A New Look at Microkernel-Based UNIX Operating Systems: Lessons in Performance and Compatibility

Allan Bricker, Michel Gien, Marc Guillemont, Jim Lipkis, Douglas Orr, Marc Rozier
Chorus systèmes

Abstract

An important trend in operating system development is the restructuring of the traditional monolithic operating system kernel into independent servers running on top of a minimal nucleus or "microkernel". This approach arises out of the need for modularity and flexibility in managing the ever-growing complexity caused by the introduction of new functions and new architectures. In particular, it provides a solid architectural basis for distribution, fault tolerance, and security. Microkernel-based operating systems have been a focus of research for a number of years, and are now beginning to play a role in commercial UNIX® systems.

The ultimate feasibility of this attractive approach is not yet widely recognised, however. A primary concern is efficiency: can a microkernel-based modular operating system provide performance comparable to that of a monolithic kernel when running on comparable architectures? The elegance and flexibility of the client-server model may exact a cost in message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly on portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases, binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

The Chorus team has spent the past six years studying and experimenting with UNIX "kernelisation" as an aspect of its work in modular distributed and real-time systems. In this paper we examine aspects of the current CHORUS® system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering.

Appeared in the Proceedings of the EurOpen Spring'91 Conference, Tromsoe, Norway, May 20-24, 1991.

® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

® CHORUS is a registered trademark of Chorus systèmes.

1. Microkernel Architectures

A recent trend in operating system development consists of structuring the operating system as a modular set of system servers which sit on top of a minimal microkernel, rather than using the traditional monolithic structure. This new approach promises to help meet system and platform builders' needs for a sophisticated operating system development environment that can cope with growing complexity, new architectures, and changing market conditions. In this operating system architecture, the microkernel provides system servers with generic services, such as processor scheduling and memory management, independent of a specific operating system. The microkernel also provides a simple inter-process communication (IPC) facility that allows system servers to call each other and exchange data regardless of where they are executed, in a multiprocessor, multicomputer, or network configuration.

This combination of primitive services forms a standard base which in turn supports the implementation of functions that are specific to a particular operating system or environment. These system-specific functions can then be configured, as appropriate, into system servers managing the other physical and logical resources of a computer system, such as files, devices and high-level communication services. We refer to such a set of system servers as a *subsystem*. Real-time systems tend to be built along similar lines, with a simple, generic executive supporting application-specific real-time tasks.

1.1 UNIX and Microkernels

UNIX introduced the concept of a standard, hardware-independent operating system, whose portability allowed platform builders to reduce their time to market by obviating the need to develop proprietary operating systems for each new platform.

However, as more function and flexibility is continually demanded, it is unavoidable that today's versions become increasingly more complex. For example, UNIX is being extended with facilities for real-time applications and on-line transaction processing. Even more fundamental is the move toward distributed systems. It is desirable in today's computing environments that new hardware and software resources, such as specialised servers and applications, be integrated into a single system, distributed over a network. The range of communication media commonly encountered includes shared memory, buses, high-speed networks, local-area networks, and wide-area networks. This trend to integrate new hardware and software components will become fundamental as collective computing environments emerge.

To support the addition of function to UNIX and its migration to distributed environments, it is desirable to map UNIX onto a microkernel architecture, where machine dependencies may be isolated from unrelated abstractions and facilities for distribution may be incorporated at a very low level.

The attempt to reorganise UNIX to work within a microkernel framework poses problems, however, if the resultant system is to behave exactly as a traditional UNIX implementation. A primary concern is efficiency: a microkernel-based modular operating system must provide performance comparable to that of a monolithic kernel. The elegance and flexibility of the client-server model may exact a cost in message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly upon portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

There is work in progress on a number of fronts to emulate UNIX on top of a microkernel architecture, including the Mach [Gol90], V [Che90], and Amoeba [Tan90] projects. Plan 9 from Bell Labs [Pik91] is a distributed UNIX-like system based on the "minimalist" approach. CHORUS versions V2 and V3 represent the work we have done to solve the problems of compatibility and efficiency.

1.2 The CHORUS Microkernel Technology

The Chorus team has spent the past six years studying and experimenting with UNIX "kernelisation" as an aspect of its work in modular, distributed and real-time systems. The first implementation of a UNIX-compatible microkernel-based system was developed during 1984 through 1986 as a research project at INRIA. Among the goals of this project were to explore the feasibility of shifting as much function as possible out of the kernel and to demonstrate that UNIX could be implemented as a set of modules that did not share memory. In late 1986, an effort to create a new version, based on an entirely rewritten CHORUS nucleus, was launched at Chorus systèmes. The current version maintains many of the goals of its predecessor and adds some new ones, including real-time support and – not incidentally – commercial viability. A UNIX subsystem compatible with System V Release 3.2 is currently available, with System V Release 4.0 and 4.4BSD systems under development. The System V Release 3.2 implementation performs comparably with well-established monolithic-kernel systems on the same hardware, and better in some respects. As a testament to its commercial viability, the system has been adopted for use in commercial products ranging from X terminals and telecommunication systems to mainframe UNIX machines.

In this paper we examine aspects of the current CHORUS system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering.

In section 2, we review the previous CHORUS version. Section 3 evaluates the previous version and discusses how the lessons learned from its implementation led to the main design decisions for the current version. The subsequent sections focus on specific aspects of the current design.

2. CHORUS V2 Overview

The CHORUS project, while at INRIA, began researching distributed operating systems with CHORUS V0 and V1. These versions proved the viability of a modular, message-based distributed operating system, examined its potential performance, and explored its impact on distributed applications programming.

Based on this experience, CHORUS V2 [Arm86, Roz87] was developed. It represented the first intrusion of UNIX into the peaceful CHORUS landscape. The goals of this third implementation of CHORUS were:

1. To add UNIX emulation to the distributed system technology of CHORUS V1;
2. To explore the outer limits of "kernelisation"; demonstrate the feasibility of a UNIX implementation with a minimal kernel and semi-autonomous servers;
3. To explore the distribution of UNIX services;
4. And to integrate support for a distributed environment into the UNIX interface.

Since its birth, the CHORUS architecture has always consisted of a modular set of servers running on top of a microkernel (the nucleus) which included all of the necessary support for distribution.

The basic execution entities supported by the V2 nucleus were mono-threaded *actors* running in user mode and isolated in protected address spaces. Execution of actors consisted of a sequence of "processing-steps" which mimicked atomic transactions: ports represented operations to be performed; messages would trigger their invocation and provide arguments. The execution of remote operations were synchronised at explicit "commit" points. An ever-present concern in the design of CHORUS was that fault-tolerance and distribution are tightly coupled; hardware redundancy both increases the probability of faults and gives a better chance to recover from these faults.

Communication in CHORUS V2 was, as in many current systems, based upon the exchange of messages through *ports*. Ports were attached to actors, and had the ability to migrate from one actor to another. Furthermore, ports could be gathered into *port groups*, which allowed message broadcasting as well as functional addressing. For example, a message could be directed to all members of a port group or to a single member port which resided on a specified site. The port group mechanism provided a flexible set of client-server mapping semantics including dynamic reconfiguration of servers.

Ports, port groups, and actors were given global unique names, constructed in a distributed fashion by each nucleus for use only by the nucleus and system servers. Private, context-dependent names were exported to user actors. These *port descriptors* were inherited in the same fashion as UNIX file descriptors.

2.1 UNIX

On top of this architecture, a full UNIX System V was built.

In V2, the whole of UNIX was split into three servers: a *process manager*, dedicated to process management, a *file manager* for block device and file system management, and a *device manager* for character device management. In addition, the nucleus was complemented with two servers, one which managed ports and port groups, and another which managed remote communications (see Figure 1). UNIX network facilities (sockets) were not implemented at this time.

A UNIX process was implemented as a CHORUS actor. All interactions of the process with its environment, i.e. all system calls, were performed as exchanges of messages between the process and system servers. Signals were also implemented as messages.

This "modularisation" impacted UNIX in the following ways:

1. UNIX data structures were split between the nucleus and several servers. Splitting the data structures, rather than replicating them, was done to avoid consistency problems. Messages between these servers contained the information managed by one server and required by another in order to provide its service. Careful thought was given to how UNIX data structures were split between servers to minimise communication costs.

2. Most UNIX objects, files in particular, were designated by network-wide capabilities which could be exchanged freely between subsystem servers and sites. The context of a process contained a set of capabilities representing the objects accessed by the process.

As many of the UNIX system calls as possible were implemented by a process-level library. The process context was stored in process-specific library data at a fixed, read-only location within the process address space. The library invoked the servers, when necessary, using a

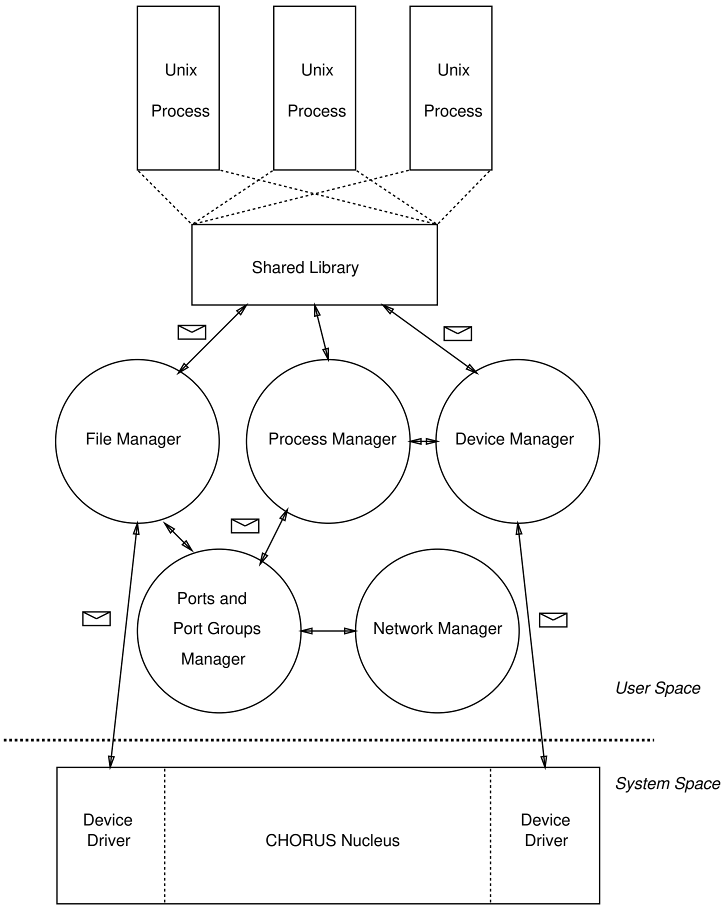

Figure 1. CHORUS-V2 Architecture

remote procedure call (RPC) facility. For example, the process manager was invoked to handle a `fork`(2) system call and the file manager for a `read`(2) system call on a file.

This library offered only source-level compatibility with UNIX, but was acceptable because binary compatibility was not a project goal. The library resided at a predefined user virtual address in a write-protected area. Library data holding the process context information was not completely secure from malicious or unintentional modification by the user. Thus, errant programs could experience new, unexpected error behaviour. In addition, programs that depended upon the standard UNIX address space layout could cease to function because of the additional address space contents.

2.2 Extended UNIX Services

CHORUS V2 extended UNIX services in two ways:

- by allowing their distribution while retaining their original interface (e.g. remote process creation and remote file access).
- by providing access to new services without breaking existing UNIX semantics (e.g. CHORUS IPC).

2.2.1 Distribution of UNIX Services

Access to files and processes extended naturally to the remote case due to the modularity of CHORUS's UNIX and its inherent protocols. Files and processes, whether local or remote, were manipulated using CHORUS IPC through the use of location-transparent capabilities.

In addition, CHORUS V2 extended UNIX file semantics with *port nodes*. A port node was an entry in the file system which had a CHORUS port associated with it. When a port node was encountered during path-name analysis, a message containing the remainder of the path to be analysed was sent to the associated port. Port nodes were used to automatically interconnect file trees.

For processes, new protocols between process managers were developed in order to distribute `fork` and `exec` operations. Remote `fork` and `exec` were facilitated because:

- the management of a process context was not distributed; each process context was managed entirely by only one system server (the process manager),
- a process context contained only global references to resources (capabilities).

Therefore, creating a remote process could be done almost entirely by transferring the process context from one process manager to another.

Since signals were implemented as messages, their distribution was trivial due to the location transparency of CHORUS IPC.

2.2.2 Introduction of New Services

CHORUS IPC was introduced at user-level. Its UNIX interface was designed in the standard UNIX style:

1. Ports and port groups were known, from within processes, by local identifiers. Access to a port was controlled in a fashion analogous to the access to a file.
2. Ports and port groups were protected in a similar fashion to files (with *uids* and *gids*).
3. Port and port group access rights were inherited on `fork` and `exec` exactly as are file descriptors.

3. Analysis of CHORUS V2

Experience developing and using CHORUS V2 gave us valuable insight into the basic operating system services that a microkernel must provide to implement a rich operating system environment such as UNIX. CHORUS V2 was our third reimplementation of the CHORUS nucleus, but represented our first attempt at integrating an existing, complex operating system interface with microkernel technology. This research exercise was not without faults. However, it demonstrated that we did a number of things correctly. The CHORUS V2 basic IPC abstractions – location transparency, untyped messages, asynchronous and RPC protocols, ports, and port groups – have proven to be well suited to the implementation of distributed operating systems and applications. These abstractions have been entirely retained for CHORUS V3; only their interface has been enriched to make their use more efficient.

The basic modular architecture of the UNIX subsystem has also been retained in the implementation of CHORUS V3 UNIX subsystems. Some new servers, such as a BSD *socket manager*, have been added to provide new function that was not included in CHORUS V2.

Version 3 of the CHORUS nucleus has been completely redesigned and reimplemented around a new set of project goals. These goals were put in place as a direct result of our experience implementing our first distributed UNIX system.

In the following subsections we briefly state our new goals and then explain how these new goals affected the design of CHORUS V3.

3.1 CHORUS V3 Goals

The design of CHORUS V3 system [Arm89, Arm90, Her88, Roz88] has been strongly influenced by a new major goal: to design a microkernel technology suitable for the implementation of commercial operating systems. CHORUS V2 was a UNIX-compatible distributed operating system. The CHORUS V3 microkernel is able to support operating system standards while meeting the new needs of commercial systems builders.

These new goals determined new guidelines for the design of the CHORUS V3 technology:

- **Portability:** the CHORUS V3 microkernel must be highly portable to many machine architectures. In particular, this guideline motivated the design of an architecture-independent memory management system [Abr89], taking the place of the hardware-specific CHORUS V2 memory management.
- **Generality:** the CHORUS V3 microkernel must provide a set of functions that are sufficiently generic to allow the implementation of many different sets of operating system semantics; some UNIX-related features had to be removed from the CHORUS V2 nucleus. The nucleus must maintain its simplicity and efficiency for users or subsystems which do not require high level services.
- **Compatibility:** UNIX source compatibility in CHORUS V2 had to be extended to binary compatibility in V3, both for user applications and device drivers. In particular, the CHORUS V3 nucleus had to provide tools to allow subsystems to build binary compatible interfaces.
- **Real-time:** process control and telecommunication systems comprise important targets for distributed systems. In these areas, the responsiveness of the system is of prime importance. The CHORUS V3 nucleus is, first and foremost, a distributed real-time executive. The real-time features may be used by any subsystem, allowing for example, a UNIX subsystem to be naturally extended to be suitable for real-time applications needs.
- **Performance:** for commercial viability, good performance is essential in an operating system. While offering the base for building modular, well-structured operating systems, the nucleus interface must allow these operating systems to reach at least the same performance as conventional, monolithic, implementations.

These new goals forced us to reconsider CHORUS V2 design choices. In most cases, the architectural elements were retained in CHORUS V3; only their interface evolved. Whenever possible, the V3 interface reflects our desire to leave it to the subsystem designer to negotiate the tradeoffs between simplicity and efficiency, on the one hand, and more sophisticated function, on the other.

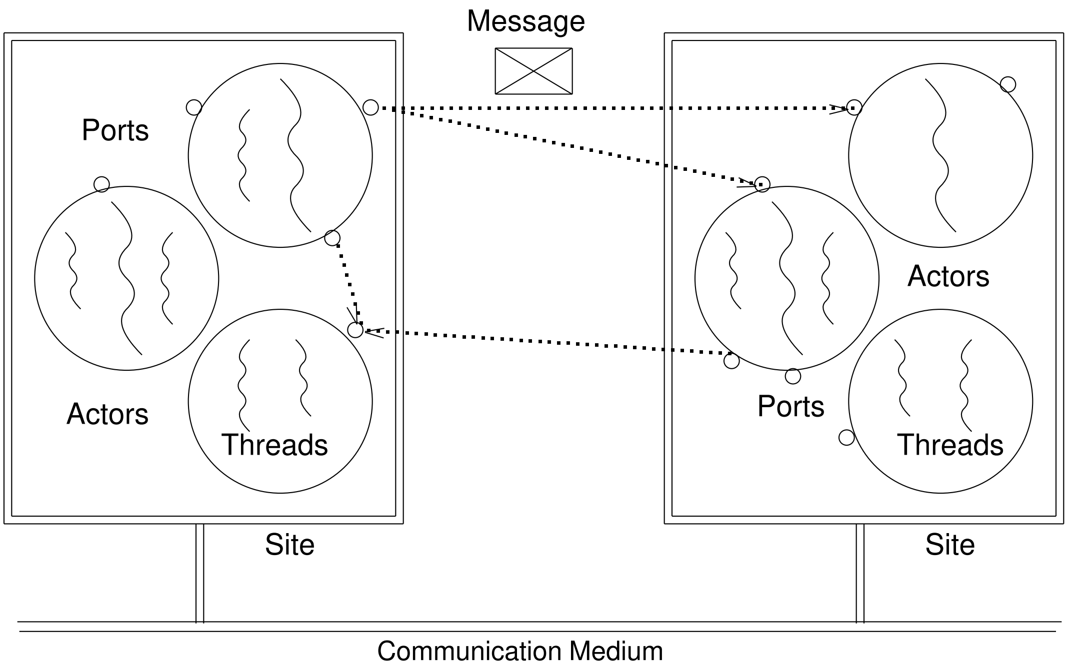

Figure 2. CHORUS V3 Nucleus Abstractions

3.2 CHORUS Processing Model

Problems arose with the CHORUS V2 processing model when UNIX signals were first implemented. To treat asynchronous signals in V2 mono-threaded actors, it was necessary to introduce the concept of priorities within messages to expedite the invocation of a signaling operation. Even so, the priorities went into effect only at fixed synchronisation points, making it impossible to perfectly emulate UNIX signal behaviour. Further work has shown that signals are one of the major stumbling blocks for building fault tolerant UNIX systems.

> **Lesson:** *We found the processing-step model of computation to be a poor fit with the asynchronous signal model of exception handling. In order to provide full UNIX emulation, a more general computational model was necessary for CHORUS V3.*

The solution to this problem gave rise to the V3 multi-threaded processing model. A CHORUS V3 actor is merely a resource container, offering, in particular, an address space in which multiple threads may execute. Threads are scheduled as independent entities, allowing real parallelism on a multiprocessor architecture. In addition, multiple threads allow the simplification of the control structure of server-based applications. New nucleus services, such as thread execution control and synchronisation have been introduced.

3.3 CHORUS Inter-Process Communication

As a consequence of the change to the basic processing model, the inter-process communication model also evolved. In the V2 processing-step model, IPC and execution were tightly bound, yielding a mechanism that resembled atomic transactions.

This tight binding of communication to execution did not necessarily make sense in a multi-threaded CHORUS V3 system. Thus, the atomic transactions of V2 have been replaced, in V3, by the remote procedure call paradigm and has since evolved into an extremely efficient lightweight RPC protocol.

One aspect of the IPC mechanism that has not changed in CHORUS V3 is that messages remain untyped. The CHORUS IPC mechanism is simple and efficient when communicating among homogeneous sites. When communicating between heterogeneous sites, higher-level protocols are used, as needed. A guideline in the design of CHORUS V2, retained in V3, was to allow the construction of simple and efficient applications without forcing them to pay a penalty for sophisticated mechanisms which were required only by specific classes of programs.

3.4 CHORUS Ports

A number of enhancements concerning CHORUS ports have been made to provide more generality and efficiency in the most common cases.

3.4.1 Port Naming

Recall that in V2 context-dependent port names were exported to the user-level while global port names were used by the nucleus and system servers. The user-level context-dependent port names of V2 were intended to provide security and ease of use. It was difficult, however, for applications to exchange port names, since it required intervention by the nucleus and posed bootstrapping problems. As a result, context-dependent names were inconvenient for distributed applications, such as name servers. In addition, many applications had no need of the added security the context-dependent names provided.

> **Lesson:** *CHORUS V3 makes global names of ports and port groups (unique identifiers) visible to the user, discarding the UNIX-like CHORUS V2 contextual naming scheme. Contextual identifiers turned out not to be an effective paradigm.*

The first consequence of using unique identifiers is simplicity: port and port group names may be freely exchanged by nucleus users, avoiding the need for the nucleus to maintain complex actor context. The second consequence is a lower level of protection: the CHORUS V3 philosophy is to provide subsystems with the means for implementing their own level and style of protection rather than enforcing protection directly in the microkernel. For example, if the security of V2 context-dependent names is desired, a subsystem can easily and efficiently export a protected name-space server. V3 unique identifiers have proven to be key to providing distributed UNIX services in an efficient manner.

3.4.2 Port Implementation

A goal of the V2 project was to determine what were the minimal set of functions that a microkernel should have in order to support a robust base of computing. To that end, the management of ports and port groups was put into a server, external to the nucleus. Providing the ability to replace a portion of the IPC did not prove to be useful, however, since IPC was a fundamental and critical element of all nucleus operations. Maintaining it in a separate server rendered it more expensive to use.

> **Lesson:** *We found that actors, ports, and port groups are basic nucleus abstractions. Splitting their management did not provide significant benefit, but did impact system performance. Actor, port, and port group management has been moved back into the nucleus for V3.*

3.4.3 UNIX Port Extensions

When extending the UNIX interface to give access to CHORUS IPC, we maintained normal UNIX-style semantics. Employing the same form as the UNIX file descriptor for port descriptors was intended to provide uniformity of model. The semantics of ports were sufficiently different from the semantics of files to negate this advantage. In operations such as `fork`, for example, it did not make sense to share port descriptors in the same fashion as file descriptors. Attempting to force ports into the UNIX model resulted in confusion.

> **Lesson:** *A user-level IPC interface was important, but giving it UNIX semantics was cumbersome and unnecessary. This lesson is an example of a larger principle; the nucleus abstractions should be primitive and generally applicable – they should not be coerced into the framework of a specific operating system.*

V3 avoids this issue by, as previously mentioned, exporting global names. Since the V3 nucleus no longer manages the sharing of global port and port group names, it is up to the individual subsystem servers to do so. In particular, if counting the number of references to a given port is important to a subsystem, it is the subsystem itself that must maintain the reference count. On the other hand, a subsystem that has no need for reference counting is not penalised by the nucleus.

Using V2 port nodes to interconnect file systems was a simple, but extremely powerful, extension to UNIX. Since all access to files was via CHORUS messages, port nodes provided network transparent access to regular files as well as to device nodes. They also, however, introduced a new file type into the file system. This caused many system utilities, such as `ls` and `find`, to not function properly. Thus, all such utilities had to be modified to take the new file type into account.

Port nodes have been maintained in CHORUS V3 (however, they are now called "*symbolic ports*"). In future CHORUS UNIX systems, the file type "symbolic port" may be eliminated by inserting the port into the file system hierarchy using the `mount` system call. These "*port mount points*" would carry the same semantics as a normal mounted file system.

3.5 Virtual Memory

The virtual memory subsystem has undergone significant change. The machine dependent virtual memory system of CHORUS V2 has been replaced, in V3, by a highly portable VM system. The VM abstractions presented by the V3 nucleus include "segments" and "regions." Segments encapsulate data within a CHORUS system and typically represent some form of backing store, such as a swap area on a disk. A region is a contiguous range of virtual addresses within an actor that map a portion of a segment into its address space. Requests to read or to modify data within a region are converted by the virtual memory system into read or modify requests within the segment. "External Mappers" interact with the virtual memory system using a nucleus-to-Mapper protocol to manage data represented by segments. Mappers also provide the synchronisation needed to implement distributed shared memory. For more details on the CHORUS V3 virtual memory system, see [Abr89].

3.6 Actor Context

CHORUS V2 was built around a "pure" message-passing model, in which strict protection was incorporated at the lowest level; all servers were implemented in protected user address spaces. This distinct separation enforced a clean, modular design of a subsystem. However, it also led to several problems:

- A UNIX subsystem based on CHORUS V2 required the use of user-level system call stubs and altered the memory layout of a process and, therefore, could never provide 100% binary compatibility;
- All device drivers were required to reside within the nucleus;
- Context switching expense was prohibitively high.

The most fundamental enhancement made between CHORUS V2 and V3 was the introduction of the *supervisor actor*. Supervisor actors share the supervisor address space and their threads execute in a privileged machine state. Although they reside within the supervisor address space, supervisor actors are truly separate entities; they are compiled, link edited, and loaded independently of the nucleus and of each other.

The introduction of supervisor actors creates several opportunities for system enhancement in the areas of compatibility and performance. Section 4 discusses the ramifications of supervisor actors in-depth.

3.7 UNIX Subsystem

As a consequence of these nucleus evolutions, the UNIX subsystem implementation has also evolved. In particular, full UNIX binary compatibility has been achieved. Internally, the UNIX subsystem makes use of new nucleus services, such as multi-threading and supervisor actors. The CHORUS V2 user-level UNIX system-call library has been moved inside the process manager and is now invoked directly by system-call traps.

Experience with the decomposition of UNIX System V for V2 showed, not surprisingly, that performing this modularisation is difficult. Care must be taken to decompose the data structures and function along meaningful boundaries. Performing this decomposition is an iterative process. The system is first decomposed along broad functional lines. The data structures are then split accordingly, possibly impacting the functional decomposition.

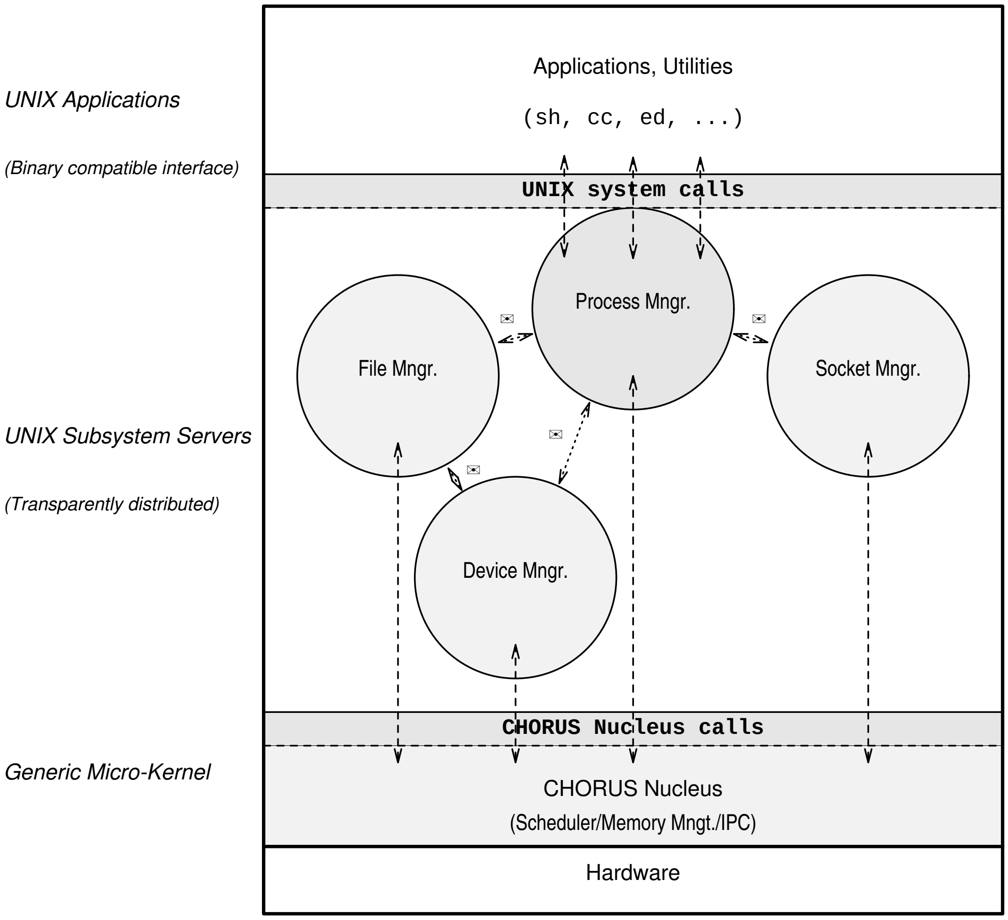

Figure 3. CHORUS/MiX-V3 Architecture

4. Evolution in Nucleus Support for Subsystems: Supervisor Actors

Supervisor actors, as mentioned above, share the supervisor address space and their threads execute in a privileged machine state, usually implying, among other things, the ability to execute privileged instructions. Otherwise, supervisor actors are fundamentally similar to regular user actors. They may create multiple ports and threads, and their threads access the same nucleus interface. Any user program can be run as a supervisor actor, and any supervisor actor which does not make use of privileged instructions or *connected handlers* (see below) can be run as a user actor. In both cases a recompilation of the program is not needed. Although they share the supervisor address space, supervisor actors are paged just as user actors and may be dynamically loaded and deleted.

Supervisor actors, alone, are granted direct access to the hardware event facilities. Using a standard nucleus interface, any supervisor actor may dynamically establish a handler for any particular hardware interrupt, system call trap, or program exception. A connected handler executes as an ordinary subroutine, called directly from the corresponding low-level handler in the nucleus. Several arguments are passed to it, including the interrupt/trap/exception number and the processor context of the executing thread. The handler routine may take various actions, such as processing an event and/or awakening a regular thread in the actor. The handler routine then returns to the nucleus.

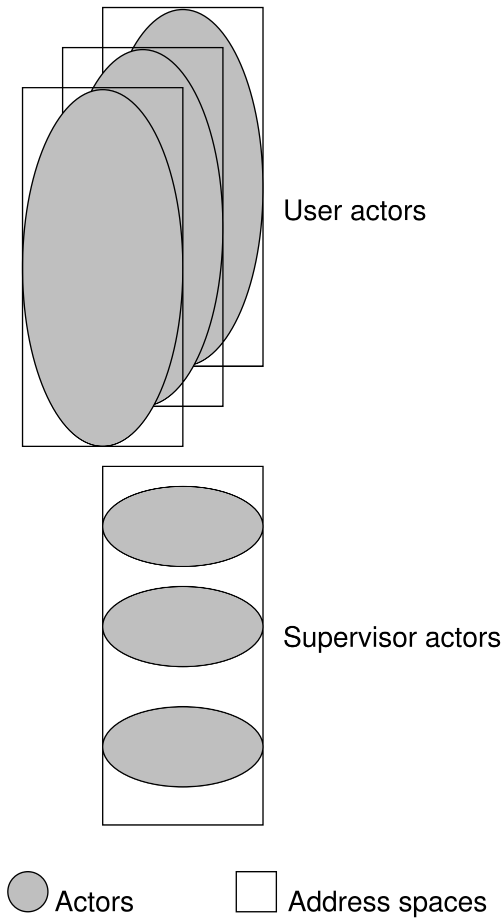

Figure 4. Supervisor Actors

4.1 External Device Drivers

It is important to note that no subsystem in CHORUS V3 is ever *required* to use connected handlers or supervisor actors. For example, a subsystem designer may choose to export a programming interface based entirely upon messages rather than upon traps. The CHORUS nucleus can handle program exceptions either by sending an RPC message to a designated exception port or by calling a connected exception handler. Only actors that process device interrupts are required to be implemented as supervisor actors. Even so, device drivers may be split into two parts, if desired; a “stub” supervisor actor to translate interrupts into messages and a user-mode actor that processes these interrupt messages. Connected handlers, however, provide significant advantages in both performance and binary compatibility:

- The nucleus need not be modified each time that a new device type is to be supported on a given machine;
- Interrupt processing time is greatly reduced, allowing real-time applications to be implemented outside of the nucleus.

Connected interrupt handlers allow device drivers to exist entirely outside of the nucleus, and to be dynamically loaded and deleted, with no loss in interrupt response or overall performance. For example, to demonstrate the power and flexibility of the CHORUS V3 nucleus, we have constructed a user-mode file manager that communicates using CHORUS IPC with a supervisor actor which manages a physical disk. Both the supervisor actor and the user-mode file manager can be dynamically loaded from a remote site. Additionally, the user-mode file manager can be debugged using standard debuggers.

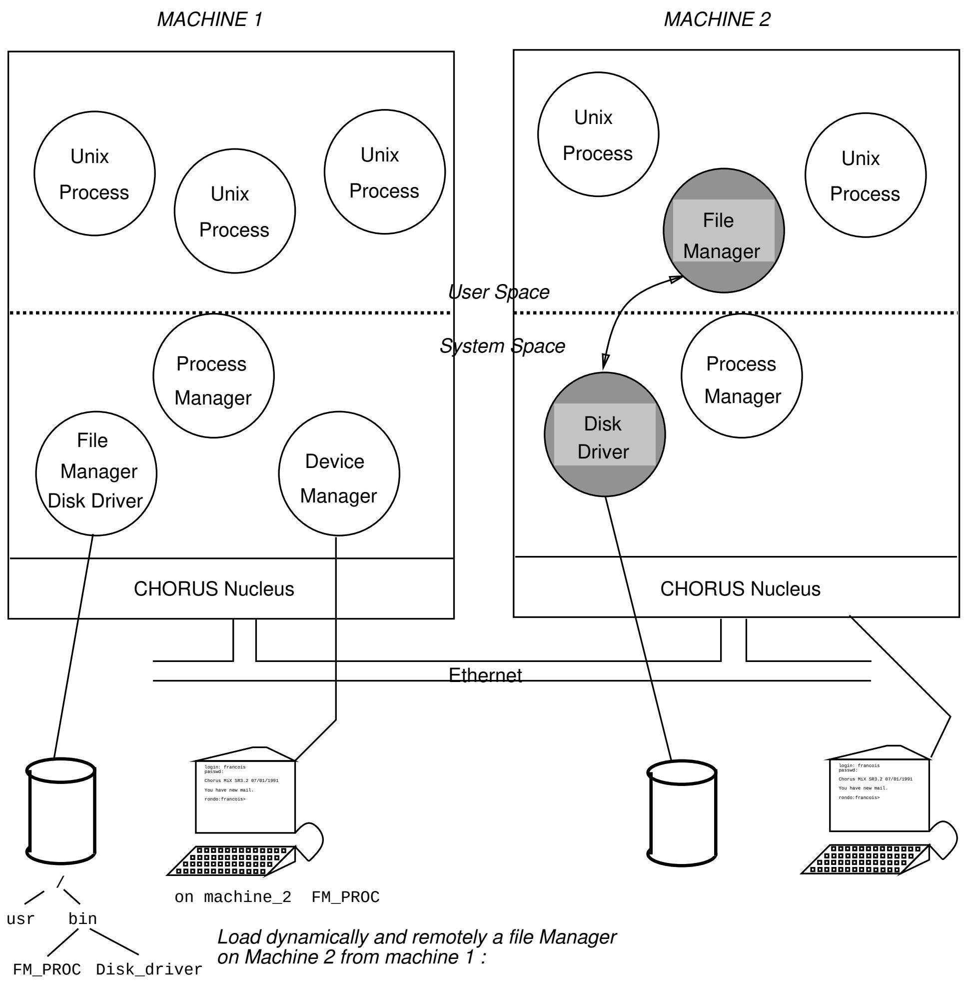

Figure 5. CHORUS/MiX File Manager as a UNIX Process in User Space

Interrupt handlers may be stacked, since multiple device types often share a single interrupt level. In this case the sequence of handlers is executed in priority order until one of them returns a code indicating that no further handlers should be called. Connected interrupt handlers have been designed to allow subsystems to incorporate proprietary, object-only device drivers that conform to one of the relevant binary standards that are emerging in this area. Without this mechanism, object compatibility would require incorporating entire device drivers within the nucleus.

4.2 Compatibility

System call trap handlers are essential for both performance and, as it has been pointed out in [Tan90], binary compatibility. Any subsystem may dynamically connect either a general trap-handling routine or a table of specific system call handlers, the latter providing an optimised path for UNIX-style interfaces. An alternative mechanism, the system-wide user-level shared library used in CHORUS V2, would seem to provide equivalent system call performance. However, we found that it is difficult to protect subsystem data that share the address space of the user program, especially if processes are multi-threaded. As we have seen, malicious or innocent but erroneous programs can change the behaviour of system calls. If functions must be moved from the shared library into separate servers for protection, increased IPC traffic results. Finally, the presence of the library code and data in the user context can interfere with programs that use a large portion of the address space or manage the address space in some particular fashion. Traps to supervisor actors, by contrast, provide a low-overhead, self-authenticating transfer to a protected server, while maintaining full transparency for the user program.

> **Lesson:** *Use of shared libraries produces compatibility and error-detection problems. For 100% UNIX binary compatibility, it is necessary to maintain the standard UNIX trap interface and address space layout.*

4.3 Performance Benefits

Performance benefits of supervisor actors come in several areas. Memory and processor context switches are minimised through use of connected handlers rather than messages, and in general through address-space sharing of actors of a common subsystem which happen to be running on a single site. Trap expense can be avoided for nucleus system calls executed by supervisor actors. Finally, supervisor actors allow a new level of RPC efficiency. The "lightweight RPC" mechanism of [Ber90] optimises pure RPC for the case where client and server reside on the same site. We further optimise for the case where no protection barrier need be crossed between client and server. This "featherweight" RPC is substantially lower in overhead, while still mediated by the nucleus and still using an interface similar to that of pure RPC.

> **Lesson:** *Implementing part of an operating system in user-level servers, while elegant, imposes prohibitive message passing and context switching overheads not present in a monolithic implementation of the system. To allow microkernel technology to compete in the marketplace, it was necessary to provide a solution to these problems. Supervisor actors provide the advantages of a modular system while minimally sacrificing performance.*

4.4 Construction of Subsystems

Subsystems may be constructed using combinations of supervisor or user actors. Any server may itself belong to a subsystem, such as UNIX, as long as it does not produce any infinite recursions, and may be either local or remote. Servers that need to issue privileged instructions or that are responsible for handling traps or interrupts must be supervisor actors.

4.5 Protection Issues

Computer systems often give rise to tradeoffs between security and performance, and we must consider the nature of the sacrifice being made when multiple servers and the microkernel share the supervisor address space. Protection barriers are weakened, but only among mutually-trusted system servers. The ramifications of the weakening of protection barriers can be minimised by systematically adhering to the following design rule: individual servers must never pass data through shared memory.

Allowing a server to explicitly access other servers' data would completely break system modularity. This rule being enforced, the only genuine sacrifice for using supervisor actors is a degree of bug isolation among the components of a running system. This is somewhat mitigated by the fact that subsystem servers may be debugged in user mode. In fact, this forms our day-to-day development activity: servers are developed and debugged in user mode. When validated, they are loaded as supervisor actors for better performance, if desired. However, the overall CHORUS philosophy is to allow the subsystem designer or even a system manager to choose between protection and performance on a case-by-case basis, and to alter those choices easily.

5. Evolution in CHORUS IPC

CHORUS V3 IPC is based on the accumulated experience gained since CHORUS V0. Here again, the main characteristics of the IPC facilities are their simplicity and performance.

5.1 Naming

The first aspect which has evolved since V2 is naming: for many reasons, distributed applications need to transfer names among their individual components. This is most efficiently achieved with a single space of global names that are usable in any context, from nucleus to application level. The main difficulty with this style of naming is protection.

In CHORUS V3, ports and port groups are named using unique identifiers which are visible at every level. Basic protection for these names is threefold:

1. All messages are stamped by the nucleus with the sending port's unique identifier as well as its *protection identifier*. Protection identifiers allow the source of a message to be reliably identified as they may be modified only by trusted actors. Using these facilities provided by the nucleus, subsystems have the choice to implement their own more stringent user authentication mechanisms if needed.

2. Global names are randomly generated in a large, sparse name space; knowing a valid global name does not help much in finding other valid names.

3. Objects within CHORUS may be named using capabilities which consist of a <name, key> tuple. Capabilities are constructed using whatever techniques are deemed appropriate by the server that provides them, and may incorporate protection schemes.

Port groups, as implemented by the nucleus, have keys related to the group name by means of a non-invertible function. Knowledge of the group name conveys the right to send messages to the group, but knowledge of the key is required to insert or delete members from the group.

Higher degrees of port and/or message security can be implemented by individual subsystems, as required. Subsystems may act as intermediaries in message communications to provide protection, or may choose to completely exclude CHORUS IPC from the set of abstractions they export to user tasks.

5.2 Message Structure

A second area of evolution in the CHORUS V3 IPC is message structure.

The memory management units of most modern machines allow moving data from the address space of one actor to the address space of another actor by remapping. This facility is exploited in CHORUS V3 IPC, which allows transmission of message bodies between actors within a single site by means of address remapping. In situations where data is to be copied and not moved between address spaces, CHORUS V3 has copy-on-write facilities that allow the data to be efficiently transferred only as needed. The typical communication that makes use of this facility involves the exchange of a large amount of data (e.g. I/O operations).

It is often the case that messages contain a large data area, accompanied by some auxiliary information such as a header or some parameters, such as a path-name, a size, or the result of an I/O operation. Frequently, the auxiliary information is physically disjoint from the primary data. In CHORUS V2, assembling these two discontiguous fragments into a single message required that extra copying be done by the user.

CHORUS V3 splits message data into two parts:

- a message body, which has a variable size and may be copied or moved; it typically contains the raw data;
- the message annex, which has a fixed size and is always copied; it typically contains the associated parameters or headers.

This division also allows one software layer to provide data, while another provides header or parameter information. For example, the V3 implementation of the `write` system call receives the address of a data buffer from the caller and appends a header describing the data area and sends both to the device responsible for performing the operation.

5.3 Processing vs. Communication

A third issue is the relationship between the processing model and communication model. The CHORUS V2 execution model was event or communication-driven. In CHORUS V3, the processing model has been inverted – actors are multi-threaded and the basic mechanism for interprocess synchronisation is RPC. Thus, the CHORUS V3 model is much closer to the traditional procedural model of computation. Multi-threading allows the multiplexing of servers, simplifying their control structure while potentially increasing concurrency and parallelism. RPC is well understood and straightforward to program.

In addition, for applications that require basic, low-level communication, asynchronous IPC is provided. This IPC has very simple semantics – it provides unidirectional communication incorporating location transparency, with no error detection or flow control. Higher-level protocol layers provided by the user or subsystem can be built on top of this minimal nucleus function.

6. Conclusion

With CHORUS V2, we experimented with a first-generation microkernel-based UNIX system. UNIX emulation was built as an application of a pure message-based microkernel. Our microkernel approach proved its applicability to building UNIX operating systems for distributed architecture in a research environment.

The challenge in designing CHORUS V3 was to make this technology suitable for commercial systems requirements; to provide performance comparable to similar monolithic systems and to provide full compatibility with these systems. Our second-generation microkernel design was driven by these requirements and we were forced to reconsider the role of the microkernel. Instead of strictly enforcing a single, rigid, system architecture, the microkernel is now comprised of a set of basic, flexible, and versatile tools. Our experience with CHORUS V2 taught us that some functions, such as IPC management, belong within the microkernel. Device drivers and support for heterogeneity, on the other hand, are best handled by separate servers and protocols. Supervisor actors are crucial to both performance and binary compatibility with existing systems. A global name space is necessary to simplify the interactions between system servers and the nucleus. Using CHORUS V3, subsystem designers have the freedom to define their operating system architecture and to select the most appropriate tools. Decisions, such as the choice between high security and high performance, are not be enforced a priori by the microkernel.

The CHORUS V3 microkernel has met its requirements: the CHORUS/MiX microkernel-based UNIX system provides the level of performance of real-time executives, is compatible with UNIX at the binary level, and is truly modular and fully distributed. It has been adopted by a number of manufacturers for real-time and distributed commercial UNIX systems.

Further work will concentrate on exploiting this technology to provide advanced operating system features, such as a distributed UNIX with a single system image and fault tolerance.

References

[Abr89] V. Abrossimov, M. Rozier, and M. Shapiro, ‘‘Generic Virtual Memory Management for Operating System Kernels,’’ in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ (USA), (December 1989).

[Arm86] François Armand, Michel Gien, Marc Guillemont, and Pierre Léonard, ‘‘Towards a Distributed UNIX System – The CHORUS Approach,’’ in *Proceedings of the EUUG Autumn’86 Conference*, Manchester, England, (Autumn 1986).

[Arm89] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier, ‘‘Revolution 89 or ‘Distributing UNIX Brings it Back to its Original Virtues’,’’ in *Workshop on Experiences with Building Distributed and Multiprocessor Systems*, Ft. Lauderdale, FL (USA), (October 1989).

[Arm90] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier, ‘‘Multithreaded Processes in CHORUS/MiX,’’ in *Proceedings of the EUUG Spring’90 Conference*, Munich, Germany, (April 1990), pp. 1-13.

[Ber90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy, ‘‘Lightweight Remote Procedure Call,’’ *ACM Transactions on Computer Systems*, vol. 8, no. 1, (February 1990), pp. 37-55.

[Che90] David R. Cheriton, Gregory R. Whitehead, and Edward W. Sznyter, ‘‘Binary Emulation of UNIX using the V Kernel,’’ in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA (USA), (June 1990), pp. 73-86.

[Gol90] Davic Golub, Randall Dean, Alessandro Forin, and Richard Rashid, ‘‘UNIX as an Application Program,’’ in *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA (USA), (June 1990), pp. 87-96.

[Her88] Frédéric Herrmann, François Armand, Marc Rozier, Vadim Abrossimov, Ivan Boule, Michel Gien, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, ‘‘CHORUS, a New Technology for Building UNIX Systems,’’ in *Proceedings of the EUUG Autumn’88 Conference*, Cascais, Portugal, (October 1988), pp. 1-18.

[Pik91] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ‘‘Designing Plan 9,’’ *Dr. Dobb’s Journal*, vol. 16, no. 1, (January 1991), pp. 49-60.

[Roz87] Marc Rozier and José Legatheaux-Martins, ‘‘The CHORUS Distributed Operating System: Some Design Issues,’’ in *Distributed Operating Systems, Theory and*

Practice, Springer-Verlag, Berlin, BRD, (1987), pp. 261-286.

[Roz88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser, ''CHORUS Distributed Operating Systems,'' *Computing Systems Journal*, vol. 1, no. 4, The Usenix Association, (December 1988), pp. 305-370.

[Tan90] Andrew Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum, ''Experiences with the Amoeba Distributed Operating System,'' *Communications of the ACM*, vol. 33, no. 12, (December 1990), pp. 46-63.

Author Biographies

Allan Bricker <allan@chorus.fr> is a Senior Engineer at Chorus systèmes. In 1985 he received his Master of Science Degree in Computer Science from the University of Wisconsin-Madison. From 1984 to 1989 he was the senior researcher in charge of the design and development of the multi-threaded communications kernel for the Gamma parallel database machine at the University of Wisconsin.

Michel Gien <mg@chorus.fr> is a co-founder, general manager, and director of R&D at Chorus systèmes. He joined the Cyclades computer network team at INRIA in 1971 after graduating from Ecole Centrale des Arts et Manufactures de Paris. He then led a project that introduced UNIX in France and helped to understand how it could be re-designed along the CHORUS distributed systems concepts. Michel is currently serving as the chair of EurOpen (formerly EUUG) after having served as its vice chair since 1985.

Marc Guillemont <mgu@chorus.fr> is a co-founder and the director of engineering at Chorus systèmes. He joined INRIA in 1977 to work on the Cyclades project and was also member of the initial CHORUS research project team in 1980 before he became head of the team. He managed the final research phases of CHORUS before developing the commercial version. Marc Guillemont graduated from Ecole Polytechnique in 1971 and earned a PhD in Computer Science from Grenoble University.

Jim Lipkis <lipkis@chorus.fr> is a Senior Engineer at Chorus systèmes. At New York University, he was responsible for design and development of operating system and programming language software for highly parallel shared-memory multi-processors, including the NYU Ultracomputer.

Douglas Orr <doug@chorus.fr> is a Senior Engineer at Chorus systèmes. He graduated from the University of Michigan and has worked for Apollo Computer and Carnegie Mellon University. His interests include operating systems, computer networks, and existentialism.

Marc Rozier <mr@chorus.fr> is the head of the CHORUS distributed microkernel development team within Chorus systèmes. He graduated from ENSIMAG before earning a PhD in Computer Science from the University of Grenoble. He joined INRIA in 1982 as a researcher in the CHORUS distributed operating system project. He worked on both the design and implementation of two versions of CHORUS. In 1987, he became one of the founders of Chorus systèmes.

Authors may be contacted at

Chorus systèmes
6, avenue Gustave Eiffel
F-78182 Saint Quentin-en-Yvelines Cedex
France

Tel: +33 1 30 64 82 00
Fax: +33 1 30 57 00 66