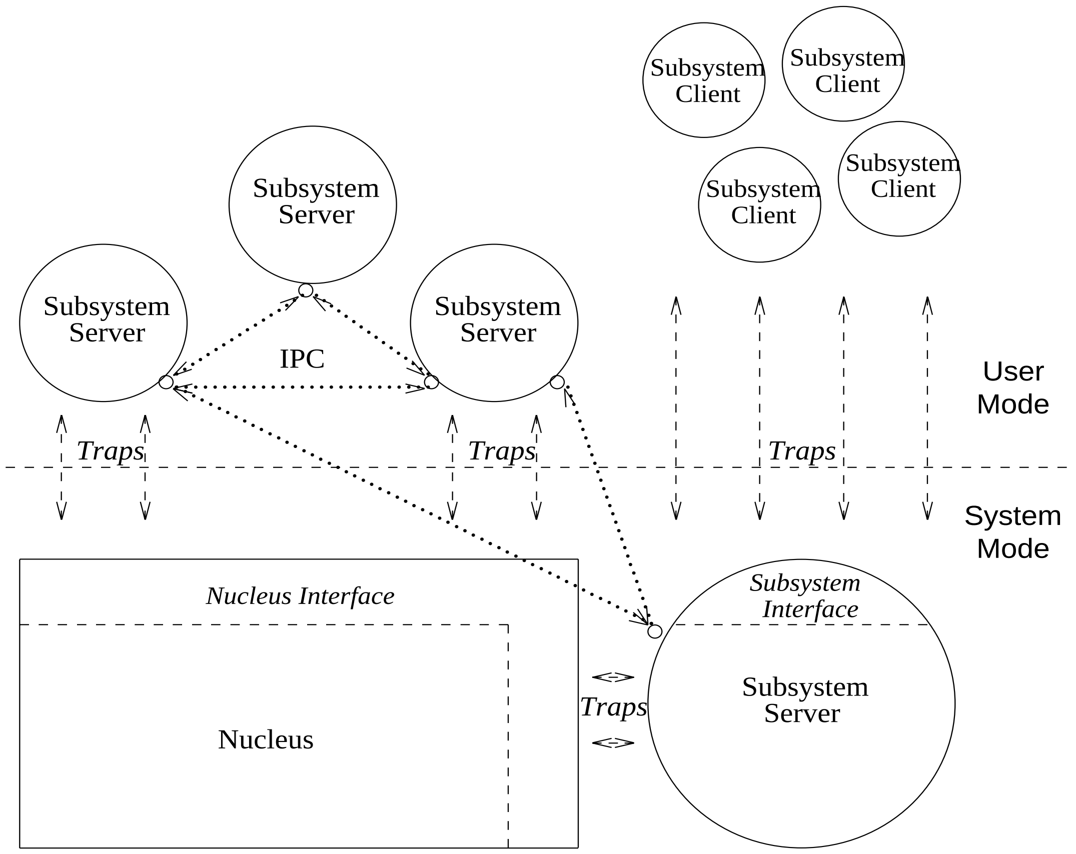

Figure 12. – Structure of a Subsystem

4. CHORUS/MiX: A UNIX SUBSYSTEM

4.1 Overview

The first subsystem implemented within the framework of the CHORUS architecture was a UNIX SYSTEM V subsystem. In the remainder of this paper, we will refer to the combination of the CHORUS Nucleus and the set of UNIX SYSTEM V subsystem servers as the CHORUS/MiX™ operating system. The facilities provided by the CHORUS Nucleus have permitted the design of coherent extensions of UNIX for distributed computing.

The implementation of the abstractions of this extended UNIX interface are described in the following sections. Some of the abstractions are implemented by the CHORUS Nucleus and are provided by CHORUS Nucleus calls. Others are implemented in terms of CHORUS actors.

Some implementation choices are explained in more detail, emphasising problems which arise when introducing distributed processing into a UNIX system.

4.1.1 Objectives

The CHORUS technology, when applied to UNIX overcomes a number of widely recognized limitations of traditional UNIX implementations. It has been applied with the following general objectives:

- To implement UNIX services in a modular fashion, as a collection of servers. These autonomous servers have the property that they need only be resident on those sites that make use of them. In addition, these servers can be dynamically loaded or unloaded as required.
- To permit application developers to implement their own servers, such as window managers and fault-tolerant file managers, and to integrate them dynamically into CHORUS/MiX.
- To extend UNIX functionality with respect to real-time constraints, distribution of resources, and concurrent programming. The CHORUS real-time executive provides real-time facilities previously missing in UNIX systems. CHORUS/MiX provides distributed resource sharing within the CHORUS domain. The CHORUS thread facilities have been integrated into CHORUS/MiX, providing multi-threaded UNIX processes.
- To build UNIX functionality using the CHORUS primitives as a basis. The CHORUS/MiX subsystem can be viewed as a typical client of these primitives; its implementation does not affect the implementation of the CHORUS Nucleus.
- To provide compatibility with existing application programs and device drivers without a severe degradation in performance. Existing applications should run without modification or recompilation; existing device drivers should be integrated into the CHORUS/MiX system with minimal effort.

4.1.2 Extensions to UNIX Services

In addition to the standard UNIX functionality, CHORUS/MiX offers extensions to UNIX abstractions and operations. The UNIX operations have been extended to work in a distributed environment and the UNIX process model has been extended to provide process semantics that are consistent within multi-threaded programs.

4.1.2.1 Distribution

- The file system is fully distributed and file access is location independent. File trees are *automatically* interconnected to provide a name space within which all files, whether remote or local, are designated with no syntactic change from current UNIX.
- Operations on processes have been extended to remote sites. The UNIX `exec` system call can be made to execute on a distant 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 way which is compatible with existing applications. Most applications benefit from the extensions to file, process and signal management without modification or recompilation.

4.1.2.2 Multiprogramming a UNIX process

Each UNIX process is implemented as an actor; hence the multi-threaded Nucleus model extends naturally into the UNIX layer. In order to distinguish the thread interface provided by the Nucleus from the thread interface provided by CHORUS/MiX, CHORUS/MiX threads will be referred to as **u_threads** for the remainder of this paper.

The u_thread management interface has been defined to achieve two major objectives. U_threads are intended to provide a low-level, generic thread interface capable of supporting needs of typical concurrent programming clients. At the same time, the use of u_threads should have a minimal impact on the syntax and semantics of UNIX system calls so that UNIX system calls can easily be used within multi-threaded programs.

The u_thread interface includes primitives for creating and deleting u_threads, suspending and resuming their execution, and modifying their priorities. All of these are low-level services and incorporate a minimum of semantic assumptions. Policies regarding stack management and ancestor/descendant relationships among u_threads, for example, are left to higher-level library routines.

In several respects, traditional UNIX system services exploit the fact that memory, resource ownership, and signal delivery are tied to the unit of CPU scheduling. In adapting these services for multi-threaded processes, CHORUS has sought preserve the existing interfaces and functions to the greatest degree possible.

Most of the basic UNIX process management functions extend readily, however, some pose compatibility problems. For example, only the context of the u_thread executing the `fork` system call is duplicated in the child process; other threads are discarded. Similarly, the `exec` system call produces a new mono-threaded process, regardless of how many threads were executing prior to its invocation.

In general, multiple u_threads can execute CHORUS/MiX system calls concurrently. The CHORUS/MiX subsystem contains resources, however, such as shared data structures, which cannot be acquired for use by more than one u_thread at a time. The first u_thread that acquires such a resource within a system call will cause the other u_threads seeking that resource to block until the resource is made available, serializing their execution. Typically, resources are not held for periods of long duration; it is necessary that u_threads release resources before waiting for I/O to complete, for example. These resources are private to the CHORUS/MiX subsystem, so u_threads making CHORUS/MiX system calls will not interfere with threads executing user code or threads executing in other subsystems.

The introduction of multi-threaded processes mandates a reconsideration of signal handling. CHORUS/MiX adopts a new approach to signal delivery and management. The goal is simple: signals should be processed by the u_threads that have expressed interest in receiving them. Each u_thread has its own signal context consisting of components such as signal handlers, blocked signals, and a signal stack. System calls used to manage this information affect only the context of the calling u_thread. This design arose from considerations of common signal usage in existing mono-threaded programs and the desire to keep the same semantics in multi-threaded programs.

In order to determine which u_threads should receive signal delivery, CHORUS/MiX distinguishes two types of signals:

1. *Signals for which the identity of the target u_thread is not ambiguous*. These signals correspond to synchronous exceptions, timer and asynchronous I/O signals. In these cases the signal is sent only to the u_thread concerned; that is, the u_thread generating an exception or the u_thread that has initiated a timer or asynchronous I/O event. If that u_thread has not either declared a handler or chosen to ignore the signal, the default action is taken for the entire process.
2. *Signals that are logically sent to an entire process*. For some signal-delivery purposes a process is viewed as a single entity. Line-discipline related signals, such as the interrupt or quit signals, and signals generated by the `kill` system call are broadcast to all u_threads of the target process. Each u_thread that has declared a signal handler for the particular signal will receive delivery. If no u_thread either declares a handler or explicitly ignores the signal, the default action is again taken relative to the entire process.

Thus, signal semantics and usage extend smoothly from the traditional UNIX mechanisms to the multi-threaded environment of CHORUS/MiX. The per-u_thread signal status adds to the size of the state information that must be initialized and maintained for each u_thread. No extra expense is added to the u_thread context switch, however, so the real-time responsiveness of the system is not compromised.

CHORUS/MiX has added the u_threadKill system call, to provide the additional functionality of allowing u_threads within a process to send signals to one another.

4.1.2.3 Interprocess Communication and U_thread Synchronization

U_threads use the CHORUS Nucleus IPC functionality for communication and synchronization with each other.

4.1.2.4 Real-Time Facilities

CHORUS/MiX real-time facilities result directly from the services provided by the CHORUS Nucleus.

4.2 CHORUS/MiX Architecture

UNIX functionalities may logically be partitioned into several classes of services according to the different types of resources managed: processes, files, sockets, or devices. The design of the UNIX Subsystem provides a clean definition of the interactions between these different classes, resulting in a modular structure.

CHORUS/MiX has been implemented as a set of system servers, running on top of the CHORUS Nucleus. Each system resource is isolated and managed by a dedicated system server. Interactions between these servers are based on the CHORUS message passing model, enforcing clean interface definitions.

Several types of servers may be distinguished within a typical UNIX subsystem: **Process Managers (PM), File Managers (FM), Socket Managers (SM), Device Managers (DM), IPC Managers (IPCM), Key Manager (KM)** and User Defined Servers (Figure 13).

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

4.2.1 Structure of a UNIX Server

Each UNIX server is implemented as a CHORUS actor. Each has its own context and, thus, may be debugged using standard debuggers. Furthermore, UNIX subsystem servers may be developed as user or system actors.

CHORUS/MiX servers are normally multi-threaded. Each request to a server is processed by one of its threads, which manages the context of the request until the response has been issued.

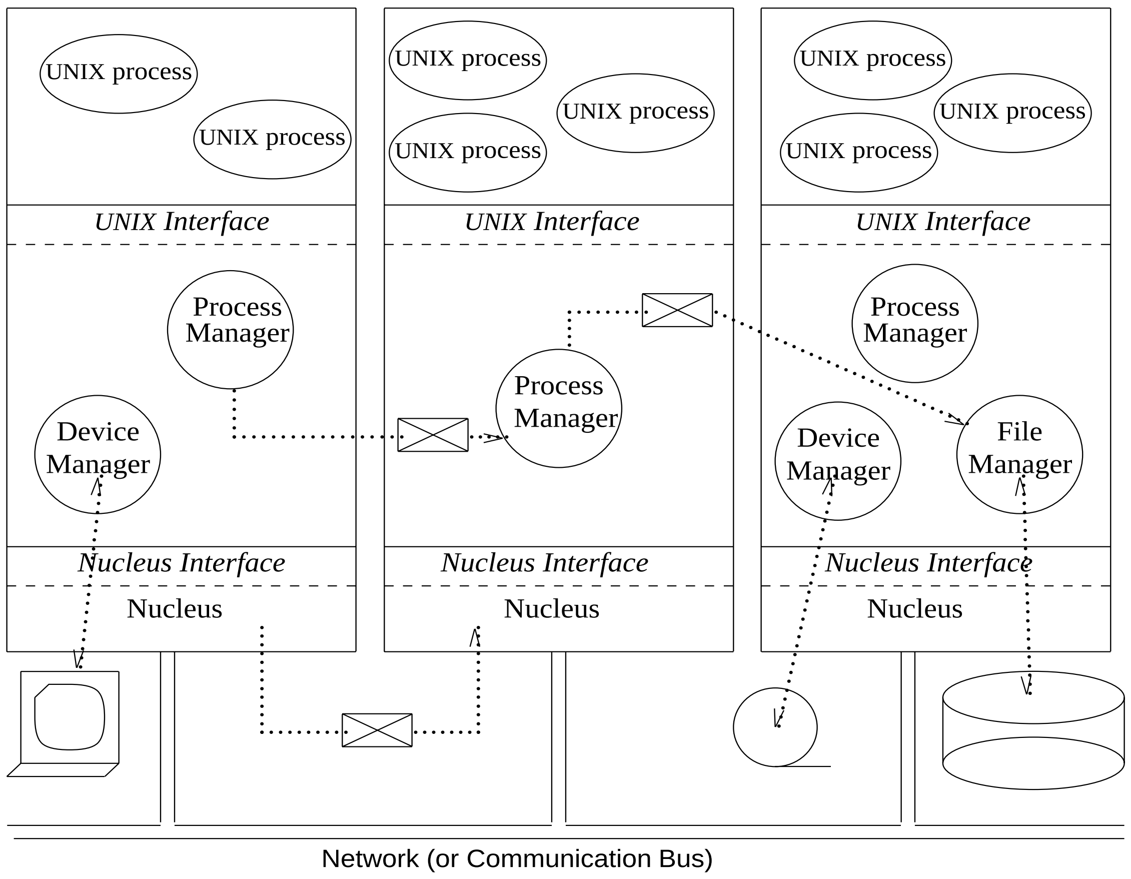

Figure 13. – CHORUS/MiX:UNIX with the CHORUS Technology

Each server creates one or more ports to which clients send requests. Some of these ports may be inserted into port groups with well-known names. Such port groups can be used to access a service independent of the provider of the service.

Servers may also provide a trap interface to services. These interfaces can be compatible with existing UNIX system interfaces, but do not extend transparently across the network. The Process Manager offers this interface to provide binary compatibility with the target UNIX system.[5]

In order to facilitate the porting of device drivers from a UNIX kernel into a CHORUS server, a UNIX kernel emulation library, which is linked with UNIX device driver code, has been developed. It provides such functions as `sleep`, `wakeup`, `splx`, `copyin`, and `copyout`. Interrupt handlers are one of the few parts of a traditional UNIX device driver that may need to be modified to adapt to the CHORUS environment.

5. The target system CHORUS/MiX is the COMPAQ386 running SCO UNIX.

4.2.2 Process Manager

The **Process Manager** maps UNIX process abstractions onto CHORUS abstractions. It implements all of the UNIX process semantics including creation, context consistency, inheritance, and signaling.

On each UNIX site, a Process Manager implements the entry points used by user processes to access UNIX services. Since traditional UNIX system calls are implemented using traps, CHORUS/MiX achieves binary compatibility with UNIX utilities by attaching PM routines to those traps through the CHORUS Nucleus (see § 3.7.2). The PM itself satisfies requests related to process and signal management, such as `fork`, `exec`, and `kill`. For other system calls, such as `open`, `close`, `read`, `write` and System V IPC related calls (messages, semaphores and shared memory), the PM invokes other CHORUS/MiX subsystem servers to handle the request.

For example, when a CHORUS/MiX process issues the `open` system call, a trap is generated and handled by the local Process Manager. The PM, using the context of the client process and running in system mode, uses CHORUS RPC mechanism to invoke the File Manager to perform the pathname analysis. If the indicated file is not managed by the File Manager, as is the case for device files, the request is automatically sent by the FM to the appropriate Device Manager.

Process Managers interact with their environment through clearly defined interfaces:

- Nucleus services are accessed through system calls;
- File Manager, Socket Manager, Device Manager , IPC manager and Key Manager services used for process creation and context inheritance are accessed by means of CHORUS RPC.

Process Managers cooperate to implement remote execution and remote signaling:

- Each Process Manager creates a dedicated port, referred to as the **request port**, to receive remote requests. Such requests are processed by Process Manager threads.
- The request port of each Process Manager, in a given CHORUS domain, is inserted into the Process Manager static port group. Any Process Manager of any given site may, thus, be reached using one unique functional address.
- Process operations that do not apply to the local site are sent to the functional address of the appropriate Process Manager.

4.2.3 File Manager

Each site that supports a disk requires a **File Manager**; diskless stations make requests of remote File Managers. The two main functions of File Managers are to provide disk-level UNIX file system management and to act as mappers to the CHORUS Nucleus.

As file servers, File Managers process traditional UNIX requests transmitted via IPC. File Managers also provide some new services needed for process management, such as handling the sharing of open files between a process and its child. In addition, they perform the mapping between the UNIX name and the CHORUS text and data segment capabilities of an executable file. These two services represent the only interactions between process and file management.

As external mappers, File Managers implement services required by CHORUS virtual memory management, such as managing a backing store. They use the standard mapper interface services provided by the CHORUS Nucleus to maintain cache consistency when local virtual memory caches are accessed from remote sites.

CHORUS/MiX file system caches are also implemented using the CHORUS virtual memory mechanisms. Use of the interface optimizes physical memory allocation and makes the implementation of CHORUS/MiX system calls such as `read` and `write` network transparent.

The naming facilities provided by the UNIX file system have been extended to permit the designation of services accessed via ports. A new UNIX file type called a **symbolic port** can be inserted into the UNIX file tree. Symbolic ports associate a file name with the UI of a port. When such a file is encountered during pathname analysis, the corresponding request is sent to associated port (see § 4.4.1.). Figure 14 illustrates the interconnection of two machines, `pipo` and `piano`, using symbolic ports. In the example, the pathname `/fs/piano/usr/fred` refers to the same directory from both machines because `/fs/piano` is a symbolic port to the File Manager of the `piano` root file system.

CHORUS/MiX also implements **symbolic links** à la BSD. When combined with symbolic ports, this functionality provides an extremely powerful and flexible **network file system**.

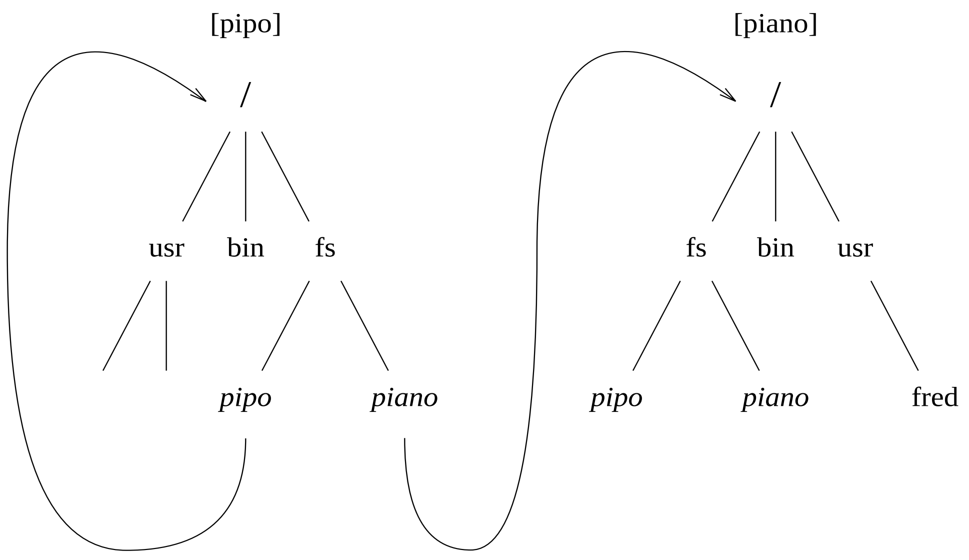

Figure 14. – File Trees Interconnection

4.2.4 Socket Manager

The **Socket Manager** implements UNIX 4.3 BSD socket management for the Internet address family. Socket Managers are only loaded onto sites that have network access.

4.2.5 Device Manager

Devices such as tty's and pseudo-tty's, bitmaps, tapes, and network interfaces are managed by **Device Managers**. There may be several Device Managers per site, depending upon site device requirements, which may be dynamically loaded or unloaded. Software configurations can be adjusted to suit the local hardware configuration or the needs of the user community.

A CHORUS IPC-based facility is used to replace the `cdevsw` table found in traditional UNIX systems. During initialization, the Device Manager sends its port and the major numbers of devices that it manages to the File Manager. When these major numbers are encountered during pathname analysis for an `open` system call, the request is sent to the associated port.

4.2.6 IPC Manager

The CHORUS/MiX IPC Manager (IPCM) provides user services equivalent to those supplied by a UNIX System V.3.2 kernel on inter process communication (IPC). These services include: messages, semaphores and shared memory. The IPCM was built from UNIX System V.3.2 IPC code.

The IPCM interacts with the PM, the KM (Key Manager, see below) and the FM. The PM is the main IPCM's "client": it transfers user system call requests and arguments to the IPCM, which handles them and returns values according to UNIX IPC semantics. Note that since they do not have any device connection, IPCM's may be present on any CHORUS/MiX site.

For shared memory operation, when a request is made to create a new shared segment, the IPCM requests service from the Object Manager to initialize a new segment capability. This capability will be used by processes trying to attach the shared segment to their address space, using the *rgnMap*() system call.

4.2.7 Key Manager

The CHORUS/MiX Key Manager (KM) is an internal CHORUS/MiX server. This means it does not provide any service to user programs, but accepts requests from the PM and the IPCM, in the context of the UNIX System V.3.2 IPC services.

The Key Manager creates and maintains a mapping between user provided *keys* and CHORUS/MiX internal *descriptors*. It ensures the uniqueness and coherence of the mapping accross a distributed system. There must be only one KM on a CHORUS/MiX system.

4.2.8 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 that it provides a powerful and convenient platform for experimentation with system servers. For example, new file management strategies or fault-tolerant servers can be developed and tested as a user level utility without disturbing a running system.

4.3 Structure of a UNIX Process

A traditional UNIX process can be viewed as a single thread of control executing within one address space. Each UNIX process is, therefore, mapped onto a single CHORUS actor whose UNIX system context is managed by the Process Manager. The actor's address space comprises memory regions for text, data, and execution stacks.

In addition, the Process Manager attaches a **control port** to each UNIX process actor. This control port is not visible to the user of that process. A control thread in the Process Manager is dedicated to receive and proceed all the requests on this port. This control thread executing within process contexts has two main properties:

- It shares the process address space and can easily access and modify the memory of the process in order to perform text, data, and stack manipulations during signal delivery or during debugging.
- It is ready to handle asynchronous events, such as signals, which are received by the process. These events are implemented as CHORUS messages received on the control port (Figure 15).

Allowing for multi-threaded processes has impacted the implementation of each process system context. The UNIX system context attached to a process has been split into two system contexts: a process context (`Proc`) (Table 1) and a u_thread context (`UThread`) (Table 2).

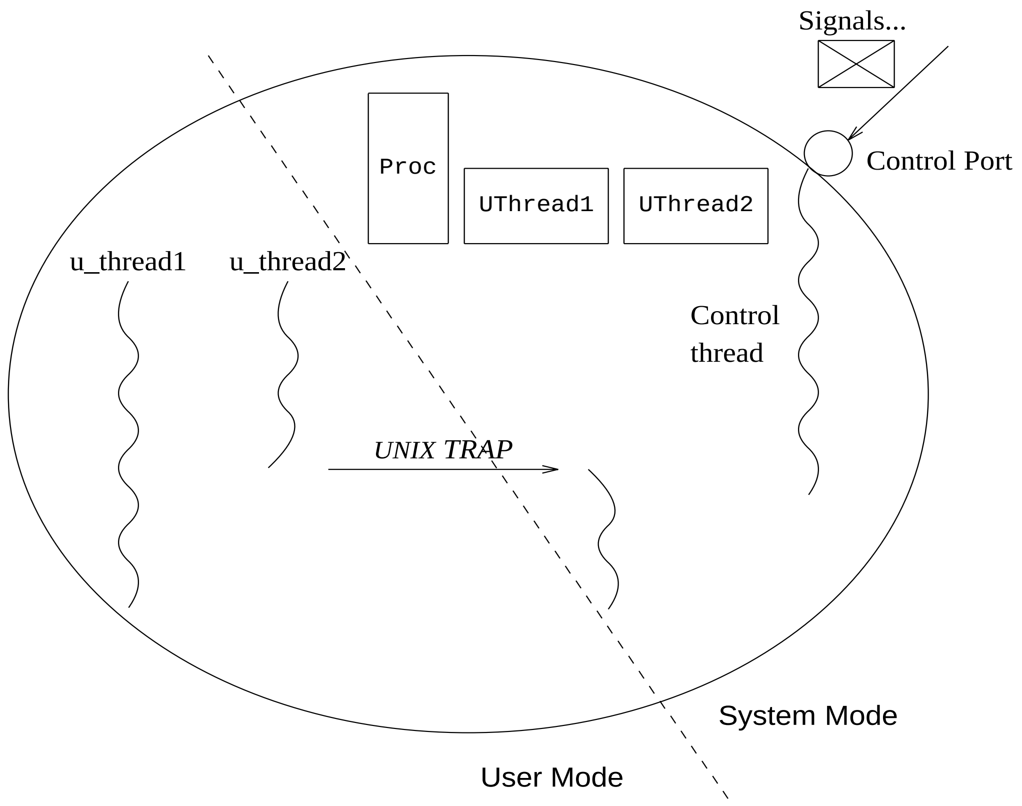

Figure 15. – UNIX Process as a CHORUS Actor

Table 1. – Process Context

Proc Context	
Actor implementing the Process	*actor name, actor priority, ...*
Unique Identifiers (UI)	*PID, PGRP, PPID, ...*
Protection Identifiers	*real user id, effective user id, ...*
Ports	*control port, parent control port, ...*
Memory Context	*text, data, stack, ...*
Child Context	*SIGCLD handler, creation site, ...*
File Context	*root and current directory, open files, ...*
Time Context	*user time, child time, ...*
Control Context	*debugger port, control thread descriptor, ...*
UThreads	*list of process' UThread contexts,*
Semaphore	*for concurrent access to* `Proc` *Context.*
IPC context	*UNIX messages, semaphores, shared memory.*

Table 2. – UThread Context

U-thread Context	
Thread implementing the u_thread	*thread descriptor, priority, ...*
Owner Process	*owner process proc descriptor*
Signal Context	*signal handlers, ...*
System Call Context	*system call arguments, ...*
Machine execution Context	

The two system contexts, `Proc` and `U_thread`, are maintained by the Process Manager of the current process execution site. These contexts are accessed uniquely by the Process Manager.

4.3.1 Process Identifiers

Each process is uniquely designated by a 32 bit global PID which results from the concatenation of two 16 bit integers. These integers consist of the creation-site id and a traditional UNIX process id.

4.3.2 Process Execution Site

As an extension to the standard UNIX process semantics, CHORUS/MiX maintains a notion of the child process creation site for each process. This site identifies the target site to which the `exec` operation is applied. By default, the child process creation site is set to the site on which the process currently resides.

4.3.3 Process Control

The Process Manager attaches a **control port**[6] to each CHORUS/MiX process. A dedicated thread in the Process Manager (called the control thread) listens on the all PM's ports for process management directives from CHORUS/MiX subsystem servers. These **control messages** include: UNIX signal messages, debugging messages, and process exit messages. Only the control thread may receive messages on the control port; the control port is not exported to the CHORUS/MiX process.

When the process is the target of a `kill` system call or of a keyboard-generated signal, a signal delivery message is sent to its process control port.

When a CHORUS/MiX process performs a `ptrace` system call to initiate a debugging session, the PM creates a debug port and sends it to the debugger. All `ptrace` functions performed by the debugger are translated into messages and sent to the debug port. Since these interactions are based on CHORUS IPC, the process and its debugger may reside on different sites.

6. A single control port is required for CHORUS/MiX processes; multiple control ports may be needed by other subsystem implementations.

A process context contains the control port of its parent. When the process exits, an exit status message is sent to its parent's control port. This status information is stored in the parent's process context where it can be retrieved using the `wait` system call.

4.3.4 Process Resources

A process refers to all server-managed resources by using their server-provided capabilities. Open files and devices, current and root directories, and text and data segments are notable examples of such resources. These capabilities are standard CHORUS capabilities and therefore can be used in conjuction with the CHORUS mapper protocol, if appropriate.

For example, opening a device file associates the capability provided by the device's server with a UNIX file descriptor. The capability will be constructed from the port of the Device Manager and a reference to the device within the manager.

All subsequent system calls pertaining to that device will be translated into messages and sent directly to the appropriate server. There is no need to locate the server again.

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` 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 (Figure 16 [1]).

Suppose that the pathname of the file is `/fs/piano/usr/fred/myfile` and `/fs/piano` is the symbolic port of a File Manager running on a site named `piano`. This pathname will be sent to the File Manager containing the root directory of the process (the `pipo` File Manager in the example). That File Manager will start the analysis of the pathname, discover that `piano` is a symbolic port, and propogate the message with the unanalyzed portion of the pathname (`/usr/fred/myfile`), to the symbolic port.

The `piano` File Manager will receive the message, complete the analysis of the pathname, open the file, build the associated capability and send the capability back to the client process that issued the `open` request (Figure 16 [3]).

Any subsequent request on that open file will be sent directly to the File Manager on `piano`. The `pipo` File Manager will not be involved in further interacations.

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.

1. the calling u_thread performs a trap handled by the local Process Manager. The PM will :
   a. determine whether the pathname is relative or absolute, and therefore whether to use the File Manager of the root directory or the current directory;
   b. invoke by RPC the File Manager to translate the binary file pathname into two capabilities, used later to map text and data into the process address space;
   c. if the child execution site is different from the current execution site, test the child execution site for validity;
   d. prepare a request with:
      - the `Proc` and `UThread` contexts of the calling u_thread;
      - the arguments and environment given as `exec` parameters;

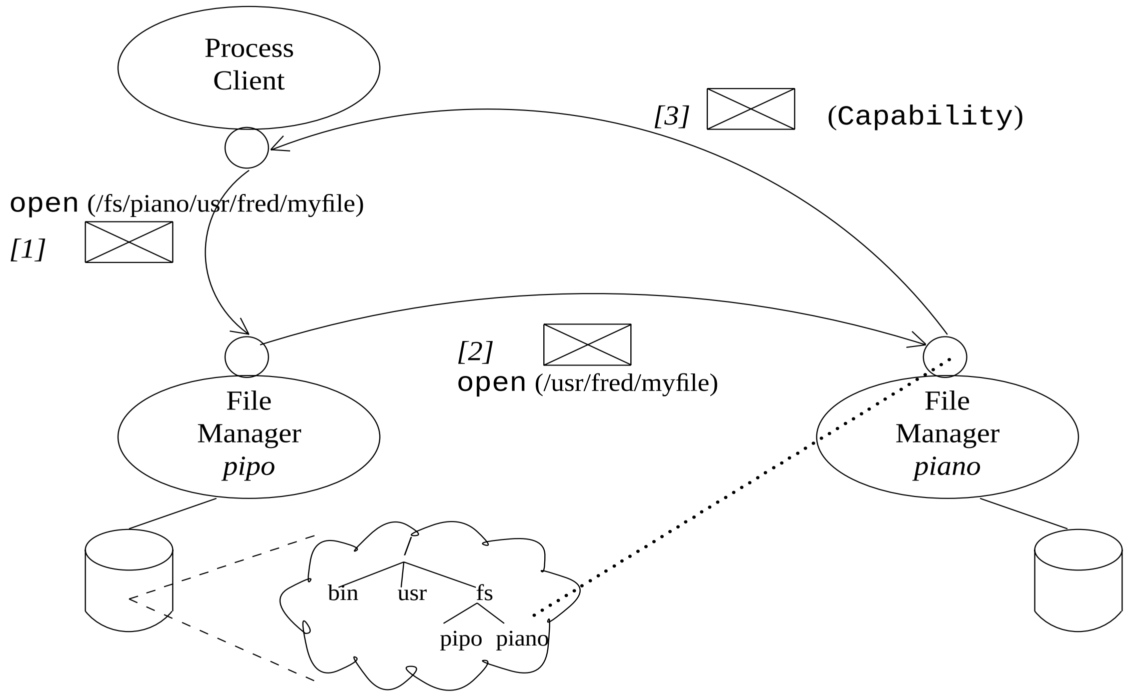

Figure 16. – File Access

- all information returned by the File Manager that characterizes the binary file.

e. perform an RPC to the Process Manager of the target creation site by means of the Process Manager port group.

2. The Process Manager of the remote creation site receives the request. One of its threads initializes a `Proc` context for the new process using information such as PID's, elapsed time, and open file capabilities, contained within this message. The Process Manager creates new CHORUS entities which implement the process, such as an actor, a control thread, a control port, and memory regions. It then proceeds with UNIX process initialization:
   - it installs arguments and the environment strings in the process address space;
   - it sends close messages to appropriate File Managers, Socket Managers and Device Managers to close open files marked ‘‘close-on-exec’’;
   - it creates one u_thread, which will start executing the new program (after `exec`, all processes are initially single-threaded). It initializes the signal context of the created u_thread using the signal context of the calling u_thread which is present in the **request message**.
   - it sends a reply message to the u_thread that originally invoked the `exec` system call.

3. The calling u_thread receives the reply message, frees the `Proc` and `U_thread` contexts of its process and removes the actor implementing the process. Upon actor destruction, the CHORUS Nucleus frees all CHORUS entities associated with this actor.

4.5 Other UNIX Extensions

The CHORUS implementation of the UNIX subsystem has lead to several significant extensions which offer, at the UNIX subsystem level, access to CHORUS functionalities:

4.5.1 IPC

UNIX processes running on CHORUS can communicate with other UNIX processes, 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 Memory management

UNIX processes running on CHORUS can create, delete and share memory regions.

4.5.3 Real-Time

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

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

Moreover, for interrupt processing, UNIX servers may immediately process an interrupt within the interrupt context or defer the majority of the processing to be handled by a dedicated thread executing within the server context.

This functionality allows CHORUS/MiX device drivers to mask interrupts for shorter periods of time than they are masked in many standard UNIX implementations. Thus, with a little tuning, real-time applications can be made to run in a UNIX environment with better response time to external events.

5. CONCLUSION

CHORUS was designed with the intention of supporting fully-functional, industrial quality operating system environments. Thus, the inherent trade-off between performance and richness of the design was often made in favor of performance.

Making the CHORUS Nucleus facilities *generic* prevented the introduction of features with complex semantics. Features such as stringent security mechisms, application-oriented protocols, and fault tolerance strategies, do not appear in the CHORUS Nucleus. The CHORUS Nucleus provides, instead, the building blocks with which to construct these features inside subsystems.

CHORUS provides effective, high performance solutions to some of the issues known to cause difficulties to system designers:

- Exceptions are posted by the Nucleus to a port chosen by the actor program. This simple mechanism allows a user actor to apply its own strategy for handling exceptions and, because of the nature of ports, it extends transparently to distributed systems.

  Exceptions can also be associated directly with actor routines, for high performance within system actors.
- Debugging within CHORUS distributed systems is facilitated since resources are isolated within actors and since the message passing paradigm provides explicit and clear interactions between actors.
- The CHORUS modular structure allows binary compatibility with UNIX in CHORUS–V3, while maintaining a well structured, portable and efficient implementation.

The experience of four CHORUS versions has validated the CHORUS concepts. Unique Identifiers provide global, location independent names which form the basis for resource location within the a CHORUS distributed system. Actors and threads provide modular, high-performance, multi-threaded computational units. Messages, ports, and port groups provide the underlying communication mechanism with which CHORUS computational entities are bound together to construct distributed systems.

The CHORUS technology has the following features:

- it uses a communication-based architecture, relying on a minimal Nucleus which integrates distributed processing and communication at the lowest level, and which implements generic services used by a set of subsystem servers to extend standard operating system interfaces. A UNIX† subsystem has been developed; other subsystems such as OS/2 and object-oriented systems are planned;
- the real-time Nucleus provides real-time services which are accessible by system programmers;
- it is 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.

The CHORUS technology has been designed to build a new generation of open, distributed, and scalable operating systems. In addition to providing the basis for the emulation of existing operating systems, CHORUS technology provides a means by which these subsystem interfaces can be extended transparently to exploit distributed environments. CHORUS technology can be used in conjunction with these subsystems, or independently, to assemble high-performance distributed applications.

6. GLOSSARY OF CHORUS TERMS

actor	A collection of resources within a CHORUS site. A CHORUS actor can contain memory regions, ports, and threads. When created, an actor contains only its default port. CHORUS supports user, system, and supervisor actors.
actor context switch	A switch from the context of one actor to another. An actor context switch may imply switching address space contexts and thread contexts.
address space context switch	A switch from one address space to another, causing the contents of virtual memory to change.
capability	A unique handle, the possession of which grants the possessor rights to perform an operation. Within the CHORUS system, capabilities consist of the concatenation of a port and a key. The port identifies a server and the key identifies the object within the server. The port is represented as a UI. The key is a 64 bit identifier, specific to the server.
CHORUS	An operating system which consists of a small distributed real-time Nucleus and a set of subsystem servers. The Nucleus supports location transparent message-passing, thread scheduling, supervisor, and virtual memory operations. Subsystems supporting more complicated operating system abstractions are built using these primitives.
CHORUS Domain	A collection of interconnected CHORUS sites which define a non-overlapping unique identifier namespace.
CHORUS Inter-Process Communication Manager	The section of the CHORUS Nucleus that provides asynchronous and synchronous location transparent message passing facilities.
CHORUS Real-Time Executive	The section of the CHORUS Nucleus that manages scheduling of threads. It controls allocation of processors, provides fine-grained synchronization and a priority-based preemptive scheduling.
CHORUS Supervisor	The section of the CHORUS Nucleus that manages site hardware. It dispatches interrupts, traps and exceptions delivered by the hardware.
CHORUS Virtual Memory Manager	The section of the CHORUS Nucleus that manipulates virtual memory hardware and local physical memory resources.
CHORUS/MiX	A set of CHORUS actors which cooperate to provide an enhanced UNIX subsystem interface. CHORUS/MiX exports features from CHORUS such as CHORUS IPC, multi-threaded processes, real-time scheduling, a unified network file system, and location transparent naming of entities within the CHORUS domain. Currently, a CHORUS/MiX subsystem consists of a Process Manager, and optionally a File Manager, a Socket Manager, and a Device Manager.
CHORUS/MiX process	A UNIX process abstraction, as implemented within CHORUS/MiX. A CHORUS/MiX process is mapped onto an actor, threads, and memory regions. In addition, CHORUS/MiX associates a control thread and a control port with the CHORUS/MiX process. The control thread

	performs actions on behalf of the process; it and the control port are hidden from the CHORUS/MiX process.
control message	A message sent to a CHORUS/MiX process control port as a result of an exceptional event. Control messages will be sent in response to one of the following conditions: hardware exceptions, UNIX signals, debugging requests, or upon the exit of a child process.
control port	The port that the Process Manager attaches to every CHORUS/MiX process. It is used to send control messages to the process. The control port is not exported to the CHORUS/MiX process.
control thread	The thread that the Process Manager attaches to every CHORUS/MiX process. It accepts control messages on the control port. The control thread executes as a supervisor thread at a higher priority than all other threads of the actor. The control thread is not exported to the CHORUS/MiX process.
default port	The port that is attached to each CHORUS actor when it is created. The unique identifier of this port is the same as that of its actor. The default port may not be deleted or migrated.
Device Manager	A CHORUS/MiX subsystem actor which manages physical devices such as keyboards, bit-map displays, or magnetic tapes.
DM	See Device Manager.
execution privilege	See privileged mode.
File Manager	A CHORUS/MiX subsystem actor which has two functions: to coordinate with Process Managers to project a unified, domain-wide file system and to act as an external mapper to fulfill segment related requests. All UNIX pathname analysis is done by the File Manager. If a symbolic port or device is encountered during pathname analysis, the request will be forwarded to the appropriate manager.
FM	See File Manager.
group manipulation key	A 64 bit key used to protect a port group on which a port group operation is to take place.
inter-process communication	A facility that allows threads to exchange information in the form of collections of bytes called messages. Messages are addressed to ports. The CHORUS IPC system manages the port namespace. The CHORUS IPC facility is location transparent; threads executing within actors residing on different sites may use CHORUS IPC to exchange messages transparently. As primitive operations, CHORUS IPC supports asynchronous message send operations, message receive operations and remote procedure call operations.
IPC	See inter-process communication.
IPCM	See IPC Manager.
IPC Manager	A CHORUS/MiX subsystem actor which implements the UNIX IPC mechanisms: messages, semaphores and shared memory.

kernel	See Nucleus.
key	A 64 bit server-specific value which, in conjunction with a server port identifier, forms a capability. The capability can be used to access a resource within the server.
Key Manager	A CHORUS/MiX subsystem actor which maintains a mapping between user provided keys and CHORUS/MiX internal descriptor, in the context of UNIX IPC services.
KM	See Key Manager.
Local Cache	A local cache is the in-memory representation of the contents of a CHORUS segment; if an offset within the segment is not in primary memory when referenced or written, the cache will be loaded by invoking its mapper. Local caches are created as a result of any segment-related Nucleus system calls. Multiple simultaneous uses of a segment on one site always refer to the same local cache. Note that when multiple sites contain caches for the same segment, consistency must be maintained by a segment's mapper; the virtual memory system merely provides the primitives to make distributed cache consistency possible.
LI	See local identifier.
local identifier	A 32 bit context-dependent identifier, used to identify CHORUS Nucleus resources of a given actor.
mapper	A server that conforms to the CHORUS mapper interface and is used to provide a backing storage facility for CHORUS actors. Mappers implement objects known as segments. Portions of the information within these objects can be mapped into actor address spaces. Page faults generated by reading the memory associated with a mapped object will, in turn, produce requests to the mapper for the corresponding data from the object. When the Nucleus wishes to free modified pages, it generates mapper requests to write back the modified data. CHORUS also supports an interface that allows actors to directly make changes to mapper-supported objects.
message	An untyped sequence of bytes which represents information to be sent from one port to another via CHORUS IPC.
NDM	See Network Device Manager.
Network Device Manager	A CHORUS server which controls network devices. The Network Device Manager is invoked by higher-level communications servers.
network file system	A file system whose contents are distributed over a network. Within CHORUS/MiX, a completely transparent network file system is provided using symbolic ports to interconnect File Managers. File Managers add the file hierarchies of newly booted sites to their own hierarchies, using a network-wide naming convention. Thus, within CHORUS/MiX, any file within any site is transparently accessible, using an easily constructed name.
Network Manager	A CHORUS server responsible for aiding the Nucleus during remote communication. The Network Manager implements the UILS, remote CHORUS IPC, and RPC. The Network Manager uses the Network

	Device Manager for low-level communication.
NM	See Network Manager.
Nucleus	A low-level executive which provides the base functions with which more complex subsystems or applications can be built. In CHORUS, the Nucleus provides virtual memory, scheduling, and message passing facilities. It is also responsible for managing site resources.
Object Manager	See File Manager.
OM	See File Manager.
PM	See Process Manager.
port	A CHORUS inter-process communication entity. Threads send and receive messages on ports, which act as globally-named message queues. Ports are initially attached to a specified actor but can be migrated to another actor. Only threads within that actor have the right to receive messages on it. Ports are named by unique identifiers; having knowledge of a port UI gives a thread the right to send messages to that port. Ports are location-transparent; a thread within an actor may send a message to the port of another actor without knowing the current location of that port.
port group	A collection of ports that are addressed as a group to perform some communication operation. Port groups can be used to send messages to one of a set of ports or to multi-cast messages to several ports simultaneously. A port can be a member of several port groups.
privileged mode	A CHORUS Nucleus concept that governs whether or not a thread is allowed to perform sensitive hardware instructions, such as modifying the state of the memory management unit or performing low-level device I/O. Typically, this concept maps directly to a hardware facility controlled by the setting of a status register, maintained within a thread context.
Process Manager	A CHORUS/MiX subsystem actor which implements UNIX process semantics. In particular, the PM implements UNIX system calls related to process and signal management such as `fork`, `exec`, and `kill`. The Process Manager catches all UNIX system call traps and will invoke other CHORUS/MiX servers if necessary.
protection identifier	A fixed-length value associated by the Nucleus with all actors and ports which is used to stamp outgoing messages. The Nucleus does not attach any semantic value to these identifiers; it is only responsible for maintaining them, for protecting them from modification, and for inserting them into the headers of messages exchanged by actors. The receiver of the message may use these identifiers to authenticate the sender.
region	A contiguous range of virtual addresses within a process, treated as a unit by the CHORUS virtual memory system. A region is associated with a segment by the CHORUS virtual memory system. Requests to read or modify data within a region are converted by the virtual memory system into requests to read or modify data within that segment. Regions are assigned virtual memory protection attributes.

remote procedure call	An inter-process communication facility, through which a thread can simulate a procedure call using a message transmission and reception. To perform an RPC, a client thread sends a message which requests a service and provides arguments to the service request. That thread then blocks, awaiting a reply to the message. A server receives the message and performs the service according to the arguments within the message. The server then replies with any result-arguments the service may have produced.
reply message	A message sent in response to a request for a service, typically holding the result of the service.
request message	A message sent to a request port to request a service.
request port	A port on which servers receive service requests. This port does not have special status among all ports owned by the server.
resource	A basic entity used within a CHORUS system. A resource might consist of data or processors which act upon data. Basic CHORUS resources include ports, messages, threads, actors and sites. CHORUS subsystems provide as resources higher-level abstractions such as files and processes.
RPC	See remote procedure call.
segment	An encapsulation of data within a CHORUS system. Segments can be read or written by actors, using direct or mapped interfaces. Portions of a segment can be mapped into an actor's address space and the data within the segment can be read or modified by accessing the associated address ranges. Segments typically represent some form of backing store, such as the contents of a file.
server	An actor that provides services to a set of clients. Within the client-server model, a client sends a request to a server which will perform some action and send a reply. Requests and replies are typically made using inter-process communication mechanisms.
site	A grouping of tightly-coupled physical resources controlled by a single CHORUS Nucleus. These physical resources include one or more processors, central memory, and attached I/O devices.
SM	See Socket Manager.
Socket Manager	A CHORUS/MiX subsystem actor that implements 4.3 BSD UNIX socket semantics for the Internet address family. The Socket Manager uses the Network Device Manager for low-level communication.
subsystem	A collection of CHORUS servers and libraries that export a given operating system interface.
subsystem interface	The set of operations exported by a CHORUS subsystem. They typically represent an operating system interface that is to be emulated within the CHORUS system.
supervisor actor	A CHORUS actor whose threads are always supervisor threads. Currently, supervisor actors only contain regions within the shared, protected system address space.
supervisor thread	A CHORUS thread that operates in privileged mode. Threads of supervisor actors are always supervisor threads. Threads of user actors

	temporarily become supervisor threads when they are in the process of executing hardware traps.
symbolic links	CHORUS/MiX symbolic links behave exactly as symbolic links in BSD UNIX. When a symbolic link is encountered during pathname analysis, the value of the symbolic link and the remainder of the original pathname are concatenated and the analysis continues.
symbolic port	An extension to the UNIX file system, which associates a file system entry with the CHORUS port of a server able to respond to file system requests. For example, when a symbolic port is encountered during pathname analysis, the remainder of the pathname is forwarded in a request to the port with which it is associated. Examples of the use of symbolic ports include interconnecting file systems, implementing a mirrored file system, and providing symbolic process tree manipulation.
system actors	Actors whose threads are trusted.
system address space	A range of virtual addresses, shared by all actors. Only supervisor threads are allowed to read and modify memory within this range of addresses.
thread	A flow of control within an actor in the CHORUS system. Each thread is associated with an actor and defines a unique execution state. An actor may contain multiple threads; the threads share the resources of that actor, such as memory regions and ports, and are scheduled independently.
thread context switch	A switch from one thread context to another, causing a new machine state to be loaded. This thread context switch may or may not imply an address space context switch, depending on whether or not the new thread executes within the same address space as the previous thread.
trusted thread	A CHORUS thread that is allowed by the CHORUS Nucleus to perform sensitive Nucleus operations, such as changing protection identifier values. Threads of system actors and supervisor threads are trusted by the CHORUS Nucleus.
UI	See unique identifier.
UILS	See unique identifier location service.
unique identifier	An identifier used for naming resources within the CHORUS system. These identifiers are guaranteed to be unique over time for all CHORUS sites within a CHORUS domain. They are constructed from a large, sparse name space.
unique identifier location service	A CHORUS service that permits the CHORUS Nucleus to determine the site location of an object which is represented by a unique identifier.
user actors	CHORUS actors whose threads are user threads. Currently, user actors have private user address spaces.
user address space	A range of virtual addresses whose contents are private to each user actor.
user threads	CHORUS threads that operate in unprivileged mode. User threads temporarily become supervisor threads while executing hardware traps.

u_thread	A thread within a CHORUS/MiX process. U_threads are implemented using CHORUS threads and have extra state associated with them which is used to provide UNIX semantics. For accounting reasons, most operations performed by u_threads are dispatched by the Process Manager.

7. ACKNOWLEDGEMENTS

Jean-Pierre Ansart, Allan Bricker, Philippe Brun, Hugo Coyote, Corinne Delorme, Jean-Jacques Germond, Steve Goldberg, Pierre Lebée, Frédéric Lung, Marc Maathuis, Denis Metral-Charvet, Douglas Orr, 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

[Bani80] 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).

[Bani82] Jean-Serge Banino and Jean-Charles Fabre, ‘‘Distributed Coupled Actors: a CHORUS Proposal for Reliability,’’ in *IEEE 3rd. International Conference on Distributed Computing Systems Proc.*, Fort-Lauderdale, FL, (18-22 October 1982), 7 p.

[Guil82] 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), 162 p.

[Guil82a] Marc Guillemont, ‘‘The CHORUS Distributed Operating System: Design and Implementation,’’ in *ACM International Symposium on Local Computer Networks Proc.*, Florence, Italy, (April 1982), pp. 207-223.

[Zimm81] Hubert Zimmermann, Jean-Serge Banino, Alain Caristan, Marc Guillemont, and Gérard Morisset, ‘‘Basic Concepts for the Support of Distributed Systems: the CHORUS Approach,’’ in *IEEE 2nd. International Conference on Distributed Computing Systems Proc.*, Versailles, France, (April 1981), pp. 60-66.

8.2 CHORUS–V1

[Bani85] Jean-Serge Banino, Jean-Charles Fabre, Marc Guillemont, Gérard Morisset, and Marc Rozier, ‘‘Some Fault-Tolerant Aspects of the CHORUS Distributed System,’’ in *IEEE 5th. International Conference on Distributed Computing Systems Proc.*, Denver, CO, (13-17 May 1985), pp. 430-437.

[Bani85a] 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).

[Fabr82] Jean-Charles Fabre, ‘‘Un Mécanisme de Tolérance aux Pannes dans l’Architecture Répartie CHORUS,’’ Thèse de Doctorat, Université Paul Sabatier, Toulouse, France, (Octobre 1982), 205 p.

[Guil84] 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), 78 p.

[Herr85] Frédéric Herrmann, ‘‘Décentralisation de Fonctions Système: Application à la Gestion de Fichiers,’’ Thèse de Docteur Ingénieur, Université Paul Sabatier, Toulouse, France, (Septembre 1985), 120 p.

[Rozz85] Ahmad Rozz, ‘‘La Gestion des Fichiers dans le Système Réparti CHORUS,’’ Thèse de Docteur Ingénieur, Université Paul Sabatier, Toulouse, France, (Octobre 1985), 144 p.

[Sena83] 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), 200 p.

[Zimm84] 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

[Arma86] François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, ‘‘Towards a Distributed UNIX System – the CHORUS Approach,’’ in *EUUG Autumn’86 Conference Proc.*, Manchester, UK, (22-24 September 1986), pp. 413-431.

[Herr87] Frédéric Herrmann, ‘‘CHORUS : un Environnement pour le Développement et l’Exécution d’Applications Réparties,’’ *Technique et Science Informatique*, vol. 6, no. 2, (Mars 1987), pp. 162-165.

[Lega86] 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), 150 p.

[Lega88] José Legatheaux-Martins and Yolande Berbers, ‘‘La Désignation dans les Systèmes d’Exploitation Répartis,’’ *Technique et Science Informatique*, vol. 7, no. 4, (Juillet 1988), pp. 359-372.

[Mzou88] Azzeddine Mzouri, ‘‘Les Protocoles de Communication dans un Système Réparti,’’ Thèse de Doctorat, Université Paris-Sud, Orsay, France, (Janvier 1988), 190 p.

[Papa88] Mario Papageorgiou, ‘‘Les Systèmes de Gestion de Fichiers Répartis,’’ Thèse de Doctorat, Université Paris-6, Paris, France, (Janvier 1988), 237 p.

[Papa88a] Mario Papageorgiou, ‘‘Le Système de Gestion de Fichiers Répartis dans CHORUS,’’ *Technique et Science Informatique*, vol. 7, no. 4, (Juillet 1988), pp. 373-384.

[Rozi86] Marc Rozier, ‘‘Expression et Réalisation du Contrôle d’Execution dans un Système Réparti,’’ Thèse de Doctorat, Institut National Polytechnique, , Grenoble, France, (Octobre 1986), 184 p.

[Rozi87] Marc Rozier and José Legatheaux-Martins, ‘‘The CHORUS Distributed Operating System: Some Design Issues,’’ in *Distributed Operating Systems, Theory and Practice*, Yakup Paker, Jean-Pierre Banâtre and Muslim Bozyigit ed., NATO ASI Series, vol. F28, Springer Verlag, Berlin, (1987), pp. 261-287.

8.4 CHORUS–V3

[Abro89] Vadim Abrossimov, Marc Rozier, and Michel Gien, ‘‘Virtual Memory Management in CHORUS,’’ in *Lecture Notes in Computer Sciences, Workshop on Progress in Distributed Systems and Distributed Systems Management*, Springer-Verlag, Berlin, Germany, (18-19 April 1989), 20 p. Chorus systèmes Technical Report CS/TR-89-30

[Abro89a] Vadim Abrossimov, Marc Rozier, and Marc Shapiro, ‘‘Generic Virtual Memory Management for Operating System Kernels,’’ in *Proc. of 12th. ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, (3-6 December 1989), 20 p. Chorus systèmes Technical Report CS/TR-89-18.2

[Arma89] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, ‘‘Revolution 89, or Distributing UNIX Brings it Back to its Original Virtues,’’ in *Proc. of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, Ft. Lauderdale, FL, (5-6 October 1989), pp. 153-174. Chorus systèmes Technical Report CS/TR-89-36.1

[Arma90] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, ‘‘Multi-threaded Processes in Chorus/MIX,’’ in *Proc. of EUUG Spring’90 Conference*, Munich, Germany , (23-27 April 1990), pp. 1-13. Chorus systèmes Technical Report CS/TR-89-37.3

[Coyo89] Hugo Coyote, ‘‘Spécification et Réalisation d’un Système de Fichiers Fiables pour le Système d’Exploitation Réparti CHORUS ,’’ Thèse de Doctorat, Université Paris VI, Paris, France, (Juin 1989), 148 p. Chorus systèmes Technical Report CS/TR-89-39

[Guil89] M. Guillemont, ‘‘CHORUS: A support for Distributed and Reconfigurable Ada Software,’’ , European Space Agency, Noordwijk, The Netherlands, (24-26 October 1989), pp. 263-268. Chorus systèmes Technical Report CS/TR-89-40

[Herr88] 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,’’ , Cascais, Portugal, (3-7 October 1988).

[Métr89] Denis Métral-Charvet and François Saint-Lu, ‘‘Le Système CHORUS: Temps Réel, Répartition et UNIX Intégrés,’’ , AFUU, Paris, France, (2-3 Mars 1989), pp. 19-42. Chorus systèmes Technical Report CS/TR-89-3

[Phil89] Laurent Philippe and Bénédicte Herrmann, ‘‘Un Serveur de Tubes UNIX pour CHORUS,’’ , AFUU, Paris, France, (2-3 Mars 1989), pp. 277-290. Chorus systèmes Technical Report CS/TR-89-2

[Rozi88] Marc Rozier, Will Neuhauser, and Michel Gien, ‘‘Scalability in Distributed Real-Time Operating Systems,’’ in *Proc. of 5th. Workshop on Real-Time Software and Operating Systems*, IEEE, Washington, DC, (May 1988), 7 p.

[Rozi88] Marc Rozier, and Michel Gien, ‘‘Resource-level Autonomy in CHORUS,’’ in *1988 ACM SIGOPS European Workshop on "Autonomy and Interdependence in Distributed Systems?"*, Cambridge, UK, (18-21 September 1988).

9. REFERENCES

[Acce86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, ‘‘Mach: A New Kernel Foundation for UNIX Development,’’ in *USENIX Summer’86 Conference Proc.*, Atlanta, GA, (9-13 June 1986), pp. 93-112.

[Bét70] 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*, vol. 13, no. 12, (December 1970).

[Cher88] David Cheriton, ‘‘The Unified Management of Memory in the V Distributed System,’’ Technical Report, Computer Science, Stanford University, Stanford, CA, (1988).

[Cher88a] David Cheriton, ‘‘The V Distributed System,’’ *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333.

[Gien83] Michel Gien, ‘‘The SOL Operating System,’’ in *Usenix Summer’83 Conference*, Toronto, ON, (July 1983), pp. 75-78.

[Ging87] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, ‘‘Virtual Memory Architecture in SunOS ,’’ in *Usenix Summer’87 Conference*, Phoenix, AR, (8-12 June 1987), pp. 81-94.

[Lega88] José Legatheaux-Martins and Yolande Berbers, ‘‘La Désignation dans les Systèmes d’Exploitation Répartis,’’ *Technique et Science Informatique*, vol. 7, no. 4, (Juillet 1988), pp. 359-372.

[Li86] Kai Li, ‘‘Shared Virtual Memory on Loosely Coupled Multiprocessors,’’ PhD. Thesis, Yale University, New-Haven, CT, (September 1986).

[Mora88] Joseph P. Moran, ‘‘SunOS Virtual Memory Implementation,’’ in *EUUG Spring’88 Conference*, London, UK, (11-15 April 1988), pp. 285-300.

[Mull87] Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers 1984 -1987*, CWI Tract No. 41, Amsterdam, Netherlands, (1987), 309 p.

[Nels88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, ‘‘Caching in the Sprite Network File System,’’ *ACM Transactions on Computer Systems*, vol. 6, no. 1, (February 1988), pp. 134-154.

[Pouz82] Louis Pouzin et al., *The CYCLADES Computer Network - Towards Layered Network Architectures*, Monograph Series of the ICCC, 2, Elsevier Publishing Company, Inc, New-York, NY, (1982), 387 p. ISBN 0-444-86482-2

[Pres86] David L. Presotto, ‘‘The Eight Edition UNIX Connection Service,’’ in *EUUG Spring’86 Conference Proc.*, Florence, Italy, (21-24 April 1986), 10 p.

[Rash87] 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,’’ in *ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, (October 1987), pp. 31-39.

[Tane86] Andrew S. Tanenbaum, Sape J. Mullender, and Robert van Renesse, ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ in *IEEE 6th. International Conference on Distributed Computing Systems*, CWI Tract No. 41, Cambridge, MA, (19-23 May 1986), pp. 558-563.

[Wein86] Peter J. Weinberger, ‘‘The Eight Edition Remote Filesystem,’’ in *EUUG Spring’86 Conference*, Florence, Italy, (21-24 April 1986), 1 p.

CONTENTS

LIST OF FIGURES

LIST OF TABLES