Revolution 89
or
‘‘Distributing UNIX Brings it Back to its Original Virtues’’

François Armand, Michel Gien, Frédéric Herrmann, Marc Rozier

Chorus systèmes
6, avenue Gustave Eiffel, F–78182, Saint-Quentin-en-Yvelines (France)
Tel: +33 1 30 64 82 00, Fax: +33 1 30 57 00 66, E-mail: mg@chorus.fr

1. Introduction

The need to handle distributed computing in a general manner leads us to structure our operating systems functions in a much more modular way than it is done in today’s systems such as current UNIX kernels, and to provide facilities for dynamic reconfiguration so the system can be adapted to the variety of configurations needed in a distributed system. When applied to the UNIX kernel, such a "restructuring" leads to same kind of "revolution" that UNIX performed on the operating systems of the 70’s, i.e., to extract from the operating system all functions that can better be performed outside, and to leave in the kernel only those generic services that are necessary to provide higher level services, such as high level file access methods, command languages (shell) or system administration functions.

The CHORUS[1] architecture is designed to support new generations of open, distributed, scalable operating systems. It allows the integration of various families of operating systems, ranging from small real-time systems to general-purpose operating systems, in a single distributed environment.

The CHORUS architecture is based on a minimal real-time Nucleus that integrates distributed processing and communication at the lowest level. CHORUS operating systems are built as sets of independent system servers, that rely on the basic, generic services provided by the Nucleus i.e., thread scheduling, network transparent IPC, virtual memory management and real-time event handling.

The CHORUS Nucleus itself can be scaled to exploit a wide range of hardware configurations, such as embedded boards, multi-processor and multi-computer configurations, networked workstations and dedicated servers.

Operating systems (called Subsystems) implemented on top of this Nucleus currently include a UNIX[2] SYSTEM V[Herr88] and the "Emeraude"[Mino88] CASE/PCTE system. Work is currently in progress to implement Object-Oriented distributed Subsystems.[Alve88]

CHORUS–V3 is the current version of the CHORUS system developed by Chorus systèmes. Earlier versions were designed and implemented within the Chorus research project at INRIA between 1979 and 1986. Related work includes the V-system[Cher88] for the message-passing kernel, Mach[Rash87] and[Li86] for the distributed virtual memory, Topaz[McJo88] and Mach[Acce86] for the ‘‘threads’’, Amoeba[Mull87] for the global addressing, and the Bell Laboratories’9th Edition UNIX[Pres86, Wein86] for the uniform file naming.

1. CHORUS is a registered trademark of Chorus systèmes
2. UNIX is a registered trademark of AT&T

In: Proceedings of "Workshop on Experiences with Building Distributed (and Multiprocessor) Systems", 5-6 Oct. 1989, Ft. Lauderdale, FL, USA, pp.153-174.

CHORUS–V3 is written in C++ (and C). It currently supports 680X0 and 80386 based machines, with implementations on networked workstations and servers, as well as multi-processor configurations.

This paper outlines the architecture and implementation of UNIX kernel functions in terms of the CHORUS architecture concepts, based on the CHORUS Nucleus basic services. It focuses on the experiences drawn from it, the resulting benefits for users as well as for systems designers and maintainers, and the issues that still need to be considered.

The next section summarizes the CHORUS Nucleus’ basic abstractions and services as described extensively in. [Rozi88] Section 3 outlines the structure of a UNIX Subsystem, in terms of independent cooperating CHORUS servers, illustrating how one can make use of the Nucleus facilities in a UNIX context. Straightforward extensions in the services provided at the UNIX kernel interface level will also be presented. The remaining sections give examples of using the CHORUS architecture in typical system configurations and operating system experiments.

2. The CHORUS Architecture

2.1 Overall Organization

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

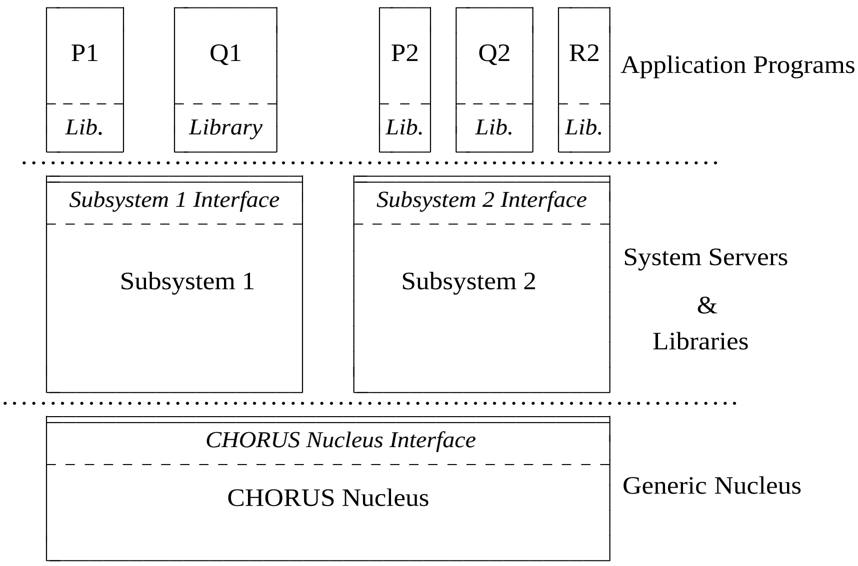

Figure 1. – The CHORUS Architecture

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 a *virtual memory manager*,
   - external events – interrupts, traps, exceptions – are dispatched by a *supervisor*.

2. Global services:
   An *IPC Manager* provides the communication service, delivering messages regardless of the location of their destination within a CHORUS distributed system. It may rely on external system servers

(i.e.,
Network Managers) to operate all kinds of network protocols.

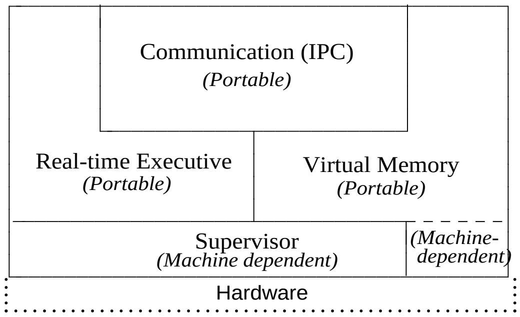

Figure 2. – The CHORUS Nucleus

2.2 The CHORUS Nucleus basic abstractions

The physical support for a CHORUS system is composed of a set of *sites* (‘‘computers’’, or ‘‘boards’’), interconnected by a communication *network* (i.e., a real network or a bus). A **site** is a tightly coupled grouping of physical resources: one or more processors, memory, and attached I/O devices. There is one CHORUS Nucleus per site.

The **actor** is the logical unit of distribution and of collection of resources in a CHORUS system. An actor defines a protected address space supporting the execution of one or more *threads* (lightweight processes) that share the address space of the actor. An address space is split into a *user address space* and a *system address space*. On a given site, each actor’s system address space is identical and its access is restricted to privileged levels of execution (Figure 3).

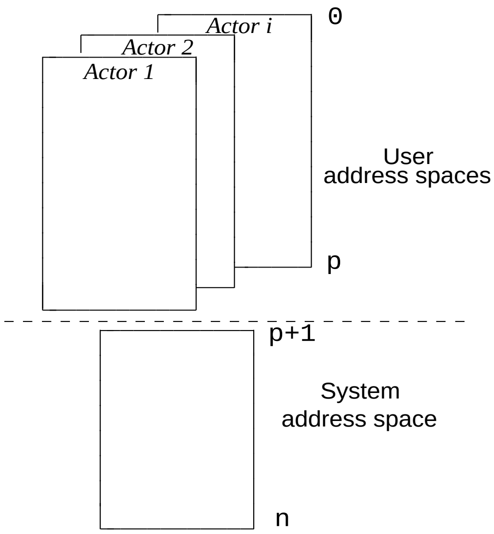

Figure 3. – Actor Address Spaces

Any given actor is tied to a site, and its threads are executed on that site. A given site may support many simultaneous actors. Since each has its own ‘‘user’’ address space, actors define protected *virtual machines*.

The **thread** is the unit of execution in a CHORUS system and is characterized by a *context* corresponding to the state of the processor (registers, program counter, stack pointer, privilege level, etc.). A thread is always tied to one and only one actor. These threads share the resources of that actor and no other actor. Threads are scheduled by the Nucleus as independent entities. In particular, threads of an actor may run in parallel on the many processors of a *multiprocessor* site. The scheduling of threads is preemptive, based on their fixed priorities.

Besides the shared memory provided by the actor address space, CHORUS offers message-based facilities (referred to as *IPC*) which allow any thread to communicate and synchronize with any other thread, on any site. The CHORUS IPC permits threads to exchange messages either *asynchronously* or by *demand/response*, also called *Remote Procedure Call* (*RPC*). Its main characteristic is its transparency with respect to the location of threads: the communication interface is uniform, regardless of whether it is between threads in a single actor, between threads in different actors on the same site, or between threads in different actors on different sites.

A **message** is composed of a (optional) *message body* and a (optional) *message annex*. Both are untyped string of bytes. Message passing is tightly coupled with the virtual memory mechanism to enable data transmission without copy.

Messages are not addressed directly to threads, but to intermediate entities called *ports* (Figure 4).

A **port** is an address to which messages can be sent, and a queue holding the messages received but not yet consumed by the threads. A port can only be attached to a single actor at a time, but can be attached to different actors successively, effectively *migrating* the port from one actor to another.