CS/TR-88-30

CHORUS, a New Technology for Building UNIX Systems

Fre'de'ric Herrmann, Franc/ois Armand, Marc Rozier, Michel Gien
V. Abrossimov, I. Boule, M. Guillemont, P. Le'onard, S. Langlois, W. Neuhauser

ABSTRACT

The CHORUS[®] technology has been designed for building ‘‘new generations’’ of open, distributed, and scalable Operating Systems. CHORUS has the following main characteristics:

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

– 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,

– real time services provided by a real-time executive, and accessible by ‘‘system programmers’’ at the different system levels.

CHORUS–V3 is the current version of the CHORUS Distributed Operating System, developed by Chorus syste‘mes. Earlier versions had been studied and implemented within the Chorus research project at INRIA between 1979 and 1986.

This paper summarizes the facilities provided by the CHORUS–V3 Nucleus, and describes the UNIX Subsystem built with the CHORUS technology that provides:

– binary compatibility with UNIX,

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

[®] CHORUS is a registered trademark of Chorus syste‘mes
[†] UNIX is a registered trademark of AT&T

Invited paper at the EUUG Autumn’88 Conference, Cascais (Portugal)

1 Introduction

The CHORUS technology emphasizes *portability*, *modularity* and *scalability*, permitting the configuration of systems on a wider range of operational environments than current UNIX systems.

The CHORUS architecture implements modularity down to the lowest levels. Each system resource is clearly isolated and managed by a dedicated system component, the ‘‘glue’’ being provided by the communication services.

This technology has been used to implement a distributed UNIX system, as a set of servers using the generic services provided by the CHORUS *Nucleus*.

After a quick presentation of the CHORUS Architecture, and the facilities provided by its Nucleus, the paper focuses on UNIX implementation. Extensions to UNIX services concerning real-time, multi-thread processes, distributed applications and servers are also outlined.

2 The CHORUS Architecture

2.1 Objectives

CHORUS aims at providing distributed computing facilities resulting in two major benefits:

- better ability to manage the increasing distribution of computation,
- better coupling (integration) between levels of computing.

Managing distributed computing tasks

CHORUS aims at allowing better control of the ‘‘distribution’’ of tasks among processing resources in a network or among the processors of a multi-processor architecture. In addition to using the network for communicating data, CHORUS is intended to provide means to use computing resources more effectively by spreading intensive tasks around or by placing specialized tasks on the most appropriate computers (or processors).

Coupling levels of computing

CHORUS aims at coupling different levels of computing − interactive, batch, and real-time − on the same system. CHORUS is modular and portable so that computer systems can be tailored to the processing needs at each level and yet work together. Even real-time systems with tight constraints on size, memory, storage and response time can operate safely and deterministically in a CHORUS based UNIX environment.

Scalability and modularity for adaptability to varied needs

CHORUS is designed to work in restrictive environments demanding real-time response as well as general-purpose computing needs. The CHORUS architecture has been designed modularly so it can be scaled to the most appropriate size. Minimal systems for restrictive environments can be constructed with the smallest subset of system functions truly needed (e.g., no filesystem or no virtual memory). New user-created system functions can be dynamically added to the system. Furthermore, different operating system interfaces can be layered on top of the CHORUS Nucleus; UNIX is one of them.

2.2 Background and Related Work

‘‘Chorus’’ was a research project on Distributed Systems at INRIA in France from 1979 to 1986. Three iterations were developed, referred to as CHORUS-V0, CHORUS-V1, and CHORUS-V2, all

based
on a communications-oriented kernel. CHORUS basic concept for handling distributed computing, for system as well as application services, is for a ‘‘Nucleus’’ to manage the exchange of ‘‘Messages’’ between ‘‘Ports’’ attached to ‘‘Actors’’.

While early versions of CHORUS had a custom interface, CHORUS–V2 is compatible with UNIX System V, and has been used as a basis for supporting half a dozen of research distributed applications.

CHORUS-V3Rozi08a is the current version, developed by Chorus syste‘mes. It builds on previous CHORUS experience 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.

CHORUS-V3 message-passing Nucleus compares to the V-system of Stanford University, distributed virtual memory and ‘‘threads’’ are similar to that of Mach of CMU, network addressing incorporates ideas from Amoeba of the University of Amsterdam, and uniform file naming is based on a scheme similar to the one used in the 9th. Edition UNIX from the Bell Laboratories.

2.3 Concepts and Services

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

2.3.1 The CHORUS Nucleus

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

1. Local services:

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

2. Global services:

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

2.3.2 The Subsystems

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

2.3.3 System Interfaces

A CHORUS system exhibits several levels of interfaces:

- **Nucleus Interface**: The Nucleus interface is composed of a set of procedures permitting the use of the services of the Nucleus. These procedures directly call ‘‘traps’’ corresponding to those services. If the Nucleus cannot render the service directly, it communicates with a distant Nucleus via the IPC.

- **Subsystem Interface**: Applications have access to system services through the system calls of standard Subsystems. This interface is composed of a set of procedures accessing the Nucleus interface, and some Subsystem specific protected data. If a service cannot be rendered directly from this information, these procedures ‘‘call’’ (RPC) the services provided by System Servers.

  CHORUS offers *system programmers* the means of constructing multiple protected interfaces allowing them to build several Subsystem interfaces, that can be used concurrently within a CHORUS system.

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

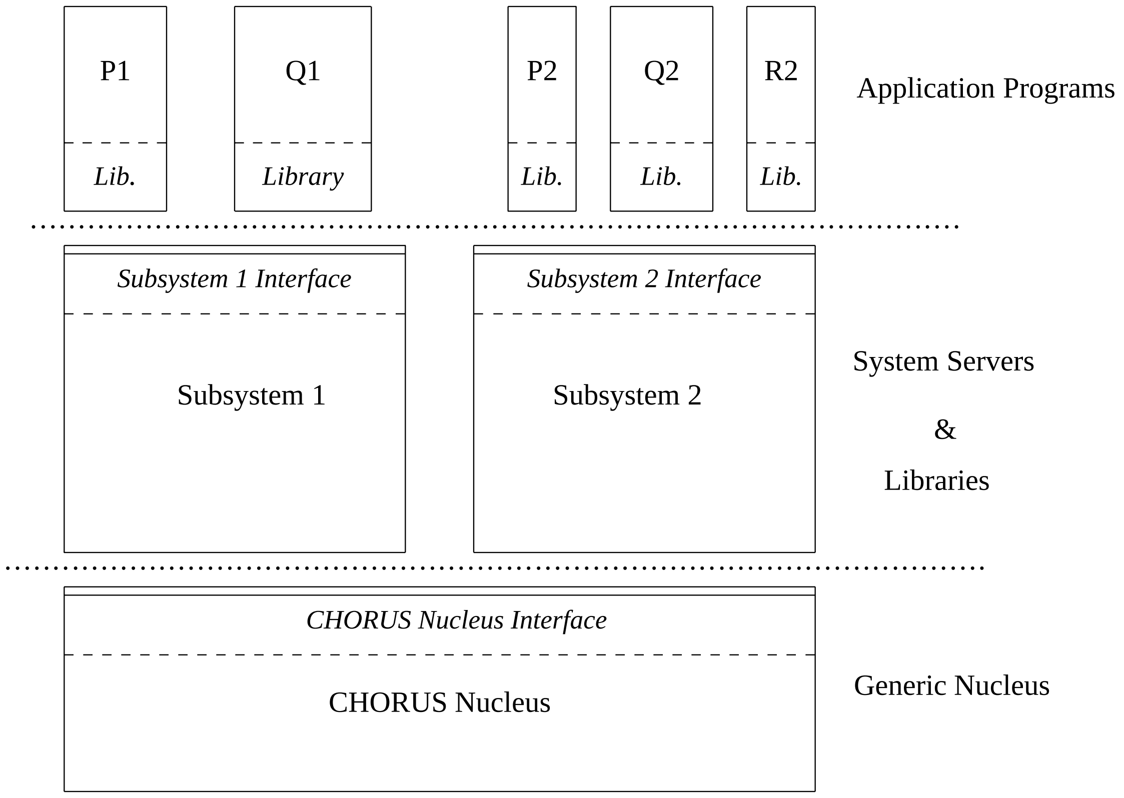

Figure 1. – The CHORUS Architecture

2.3.4 Basic Abstractions Implemented by the CHORUS Nucleus

The CHORUS Nucleus is not built as the core of a specific operating system (e.g., UNIX), rather it provides generic tools for the support of several host systems.

The basic abstractions which are implemented and managed by the CHORUS Nucleus are (Figure 3):

Actor	unit of resource allocation, and memory address space,
Thread	unit of sequential execution
Message	unit of communication
Port, Port Groups	unit of addressing, and Port structuring
Region	unit of allocation of an Actor address space
Unique Identifier (UI)	global name in a CHORUS Domain
Protection Identifier	unit of authentication

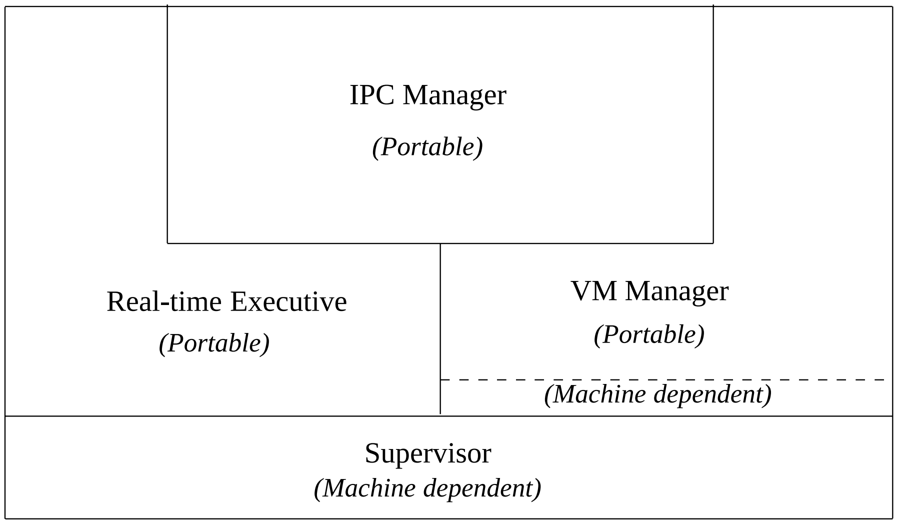

Figure 2. – The CHORUS Nucleus

Each abstraction corresponds to an object class which is private to the CHORUS Nucleus: both the object representation and the operations on the objects are managed by the Nucleus.

Other general and useful abstractions are also managed both by the CHORUS Nucleus and the Actors:

Segment	unit of data encapsulation
Capability	unit of data designation

These two abstractions are imported by the Nucleus and are implemented by dedicated System Actors, according to a well defined specification.

Each of the above abstractions plays a specific role in the System:

An *Actor* encapsulates resources:

- a virtual memory context, the *Regions* of which are coupled with segments that may be persistent or temporary, local or distant,
- a set of threads, sequential flow of control, sharing the actor's virtual memory context,
- a set of ports, which are the sole gates to the external world of an actor and on which it can receive messages.

Threads are characterized by a thread context corresponding to the state of the processor (registers, program counter, stack pointer, etc.). They may interact within an actor through the actor's memory or in general by exchanging messages. They are scheduled independently by the Nucleus according to their respective priorities. They are generally executed in ‘‘*user mode*’’, but may ask – the CHORUS Nucleus – to be executed in ‘‘*system mode*’’, depending on the levels of protection provided by the supporting hardware.

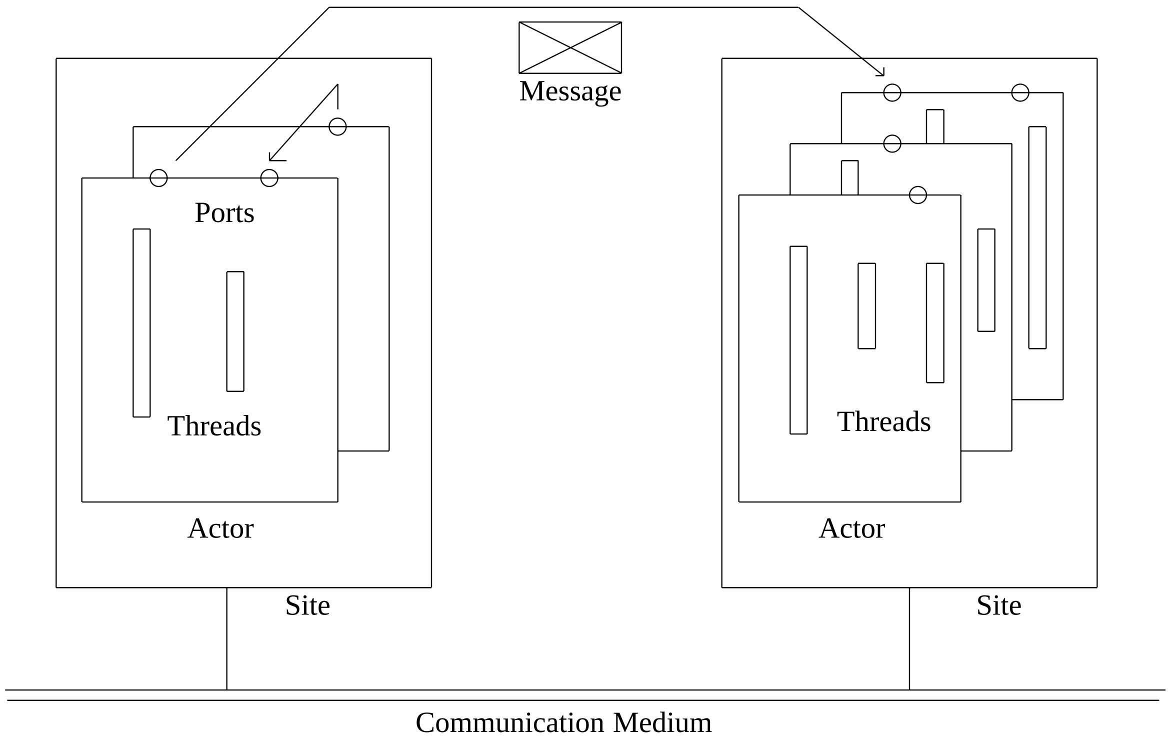

Figure 3. – CHORUS Main Abstractions

Messages are addressed to ports. Upon creation, a port is attached to one actor. A port can be attached to only one actor at a time. However, a port can migrate from one actor to another. Threads of the actor to which the port is attached can receive messages on that port; no other actor can receive those messages. Any thread knowing a port can send messages to it.

Ports can be grouped into *Port Groups* providing multicast or functional addressing facilities. These allow sending messages to all the ports belonging to a group or to one specific port within a group.

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

Actors and threads receive also *Protection Identifiers* used for user authentication.

Capabilities are global names of resources implemented by Servers. They are made of a server's port UI and server specific informations (i.e., internal identification and protection) (Figure 4).

Segments are collections of data (e.g., files) managed by System Servers, called *Mappers* (Figure 5). Segments are designated by capabilities. The interactions between a CHORUS Nucleus and Mappers for managing the mapping of segments into actors regions are ruled by a ''*CHORUS Standard Mappers Interface*'', based on the CHORUS IPC.

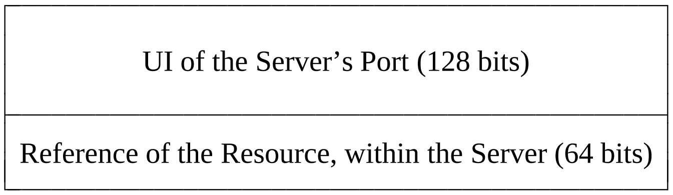
UI of the Server's Port (128 bits)
Reference of the Resource, within the Server (64 bits)

Figure 4. – Structure of a Capability

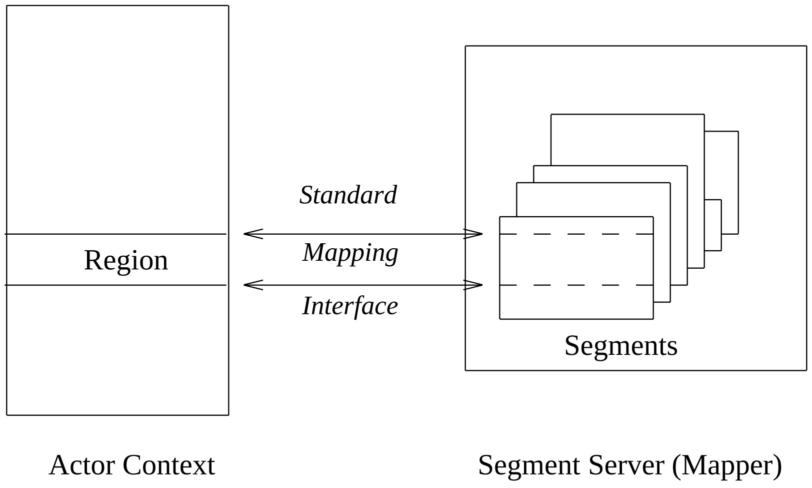

Figure 5. – Context, Regions and Segments

The first Subsystem implemented within the framework of the CHORUS Architecture has been – for obvious reasons – a UNIX Subsystem. The following sections describe the characteristics of the CHORUS UNIX implementation, and the resulting extensions to UNIX services.

3 UNIX Implementation

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

3.1 Objectives

Dynamic (Re-)configuration

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

Openness and Expandability

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

Extending UNIX Functionalities Towards

- Real-Time:
  To extend UNIX with the real-time facilities provided by the low-level CHORUS real-time Executive.
- Distribution:
  To operate UNIX in a distributed environment with no limitations on the types of resources one wants to share. In CHORUS terms this means that:
  — The file system is fully distributed and file access is location independent. File trees can be *automatically interconnected* to provide a name space where all files, whether remote or local, are designated with homogeneous and uniform symbolic names.
  — Operations on processes (at the `fork`/`exec` level as well as at the `shell` level) can be executed regardless of the execution site of these processes; on the other hand, the creation of a child process can be forced to occur on any given compatible site.
  — The network transparent CHORUS IPC is accessible at the UNIX interface level, thus allowing the easy development of distributed applications within the UNIX environment.
  Distribution extensions to standard UNIX services are provided in a natural way (in the UNIX sense) so that existing applications may benefit of those extensions even when directly ported on CHORUS, *without modification or recompilation*. This applies not only to file management but also to process and signal management.
- Multiplexed Processes:
  To extend UNIX services – as long as it makes sense – with services provided by the underlying CHORUS Nucleus, e.g., multi-threaded UNIX processes.

Orthogonality

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

Compatibility

- for application programs:
  On a given machine, to be compatible at the executable code level with a given standard UNIX system (e.g., System V Release 3.2 on a PC/AT-386), to ensure complete user software portability.
- for device drivers:
  To be able to adapt a UNIX driver into a UNIX Server on CHORUS with a minimum effort.
- regarding performances:
  To provide the same services about as fast as a given UNIX system on the same machine architecture (i.e., the one chosen for binary compatibility).

3.2 The UNIX Subsystem Architecture

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

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

Several types of servers may be distinguished within a typical UNIX subsystem: Process Managers (PM), File Managers (FM), Pipe Managers (PIM), Device Managers (DM) and User Defined Servers (UDS) (Figure 6).

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

3.3 Structure of a UNIX Server

Servers = Actors

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

Mono or Multi-Threaded

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

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

Accessible via Ports or Traps

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

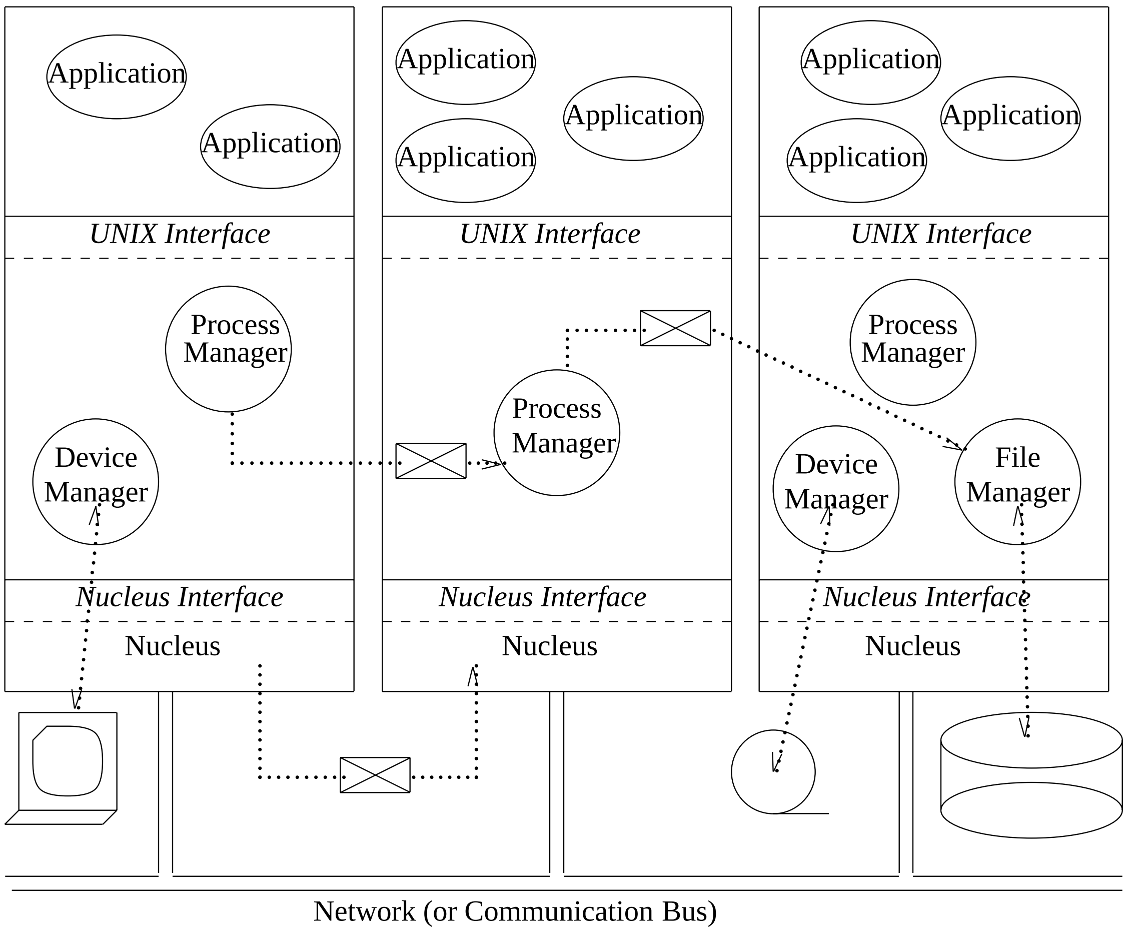

Figure 6. – UNIX with the CHORUS Technology

service
independently of the server which will actually provide it.

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

Implementation of a Service

A service (e.g., `open(2)`) is realized partly by a ‘‘stub’’ which executes, in ‘‘system mode’’ – behind a trap –, in the context of the calling client process. This routine manages the context of the process (see Process Manager) and when needed, invokes the appropriate Subsystem server via the CHORUS IPC.

Drivers

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

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

3.4 Process Manager

On each ‘‘UNIX site’’, a Process Manager maps UNIX process abstractions into CHORUS concepts (actor, thread, regions, etc.). It implements all of the UNIX process semantics including process creation, context consistency and inheritance, and process signaling.

Process Managers interact with their environment through clearly defined interfaces:

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

Process Managers are the unique entry points from a user process to UNIX services (other UNIX servers are not accessed directly by user processes). For binary compatibility reasons, these services are accessed through traps. The Process Manager uses the CHORUS Nucleus facility to attach routines to these traps. Those routines either call other Process Manager routines to satisfy requests related to process and signal management (`fork`, `exec`, `kill`, etc.) or invoke via RPC File Managers, Pipe Managers or Device Managers to handle other requests.

Process Managers cooperate to implement remote execution and remote signaling:

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

3.5 File Manager

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

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

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

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

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

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

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

Naming Extension

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

Symbolic Ports (new UNIX file type) can be created in the UNIX file tree. They associate a file name to a port Unique Identifier (this is very similar to UNIX device designation). When such a name is found during the analysis of a pathname, the corresponding request is forwarded to the port to which the Unique Identifier is associated – marked with the current status of the analysis (see section 3.10.).

Servers can be designated by such symbolic names. In particular, this functionality is used to interconnect file systems and provide a global name space (Figure 7). Transparent access to remote files is provided through symbolic links (a‘ la B.S.D.).

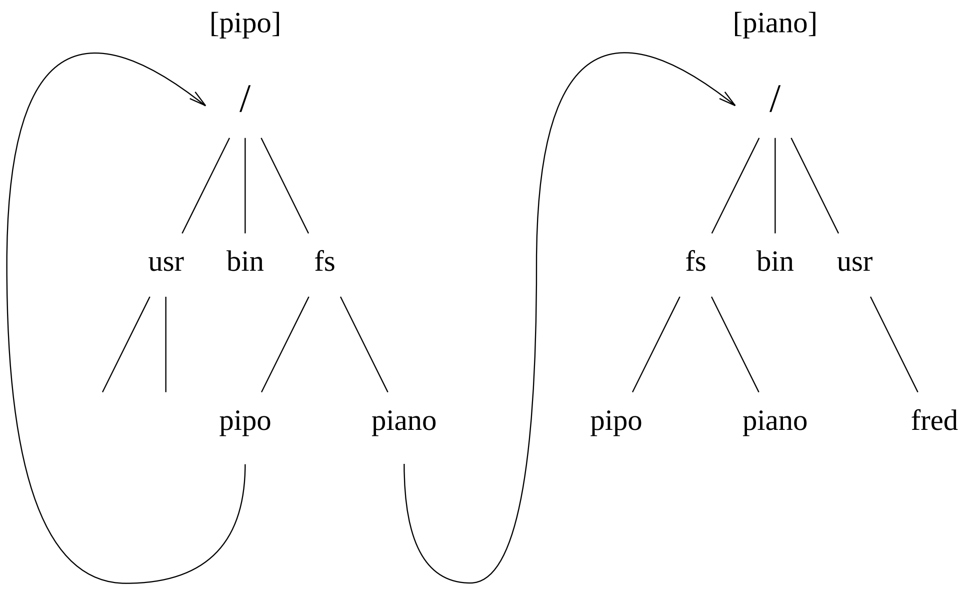

Figure 7. – File Trees Interconnection

3.6 Pipe Manager

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

3.7 Device Manager

Devices such as tty’s and pseudo-tty’s, bitmaps, tapes, network interfaces are managed by Device Managers. Needed/not used drivers can thus be loaded/unloaded dynamically (i.e., while the system is running). Software configurations can be adjusted to accommodate the use of the system or the local hardware configuration. For example, a bitmap server may be present on a diskless

station;
a File Manager may not be.

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

3.8 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 to provide a powerful and convenient platform for the experimentation of system servers: new file management strategies, fault-tolerant servers, etc., can be developed and tested just like user level utilities, without disturbing a running system.

3.9 Mapping Decisions

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

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

In addition, the Process Manager attaches one *control port* and one *control thread* to each actor implementing a UNIX process. The control port and the control thread are not visible to the user of that process. Control threads executing within process contexts have two main properties:

- they share the process address space and can easily access and modify the core image of the process (stack manipulations on signal reception, text and data access during debugging, etc.).
- they are ready to handle asynchronous events received by the process (mainly signals). These events are implemented as CHORUS messages received on the control port (Figure 8).

Even if standard UNIX processes are mono-thread, the UNIX subsystem on CHORUS has been extended to allow multiprogramming within a process. This facility translates at the UNIX level the CHORUS multi-thread facilities useful for implementing multiplexed servers. The semantics of such UNIX threads (called *U_threads*) differ slightly from ‘‘pure’’ CHORUS threads, in particular regarding signal handling, time accounting, system call multiplexing, thus their slightly different name.

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

— signal processing is done on a per thread basis,

— blocking system calls (`pause(2)`, `wait(2)`, `read(2)`, etc.) may be multiplexed within a given process,

— the UNIX system context attached to one process has been split into two system contexts: one process context (`Proc`) (Table 1) and one U_thread context (`U_thread`) (Table 2).

The two system contexts `Proc` and `U_thread` are maintained by the Process Manager present on the current process execution site. These contexts are accessed neither by the CHORUS

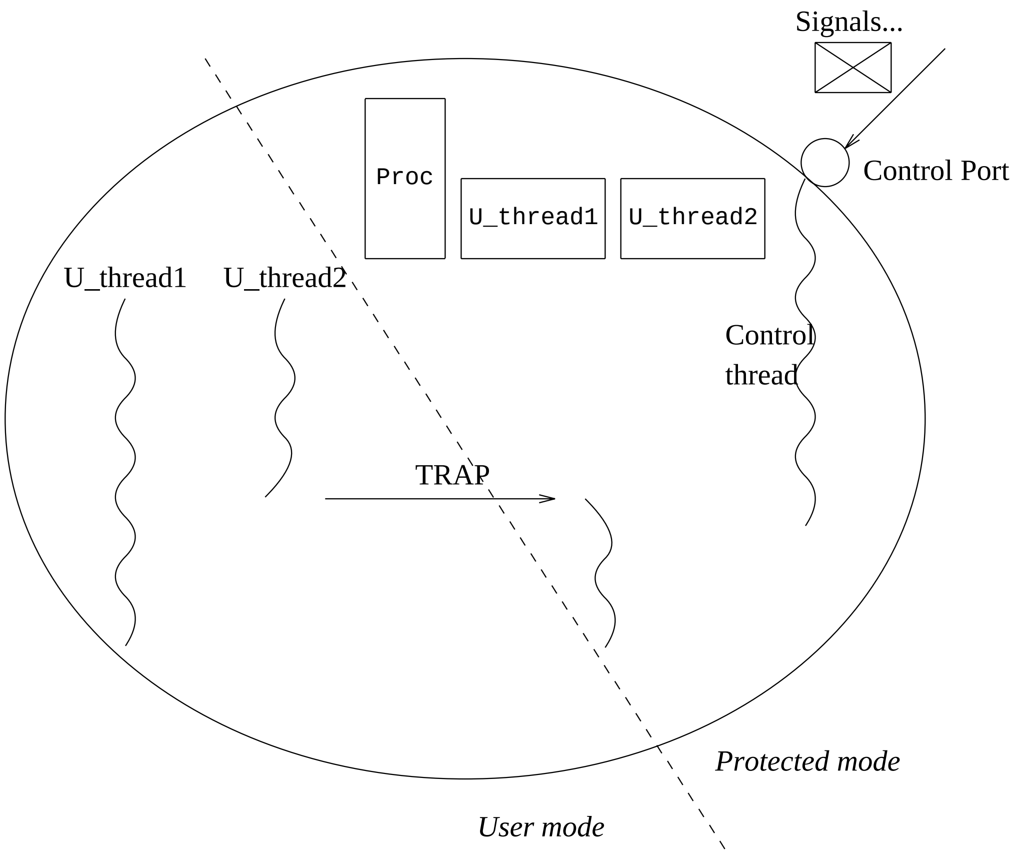

Figure 8. – UNIX Process as a CHORUS Actor

Nucleus
nor by other system servers. On the other hand the UNIX subsystem has no visibility of the internal Nucleus structures associated with actors and threads, the only way to access them is through Nucleus system calls (this is essential for allowing different subsystems to co-reside on top of the same CHORUS Nucleus).

3.9.1 Process Identifiers

The Unique Identifiers (UI) managed by the CHORUS Nucleus are typed (actors, ports, etc.) and a few types are reserved for subsystem usage. One type of UI is used by the Process Manager for process designation so that each process is uniquely designated by one UI. This enables Process Managers to take advantage of Nucleus localization facilities when they look for one particular process (signaling, debugging,...).

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

— the site where the process has been created.

— the value of a per site counter incremented at each process creation.

Process Managers convert PID's into process UI's (and conversely). They use one or the other depending of the needs (CHORUS IPC's or system call interface).

Table 1. – Process Context

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

3.9.2 Process execution site

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

3.9.3 Process environment known by its set of Ports

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

Table 2. – U_Thread Context

U-thread Context	
Thread	*Thread implementing the U_thread (thread descriptor, priority,...)*
Proc	*Owner Process*
Sig	*Signal Context (signal handlers,...)*
SysCall	*System Call Context (system call arguments,...)*
Context	*Machine execution Context*

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

3.9.3.1 Resources and capabilities

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

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

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

Thus, the following requests on that open file (`seek(2), close(2)`...) can be translated into a message and sent directly to the appropriate server.

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

3.9.3.2 Processes and system ports

Process Managers attach some ‘‘system’’ Ports (to distinguish from the ports created by the process) to each process in order to implement some UNIX services regardless of the process locations:

- Each process is created with a *Control Port* on which the control thread receives messages. The parent process control port UI is also stored in the process context. When a process exits, a message filled with all needed information (exit status, elapsed time, etc.) is sent to the parent’s control port. When received by the parent control thread, this information is stored in

the
parent `Proc` context until the parent process attempts to obtain them – `wait(2)` UNIX system call.

- Signals are implemented as messages and Process Managers, acting as intermediaries for process localization, forward these messages to the control port of the target process(es) – these messages will be processed by the process control thread.
- When a debugging session starts, the debugged process creates a debug port and sends this port name to the debugger. All interactions between the debugger and debugged processes rely on CHORUS IPC between the debug port of the debugged process and the control port of the debugger. Because it is based on the CHORUS IPC, UNIX debugging functionality is distributed – debugged and debugger processes can be on different sites.

3.10 Two Examples

3.10.1 File access

Current and root directories are represented by capabilities in the UNIX context of a process. When an `open(2)` request is issued, the open routine of the Process Manager 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.

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

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

Afterward, the following requests on that open file (`seek(2)`, `close(2)`, etc.) will be sent directly to the File Manager on ‘‘`piano`’’ avoiding any indirection through the File Manager responsible of the root directory of the process (Figure 9).

3.10.2 Remote Exec

This description of the remote `exec` algorithm will illustrate all the interactions between the UNIX subsystem servers and a 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. invoke by RPC the File Manager to translate the binary file pathname into two capabilities, used later on to map text and data into the process address space. Depending on the pathname, the File Manager of the root directory or of the current directory is invoked.
   b. test child execution site (in this example, child execution site is different from current execution site),

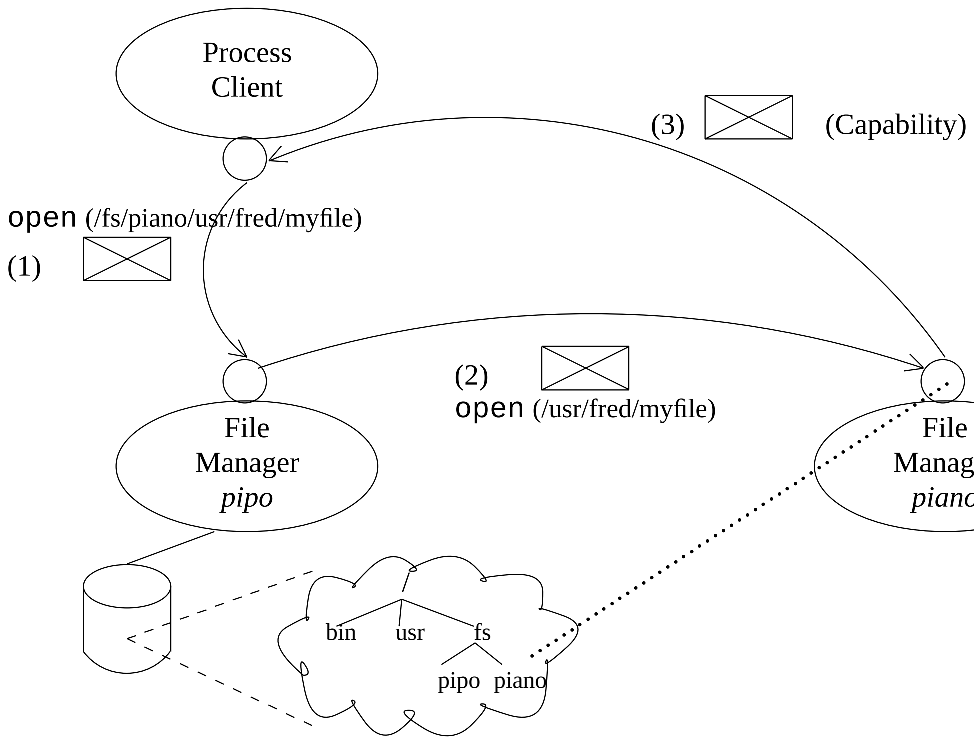

Figure 9. – File Access

c. prepare a message with :

— `Proc` and `U_thread` context of calling U_thread,

— arguments and environments given as `exec` parameters,

— all information returned by the File Manager and which characterize the binary file.

d. perform an RPC to the Process Manager of the target creation site – actually addressed via a Port group.

2. The Process Manager of the remote creation site receives the request and one of its threads will process it. This Process Manager thread initializes a new `Proc` context for the migrating process (much information as PID's, elapsed time, etc. are just copied from the request message) and creates new CHORUS entities which implement the process: actor, control thread, control port, memory regions, etc.

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

3. The process control thread receives the forwarded message and follows the normal UNIX process initialization:
   - arguments and environment are installed in the process address space.
   - close messages are sent to appropriate File Managers, Pipe Managers and Device Managers to close any close-on-exec open files.
   - one U_thread is created which will start executing the new program (even if before `exec` the process was multi-threaded, the new process is always mono-threaded after). The signal context of the created U_thread is set up with signal context of calling

U_thread
present in the request message.

- a reply message is sent back to the U_thread which had invoked the `exec(2)` system call and has blocked in the RPC awaiting the reply message.

4. The calling thread receives the reply message, frees the `Proc` and `U_thread` contexts of its process and removes the actor implementing the process (on actor destruction, the CHORUS Nucleus manages to remove all CHORUS entities attached to this actor : ports, threads, regions, etc.).

4 Extending UNIX

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

4.1 IPC

UNIX processes running on CHORUS can now communicate with other processes – and bare CHORUS Actors or entities from other Subsystems – using the CHORUS IPC mechanisms, which are better adapted to distribution and reconfiguration than existing standard UNIX IPC. In particular, processes are able to :

— create and manipulate CHORUS ports,

— send and receive messages,

— issue remote procedure calls.

4.2 Real-Time

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

— The ability to dynamically connect handlers to hardware interrupts. This facility is already used by UNIX Device Managers.

— The benefit of the priority based preemptive scheduling provided by the CHORUS Nucleus.

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

— process entirely the interrupt in its handler or,

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

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

4.3 UNIX Commands Extensions

Several UNIX commands (such as `ls(1)`, `find(1)`, `ln(1)`, etc.) have been extended also in particular to cope with the new ‘‘Port’’ file type. The `shell` as well has been extended to give access to some CHORUS functionalities:

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

— creating symbolic ports,

— sending, receiving messages between `shell`'s.

For example, one can synchronize two `shell`'s which may be running on different machines in the following way:

```
# create a Port on the current shell
# give it the symbolic name "shport"
# into the ‘‘/usr/fred’’ directory:

bind /usr/fred/shport

# wait for a message on that port:

receive /usr/fred/shport > msg_in

# reply to the sender:

reply < msg_out
```

```
# send a message to the other shell:

send "message" /usr/fred/shport
```

5 Conclusion

The CHORUS technology for building new generations of distributed operating systems is the result of a number of years of research and experimentation that lead to the current CHORUS-V3 architecture. It has been used to implement a distributed UNIX system compatible at the executable code level with current UNIX standards and providing support for distributed as well as real-time applications.

CHORUS V3 has been implemented on different computer architectures, based on different microprocessor families (MC680x0 and Intel 386) and different bus architectures and memory management units. It is mostly written in C++ (the rest being in C).

The current UNIX implementation is System V. A BSD UNIX subsystem is being started.

In line with the current trends to re-engineer UNIX so that it may still be an adequate answer to the needs of the next decade, the CHORUS technology represents a sound and validated basis.

6 Acknowledgements

Jean-Pierre Ansart, Philippe Brun, Hugo Coyotte, Jean-Jacques Germond, Pierre Lebe’e, Corinne Lucas, Fre’de’ric Lung, Denis Metral-Charvet, Bruno Pillard, Didier Poirot and Franc/ois Saint-Lu 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 and then allowing its development and transformation into an industrial venture made all this possible.

This paper would not have existed without Claude Kaiser who contributed extensively to the clarification of the ideas behind CHORUS as well as on the means to expressing them.

7 References