CHORUS: A SUPPORT FOR DISTRIBUTED AND RECONFIGURABLE ADA SOFTWARE

Marc Guillemont

ABSTRACT

In the ESA Columbus Project, including in particular the European Space Station, applications are written in Ada and are distributed; moreover, the long lifetime of the space elements requires flexibility in order to support smooth software evolution.

CHORUS® is a distributed operating system, developed and supported by Chorus systèmes, which provides a basis for supporting efficiently distributed and dynamically reconfigurable Ada software.

Keywords: real time, distribution, operating system, reconfiguration, Ada.

1. COLUMBUS REQUIREMENTS FOR DMS

The main requirements for the Columbus Data Management System (DMS) may be summarised as follows:

- Provide a real-time multitasking environment for applications in a distributed architecture. Provide an inter-application communication in a distributed architecture.
- Independent of hardware and network specificities.
- Support applications written in Ada; these applications may be distributed and they may have to cooperate with applications written in other languages.
- Support dynamic application replacement, with minimum impact on the application.

These two last requirements have led the Columbus project to introduce the notion of the SoftWare Replaceable Unit (SWRU), i.e. a piece of software written in Ada (or possibly other languages) which should be replaceable like hardware pieces are.

2. CHORUS OVERVIEW

This document describes how CHORUS fulfills the above requirements, thanks to the following properties of its architecture:

1. Modularity: the complete operating system is built as a small real-time kernel complemented by servers. *Communications are integrated within the heart of the small real-time kernel.*

   The real-time kernel is able to support efficiently Ada applications.

2. Hardware and network independence: the real-time kernel insulates applications from hardware as well as network specificities.
3. Flexibility: the versatility of the inter-application communication is the basis for supporting distribution in an efficient and simple way, in particular for dynamic configuration and reconfiguration of applications.
4. Extensibility: the modularity of the architecture and the integration of inter-application communications within the kernel allows to extend the architecture with new services as necessary.

The integration of communications within the system kernel and the flexibility of the inter-application communication facility are the key features of this architecture for fulfilling the DMS requirements.

2.1 Overall Organisation

The CHORUS System is composed of three layers (Figure 1):

1. A small real-time kernel (called **Nucleus**) integrating inter-application communications. This Nucleus insulates higher-level software from hardware and network specificities.
2. A set of **System servers** complementing the Nucleus with necessary higher-level services, in the context of **Service assemblies**. These services integrate the distributed nature of the architecture.

   Where necessary, different assemblies may be built on top of the same Nucleus.

3. **Applications**, mainly SWRUs, running on top of the assemblies.

2.1.1 The Nucleus

The Nucleus (Figure 2) plays a double role:

1. Local services:

   It manages, at the lowest level, the local physical resources of a ‘‘computer’’, called a **Site**, by means of three clearly identified components:

Part of this work has been conducted within the study *Proof of concept for Ada SWRU implementation* with the partnership of Alsys, Intecs International and Syseca, in the framework of phase C0 of Columbus.

® CHORUS is a registered trademark of Chorus systèmes.

Presented at the ESA Workshop on Communication Networks and Distributed Operating Systems within the Space Environment, 24-26 October 1989.

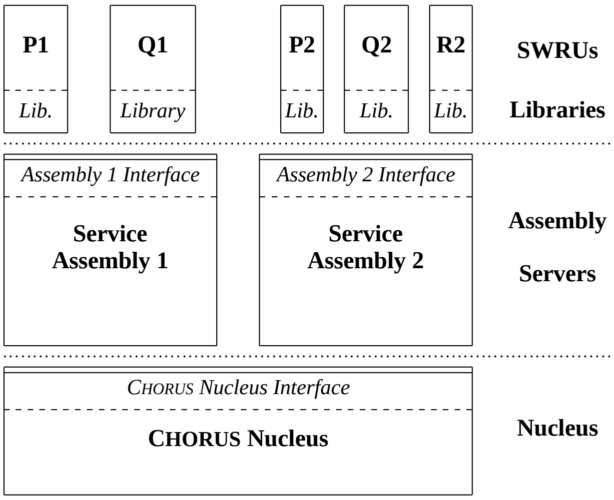


Figure 1. The CHORUS Architecture

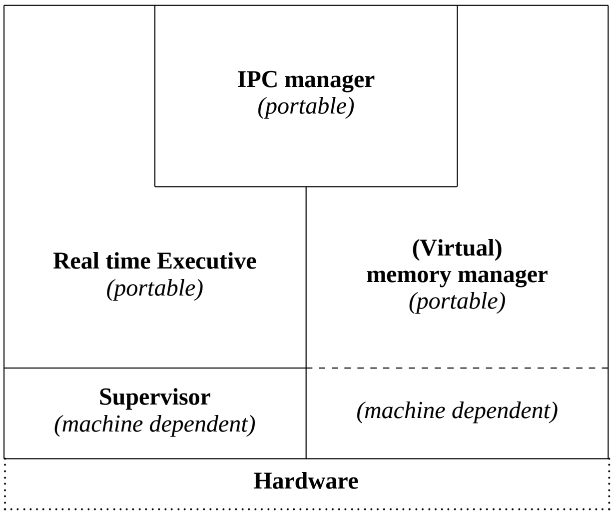


Figure 2. The Nucleus

- allocation of local processor(s) is controlled by a **Real-time Multi-tasking Executive**; this executive provides fine grain synchronisation and priority-based preemptive scheduling,
- local memory is managed by the **(Virtual) Memory Manager** controlling memory space allocation and structuring (virtual) memory address spaces,
- external events − interrupts, traps, exceptions − are dispatched by the **Supervisor**.

2. Global services:

   The **IPC** (Inter Process Communication) **Manager** provides the communication service, delivering messages regardless of the location of their destination within a distributed system.

The integration, within the same Nucleus, of a real-time executive and communication brings several advantages:

- the interaction between communications and scheduling (e.g. the reception of a message enables scheduling) is efficiently supported;
- the optimisation of communications (e.g. local transfer of a message by moving descriptors instead of by copying the value) relies on a proper usage of memory management.

> *Within the Nucleus, the non portable and portable parts are clearly separated: the transport of the CHORUS Nucleus to a new hardware impacts only a limited part of the Nucleus.*

2.1.2 The Assemblies

System servers implement high-level system services, and cooperate in order to provide a coherent operating system interface. They communicate via the Inter-Process Communication facility (IPC) provided by the Nucleus (Figure 3). The main servers which may be found in Assemblies are:

1. The *Process Manager* manages processes with all the semantics of the assembly; it deals with creating processes, loading their binary image from secondary memory into main memory, destroying processes, etc...
2. The *File Manager* is in charge of managing secondary storage (e.g. disks, tapes, floppies, etc...) and paging for Virtual Memory.
3. The *Device Manager* is in charge of managing character devices (e.g. terminals).
4. Other managers may be added for managing other devices (e.g. sensors and actuators) or other terminals (e.g. bitmap displays), for DataBase management, for specific communications (e.g. space/ground), etc...

2.1.3 Applications

SWRUs execute on top of the CHORUS assembly.

2.1.4 System Interfaces

A CHORUS system exhibits several levels of interface (Figure 3):

- **Nucleus Interface**: this interface is composed of a set of procedures providing access to the services of the Nucleus. If the Nucleus cannot render the service directly, it communicates with a distant Nucleus via the IPC.

  This interface is used only by assembly servers.

- **Assembly Interface**: this interface is composed of a set of procedures providing access to some assembly specific protected data. If a service cannot be rendered directly from this information, these procedures ‘‘call’’ (RPC) the services provided by assembly servers, which may be local or remote.

- **Application Interface**: communications between SWRUs rely on the inter-process communication (IPC) service provided by the CHORUS assembly, but this IPC will not be used directly within application software. Instead, accesses to an application program are described by interface packages which provide the necessary interface for accessing the services offered by the application with a dedicated syntax and semantic (§ 2.1).

2.2 Inter-process Communication

The inter-process communication services play a double role:

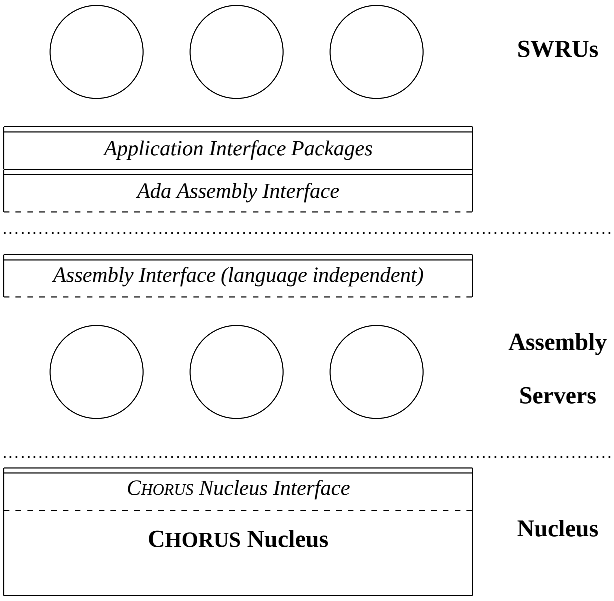


Figure 3. Architecture and interfaces

1. They provide the only means of interaction between processes of an application: these services must include the facilities for naming, binding and addressing the various processes; in addition, they must include ways for controlling inter-application interactions.
2. They provide the basis for simple and dynamic configuration and reconfiguration of applications: these services must be flexible and high-level enough in order to simplify the (re)configuration management at the application level.

In addition, the simplicity of the interface and the unicity of this inter-application interaction (called *the single interface* in [3]) allows testing any application by controlling and exercising the communications with the application.

2.2.1 Overview

Processes synchronise and communicate using a single basic mechanism: the exchange of **messages** via message queues called **Ports**.

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

2.2.2 Messages

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

2.2.3 Ports

Messages are not addressed directly to processes, but to intermediate entities called **ports**. The notion of a port provides the necessary decoupling between the interface of a service and its implementation. In particular, it provides the basis for dynamic reconfiguration (§ 3.1).

When created, a port is *attached* to one process. A port can only be attached to a single process at a time, but it can be successively attached to different processes: i.e. a port can *migrate* from one process to another. This migration can be applied also to the messages waiting behind the port.

The geographical distribution transparency provided by the IPC allows to change the configuration of an application without any change in the source of the application (Figures 4 and 5).

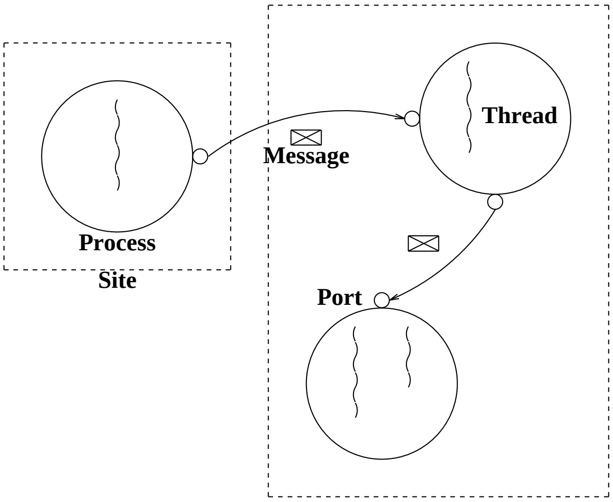


Figure 4. Processes, threads and ports

2.2.4 Port Groups

The ports provide the basis for efficient point to point communications. However, in distributed systems, this is not sufficient.

Ports can be collected into **Port Groups** (Figure 6). The notion of *group* extends port-to-port message passing between processes:

- Asking for a service may not only be done directly from one process to another process − via a port. It may also be done by ''multicast'': from one process to an *entire group of processes* − via a group of ports.
- Functional or associative access to a service can be selected from among a group of (equivalent) services.

2.2.5 Communication Semantics

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

- **Asynchronous mode**: The emitter of an asynchronous message is blocked only during the time of local processing of the message by the system. The system does not guarantee that the message has been actually received by

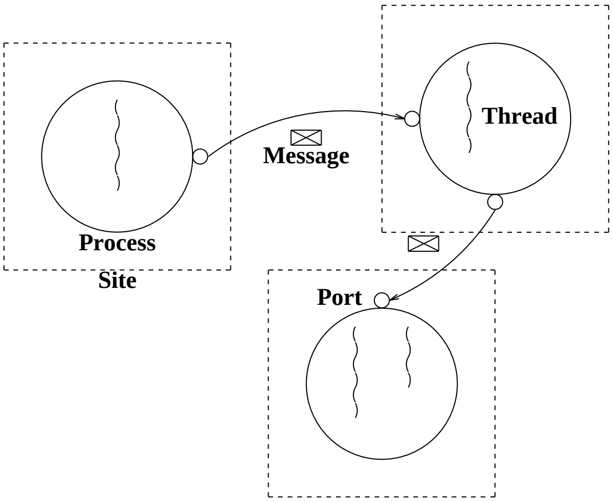


Figure 5. Changing the configuration of an application (3 sites instead of 2)

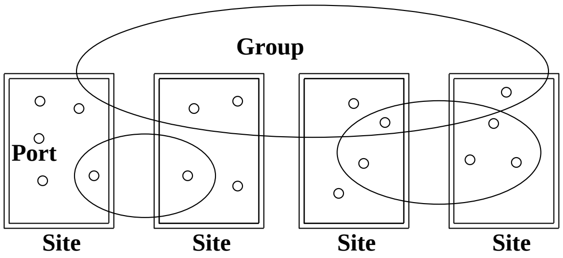


Figure 6. Port Groups

the destination port or site. When the destination port is not present, the sender is not notified, and the message is destroyed.

- **RPC mode**: The RPC protocol permits the construction of services with a **client-server** model: a demand/response protocol with management of **transactions**. RPC guarantees that the response received by a client is definitely that of the server and corresponds effectively to the request (and not to a former request to which the response would have been lost); RPC also permits a client to know if his request has been received by the server, if the server has crashed before emitting a response, or if the communication path broke.

When messages are sent to port groups, several addressing modes are provided:
— broadcast to all ports in the group,
— send to any one port of the group,
— send to one port of the group, located on a given site,
— send to one port of the group, located on the same site as a given object.

3. ADA SUPPORT

The requirements for Ada support in Columbus DMS have two main aspects:

- Distribute Ada applications.
- Efficiently support real time Ada applications.

This section briefly discusses different alternates and presents how CHORUS provides a solution.

3.1 Distribution of Ada applications

The main alternates may be summarised as follows:

a. A distributed application is one Ada program. This Ada program is "automatically" split into parts which are distributed among the various nodes; the semantics of Ada are preserved through this distribution, e.g. rendez-vous are distributed, etc...

   This alternative implies important and difficult developments both in the compiler and in the run time support.

b. A distributed application is made of several Ada programs; however, the communications between two programs are expressed in the same way as within one program, i.e. there is no difference in programming between internal and external communications.

   This alternative implies important and difficult developments both in the "compiling process" [1] and in the operating system.

c. A distributed application is made of several Ada programs; the communications between two programs are expressed in terms of facilities provided by the underlying operating system; at some level, distribution is not hidden, though it can be limited to specific interface packages.

   This alternative has the advantage of requiring the minimum adaptations to the Ada development environment and to the operating system, thus preserving the capacity to benefit from the evolutions of both technologies.

This last alternative has been chosen in order to support distributed Ada applications on CHORUS. As mentioned in § 1.1.4, the IPC facility may be directly accessed from an Ada program; however, this is restricted to specific Application Interface Packages (or Service Interface Packages – SIP) which encapsulate the communications between application. Encapsulating inter-application communications within specific packages brings several advantages:

– the interface package insulates the application from the knowledge of the protocol with the server: format of messages, rules of exchange, etc...

– the interface package may contain additional controls as well as adaptation to a specific language (in particular Ada);

– the interface package may provide the necessary protocol for dealing with possible reconfiguration or evolution of the server: retransmission of requests, server supervision, etc...

– the interface package may integrate the necessary control for dealing with software evolution: message type checking, protocol version control, etc...

1. Compiling process may include additional pre-processing, compiling and post-compiling operations.

– in the case of Ada, the interface package allows to have a better compilation control than a generic IPC interface may provide.

3.2 Efficient Ada support

Efficient Ada support may be summarised in four main requirements:

1. Each SWRU must require a minimum amount of memory; besides efficient code generation, this must be achieved through code sharing where possible and, in particular, run-time sharing.
2. Conversely, the minimum operating system necessary to support Ada programs must be really small.
3. Real time scheduling must not be restricted to the inside of one Ada program: all Ada programs sharing the same node must be submitted to a common real time scheduling, allowing any priorities combination.
4. The request of a blocking system call by a single task must not block the whole program.

It is worth noting that for mono-task programs, requirements 3 and 4 are easily provided: they can have (program) priorities interleaved and their (only) task can request blocking calls without any undesired side-effects.

The adaptation of CHORUS in order the above requirements has one possible trade-off:

a. CHORUS can be adapted towards an Ada run-time and directly support the Ada semantics.

b. The Ada run-time can be adapted in order to benefit from the facilities provided by CHORUS.

   This last trade-off has the advantage of preserving the capacity to benefit from the evolution of the CHORUS technology; in addition, it does not tie CHORUS to one particular compiler manufacturer.

This last alternative has been chosen in order to support Ada programs on CHORUS. This support is based on the following mapping:

- One Ada program is executed as one CHORUS process; one program is therefore not distributed (§ 2.1). Several Ada programs may run on the same node and may share the node with non Ada programs.
- One Ada task is mapped onto one CHORUS thread [2]. The real time task scheduling is therefore performed by the CHORUS Nucleus.

  When a task requests a blocking system call, only the corresponding thread is blocked. Other threads, i.e. other tasks, continue to be scheduled by the Nucleus.
- The Ada run-time benefits from the CHORUS services in order to implement Ada semantic (delay, rendez-vous, etc...) and CHORUS provides all necessary services.

  If necessary, further refinements in the mapping may give better performance; in addition, it can be envisaged to add to CHORUS one or two simple services which could also speed up critical mechanisms (e.g. rendez-vous).

  This mapping is complemented by the usage of CHORUS' Virtual Memory facilities which allow memory – and in particular code – sharing.

2. A thread is the execution unit in CHORUS; one process may contain several threads which share all process' resources.

4. RECONFIGURATIONS

The required reconfigurations capabilities in Columbus may be summarised as follows (Ref. [3]):

- All on board software is a set of SWRUs
- One SWRU must be replaceable by one (or several) other SWRU(s); for some critical functions, this replacement must not introduce any disruption in the executing software.
- The replacement of a SWRU may leave the overall application unchanged, or it may introduce an upgrade in the interfaces; consistency of the whole application must be kept, possibly leading to the replacement of other related SWRUs.
- The replacement of a SWRU may lead to the migration of the SWRU from one node to another; consistency of the whole application must be kept.
- In case of node failure, it must be possible to reconfigure the lost application on a backup node in order to obtain full functionality of the failed application.

CHORUS provides a basis for supporting efficient dynamic application reconfiguration: the key point is the flexibility of the inter-process communication provided by distribution transparency, ports and groups.

4.1 Reconfiguration of a program

This section first emphasises the flexibility of the inter-process communication and then presents in more detail some typical reconfiguration scenarios.

The notion of ''port'' as an indirection between communicating threads allows to dynamically modify the implementation of a service within an process.

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

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

Recall that:

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

A client that references *a group* (rather than directly referencing the port attached to a server) can continue to obtain the needed services once the terminated port has been replaced in the group (Figure 8).

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

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

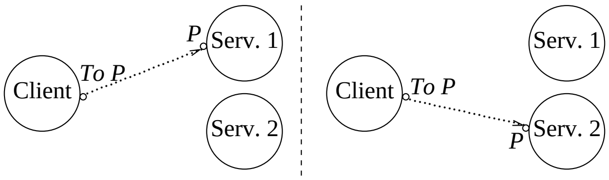
Based on these mechanisms, two typical reconfiguration mechanisms are described below:

4.1.1 Dynamic reconfiguration (Figure 7)

A Server 1 has to be replaced by another Server 2, without any service disruption:

- Server 2 is loaded on the appropriate node (same as Server 1, or different).
- Server 2 initiates itself in order to be ready for reconfiguration.
- Server 1 stops its execution in a consistent state (i.e. no pending request, etc...).
- Server 1 transmits its internal state (i.e. data which characterise its operation) to Server 2 and it requests the migration of its port P to Server 2, along with the attached messages.
- Server 2 updates its internal state with the values transmitted by Server 1.
- Server 2 is now ready to operate and processes requests received on P.
- Server 1 is destroyed.

After this reconfiguration, Server 2 is fully operational and clients cannot notice any server disruption.



Port P migrates from Server 1 to Server 2

Ports can migrate from one process to another. While Client continues communicating with port P, the port can be moved from Server 1 to Server 2. The system will, automatically, re-route the messages sent to P onto the new location of P. This allows, for example, the updating of a server with a new version or the replacement of one server with a faster one located on another site.

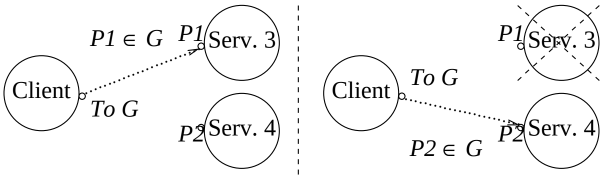
Figure 7. Reconfiguration Using Port Migration

4.1.2 Service upgrade (Figure 8)

A Server 3 has to be replaced by an upgraded version, Server 4, possibly with migration.

- Server 3 stops in a consistent state.
- Server 3 checkpoints its internal state (i.e. data which characterise its operation).
- Server 3 is destroyed (and its port P1 disappears).
- The group G is now empty: the service is now unavailable. The clients may notice it (if they try to access the service).
- Server 4 is loaded on the appropriate node and initiated.
- Server 4 reads the checkpoint data left by Server 3 and updates its internal state.
- Server 4 inserts its port P2 in the group G.
- The service is now again available.

During this reconfiguration, the service has been temporarily unavailable. It is precisely the role of application interface packages to deal with this situation and to take the appropriate decision (error reporting, request retransmission, etc...), leaving the application unaware of this situation.



P2 has replaced P1 in group G

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

Client addresses its communications to group G instead of directly to port P1. The extra level of indirection allows the replacement of Server 1, that may have ceased to function, by Server 2 even though the two servers have their own ports. The system will, automatically, re-route the messages sent to G onto P2 instead of P1.

Figure 8. Reconfiguration Using Groups

4.2 SWRUs dependencies

The above scenarios have emphasised the benefits of the IPC flexibility for ease of reconfiguration. But, reconfiguration may concern more than one SWRU, in particular in case of interface upgrade. Several alternatives may be envisaged:

1. All relations and dependencies between SWRUs are recorded in a DataBase. When a set of dependent SWRUs have to be replaced, the DataBase indicates the chain of individual operations to be performed.
2. Another approach may consist in identifying unambiguously different versions of protocols. When one SWRU is upgraded and uses another protocol version, other SWRUs may detect the upgrade (within the application interface packages) and request from the SWRU management the necessary upgrade, i.e. possibly their own reconfiguration.

These two alternatives may be combined.

5. CONCLUSION

Real time distributed Ada applications are emerging as a must in most important software developments in the near future. The technology must still mature and CHORUS can be a corner stone in this evolution.

6. ACKNOWLEDGEMENTS

This work did benefit from various contributions, in particular from Jean-Loup Gailly (Alsys), Marc Maathuis (Chorus systèmes), Michel Rabozée (Intecs International), Patrice Estève and Bertrand Hutin (Syseca).

7. REFERENCES

[1] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Pierre Léonard, Sylvain Langlois, Will Neuhauser. *CHORUS Distributed Operating Systems.* Computing Systems, vol 1, 4, Fall 1988.

[2] Chorus systèmes. *Architectural Design Document of the Distributed Operating System for the Columbus DMS.* CS/TR-89-21.2.1, May 1989.

[3] ESTEC. *Columbus System Requirements Document (SRD).* COL-RQ-ESA-001, August 1988.