The CHORUS Distributed Operating System: Some Design Issues

Marc Rozier†, José Legatheaux Martins††

Institut National de Recherche en Informatique et en Automatique
B.P. 105, 78153 Le Chesnay Cedex, France.
Tel. (1) 39 63 52 94, Telex: 697 033 F

ABSTRACT

CHORUS ™ is a portable, open, message-based, distributed operating system. Since 1980, start time of the CHORUS project at INRIA, several versions of the system have been designed and implemented. The current version, CHORUS-V2, offers a full UNIX ™ compatibility at the user level, while providing control of distribution by relying on a powerful IPC facility, based on ports and messages, as the heart of its architecture.

This paper first includes an overview of the CHORUS system, and second discusses some issues of our work: ports and port groups, IPC, naming and binding, and distributed execution control.

1. Introduction

CHORUS is the main result of a research project initiated at INRIA in 1980. It is a distributed operating system, running on a set of interconnected computers. It is designed for applications that can take advantage of distribution for dynamic reconfiguration, improved performance, better resource utilization and fault tolerance, in a transparent manner.

A prototype of the CHORUS system was written in Pascal under the Pascal-UCSD system on Intel 8086 processors. A first *real* version (CHORUS-V1) was implemented in Pascal on SM90™ computers (a 680x0 based multi-processor), interconnected with a local area network (Ethernet). This version has a proprietary interface both at the programming and at the command language levels. A second version has been designed and implemented, and is now available. One of the major extensions that can be found in this version is a UNIX interface: UNIX (System V)

† the authors list ordering reflects the responsibility engaged during the CHORUS presentation at the NATO ASI.

†† José Legatheaux Martins is with the Computer Science Departement of Universidade Nova de Lisboa - Quinta da Torre, 2825 MONTE da CAPARICA, Portugal - he is currently on leave at INRIA. Its leave has been partially supported by the Fundaçao Calouste Gulbenkian, Lisbon.

- CHORUS is a registered trademark of INRIA.
- UNIX is a registered trademark of AT&T in USA and other countries.
- SM90 is a registered trademark of CNET.

NATO ASI Series, Vol. F28
Distributed Operating Systems. Theory and Practice
Edited by Y. Paker et al.

system calls are part of the CHORUS-V2 interface, accessible to application programs. This version allows standard UNIX applications to run transparently in a distributed environment. It also provides powerful tools allowing the user to write new applications which take the real advantages of distribution.

Although CHORUS-V2 can be seen as an extended distributed UNIX system [Armand 86], its structure is very different from a standard UNIX system structure. It is based on a small message-passing kernel, and a distributed set of system processes - called **actors** - providing standard Operating System services.

In section 2, we overview the fundamentals of the CHORUS architecture. We also describe the operating system very briefly. A full description can be found in [Guillemont 86].

In CHORUS, a distributed application is a set of autonomous processes, basically clients and servers, interacting with each other by message passing. When designing such a system, the following issues are fundamental:

- defining server access points: in CHORUS, ports and port groups play this role;
- defining how clients address these access points: CHORUS offers a powerful IPC for this purpose;
- defining how are the resources named in the system: CHORUS offers several naming layers allowing location-transparent naming;
- defining how are processes and distributed protocols structured: we introduced on top of the CHORUS architecture a powerful structuring tool, the activity concept.

The discussion of these four issues is the essential purpose of this paper (sections 3 to 6 respectively).

2. System Overview

2.1. Basic concepts

In the **CHORUS basic architecture,** [Zimmermann 81, 84], a system comprises a set of autonomous entities, called **actors,** distributed across a network of computers. An actor resembles a sequential process. It has its own code, data and a context and it communicates with the outside world only by receiving and sending **messages** through **ports** which it owns. These ports have global names which are independent both of the name of the owning actor and its location.

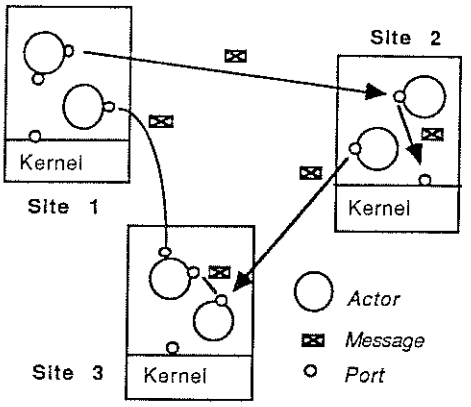

Figure 1 : CHORUS Basic Concepts

2.2. Message driven execution

One of the distinctive features of CHORUS is its message-driven model of computation. An actor operates as a sequence of execution granules, called processing-steps. Each message received by an actor triggers the execution of a processing step (basically a procedure) that ends in the sending of zero or several messages: each actor processes only one message at a time and its operation appears as a succession of processing-steps.

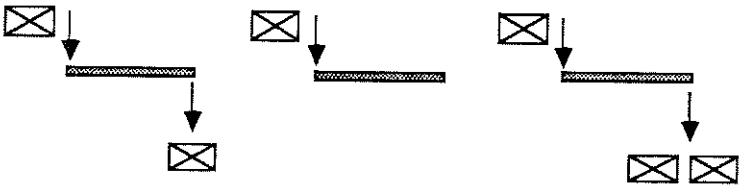

Figure 2 : Message driven execution

CHORUS distinguishes clearly between an actor's processing and communication phases: in the course of a processing-step an actor can not communicate with the outside; messages resulting from the processing-step are sent all together at the end of this step. The actor can then process the next message. This clear structure eases the design of fault-tolerant mechanisms, because checkpoints may be naturally placed at the beginning of processing-steps [Banino 82, 85a].

The succession of processing-steps carried out by an actor depends, on the one hand, on the messages it receives at its ports and, on the other hand, on the selections defined by the actor; a selection is a list of connections consisting each of a source port and a local destination port; only a message received through one of

these connections may trigger the following processing-step. In addition, message selection relies on the priority of ports and the arrival order of messages.

So, as ports can be dynamically associated with processing-steps, an actor is structured as some kind of an *active monitor* where ports are the access points to the monitor procedures, and operations on the data of the monitor are processing-steps.

2.3. Operating System Overview

The **CHORUS operating system** [Guillemont 82] consists of a kernel and a set of system actors. Located at each CHORUS site are the CHORUS kernel and at least those system actors which manage the local resources. The kernel implements the logical CHORUS machine. It schedules execution of processing-steps by actors, and handles their local communications. Moreover it entirely hides the physical machine by transforming exceptions, such as interrupts and errors, into messages addressed to ports. The kernel is small, all the system resources being handled by the system actors. Basic CHORUS system actors manage devices, communication links, files, actors, ports, names, etc.

For programming convenience, interactions with such system actors are embedded into synchronous system calls, in a system interface library. This set of system calls includes the entire UNIX system V interface. Actors may be created and destroyed both dynamically and remotely, using the standard UNIX calls (fork, exec, exit, kill, etc.). CHORUS offers a distributed file system. Location-transparent file access is done through the standard UNIX calls (open, read, write, close, etc.). The specific CHORUS facilities (ports, groups, IPC, naming, execution control, etc.) are available though additional calls. Some of them will be presented in the next sections.

3. Ports and Groups in CHORUS

In message oriented systems, messages are sent to some sort of "message receptacles". The different ways these receptacles are bound to processes introduce different message addressing methods. The most common ones are: direct addressing, or process-to-process addressing (V-Kernel [Cheriton 84] for example), and port-to-port or functional addressing (Accent [Rashid 81] for example).

In CHORUS, messages are exchanged through **ports**. An actor may own more than one port. A port designates an access point to a *service*, or to an *operation*, and not necessarily a *server*. Particularly, in CHORUS, ports can exist independently of actors and the same port may be, successively and dynamically, associated to different actors. This method allows dynamic reconfiguration (achieved by keeping a stable port interface while changing the actors) and makes the synchronization and the server structuring easier.

Logically, a port is a message queue which may be in one of the following states:

closed the port is idle: the queue is empty; it can not be used for message exchange; messages sent to a closed port are lost.

open the port is active: it can receive and send messages; only one actor, the one which opened the port, may consume messages sent to that port.

When it is created, an actor owns some open ports. In addition, an actor can *create* other ports. In order to use a port to send and receive messages, an actor must *open* it, and when the actor doesn't need the port any more, it may *close* it. A port

can be created by an actor and successively opened/closed by others, although it can be opened by only one actor at a time, i. e., by its current owner.

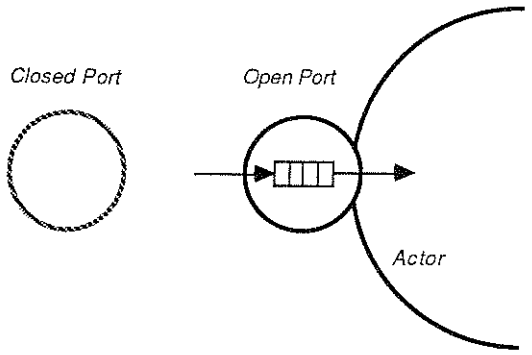

Figure 3 : Port States

In order to achieve symmetrical designation of senders and receivers, messages are also (logically) sent through opened ports: a message flows from a source port to a destination port, and not from a source actor to a destination port. As a consequence, the only communication environment of an actor is a set of ports.

In addition, ports may be associated into **port groups.** This concept, which allows broadcasting and functional addressing of services with several equivalent access points, is discussed in detail in the next sections.

3.1. Port Management

We overview here the system calls provided for port management. Within an actor, ports are designated by **contextual names** (ranging from 0 to n). These contextual names are referenced below as *pgd* (for Port or Group Descriptor). Port naming will be discussed in detail in section 5. The system call:

creatport → pgd

creates a new port in the closed state and returns a *pgd* for that port. In order to use a port, an actor must open it:

openport (pgd, priority)

opens the port designated by *pgd*; the priority of an opened port sets the priority of the processing of messages received on it. Once opened, a port may be used by the actor in order to send or to receive messages.

closeport (pgd)

closes the port designated by *pgd*; only the owner of a port can close it. Messages received at that port and not yet processed by the actor are destroyed. The port returns to the closed state and can be opened by another actor. In this way, *receive rights* on ports may be transfered from actor to actor (an actor having the *receive right* on a port is the owner of that port).

dlport (pgd)

deletes the port designated by *pgd*; that port must be in the closed state.

An alternative for deleting ports is to create them with the *self-destruction* attribute: when an actor is destroyed, all its opened ports are closed and those which have this attribute set are also automatically destroyed. This feature allows the cleaning up of ports that are not used for dynamic reconfiguration.

For an actor, there is no difference between local and distant communication. When an actor closes a port, another actor can open it, possibly at another site. In this case, the port migrates, but this fact is completely transparent to other actors (in particular, the port will continue to belong to the same groups - see 3.5). The system deals dynamically with its new location (see 5.2).

3.2. Our Experience in using Ports

Basically, the CHORUS system itself is built using the same set of mechanisms as used by applications. Most system services are implemented by actors, outside the kernel: the network server, the port/group server, the file server, the actor server, several driver servers, etc.

These servers use ports in different ways: one port by operation, one port by set of operations of the same priority, one port by set of related operations, one port by group of servers with which a dialogue is set up, etc. In each case, we found that the use of different ports, by the same server, and the dynamic association of ports to processing-steps, are very helpful and powerful facilities for server structuring and synchronization.

For instance, let us illustrate the case of dynamic association of ports to actors and port migration facilities, by an example taken from the CHORUS process management strategy. In CHORUS, an actor can call the function *exec* (like in the UNIX system), even remotely. After this call, all the ports of the actor are associated with another code and context, possibly on another site. Among the ports of an actor, a special one (hidden to the code of the actor) identifies it: the *signal port*. UNIX signals are transformed into *signal messages* sent to the *signal port* of the receiver of the signal. If an actor performs a *remote exec*, its *signal port* will migrate to another site and it will continue to receive signals sent by its partners; for them, migration is transparent. It has also been shown [Banino 82] that port migration may be used to build replicated servers which are dynamically reconfigured, when faults occur, in a transparent way to their clients.

In the next sections we illustrate the port grouping feature.

3.3. The role of Groups in Distributed Systems

There are many cases where it is desirable to address a group of entities. For example, in UNIX, a signal sent to a process group allows a user to destroy all the processes belonging to a *pipe line* in one operation. In a distributed system, message *broadcasting* - or generic addressing (one-to-N addressing) - and the possibility to materialize the logical concept of *distributed service* with several equivalent access points are critical issues. We illustrate these ideas by three examples.

For instance, if a client wants to open a file F and does not know which server manages it, it can broadcast an "open" request to all the servers; only the server that knows the file will answer to the client. This is a one-to-N request protocol with only one reply.

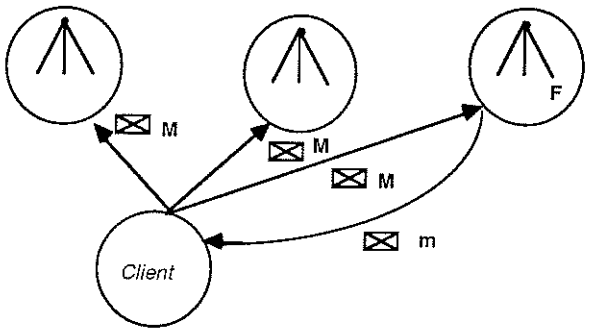

Figure 4 : File F localization using Generic Addressing

Message broadcasting is also useful to implement replication. If the file F is replicated in several servers, the client can also broadcast an "open" request to all servers, execute a majority voting on the received replies, and choose one server among those wich manage the most recent version of F. This is a one-to-N request with M replies protocol.

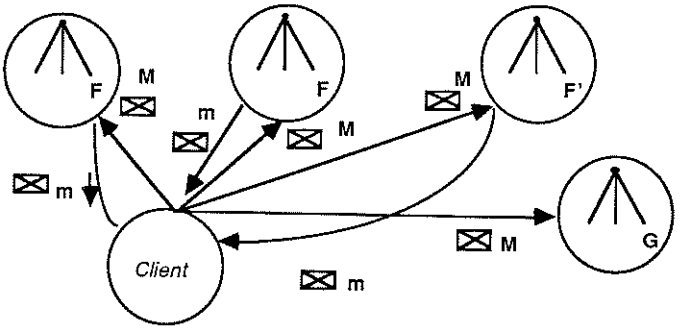

Figure 5 : Majority Voting to Open the replicated file F

Finally, a client may request a service and be unaware of which server performs it. For instance, the client wants to print a file without being aware of which printer is used. We call this form of addressing, functional addressing or "one-to-(one-among-N) addressing".

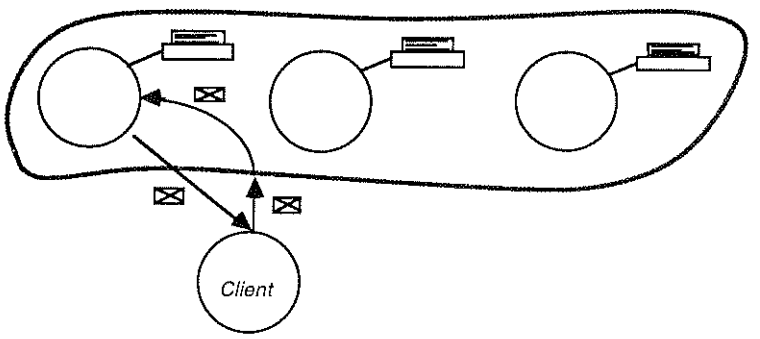

Figure 6 : Printing a file using "one-to-(one-among-N)" Addressing

In summary, group addressing provides more functionality than one-to-one addressing (port-to-port addressing for example) because clients need not know the actual membership of the group or they are not able (or they do not want) to choose a member of the group. Moreover, broadcasting is cheap in local area networks.

The most natural way to introduce the group concept in CHORUS is to support port grouping.

3.4. Groups of Ports in CHORUS

A group of ports is an arbitrary set of ports: it may be empty or it may include opened or closed ports; the ports of a group may be located on any site. A port may belong to zero or more groups, but a group may not belong to another group (in order to avoid cycles). Within an actor, groups are designated by **contextual names,** exactly like ports (*pgds*, see 3.1).

This relationship between ports of a group has only a communication significance. The life of a port is not related to the life of the group (or groups) it belongs to. Mainly, the role of port groups is to allow more powerful communications and addressing than deterministic one-to-one port addressing.

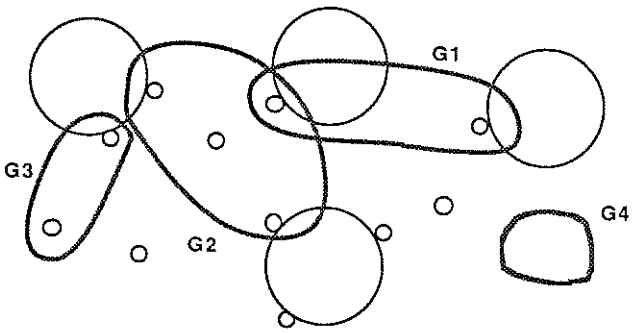

Figure 7 : Ports and Groups of Ports

3.5. Group Management

Groups may be created and destroyed; ports may be inserted or removed from a group. The operations available on groups are detailed in this section.

creatgroup → pgd

creates a new empty group and returns a *pgd* for that group.

dlgroup (pgd)

deletes the group named by *pgd*. An alternative for deleting groups is to create them with the *self-destruction* attribute: when such a group becomes empty, it is automatically destroyed and, conversely, it is the only way to destroy such a group.

So, in CHORUS, users can choose the way a group is destroyed. If a group is a set of ports belonging to an application, it is natural to destroy it automatically when the application ends. However, if the group represents a set of ports of a distributed service S, it must continue to exist even if it becomes empty; in fact, later, another server belonging to S may become active. Here our options are different from those taken by systems that systematically destroy the empty groups: UNIX and the V-System [Cheriton 85] process groups for example.

The last operations on groups control the group membership:

portinsert (port pgd, group pgd)

portremove (port pgd, group pgd)

portisin (port pgd, group pgd) → { yes, no }

respectively insert *port* in *group*, remove *port* from *group* and check whether *port* belongs to *group*. Even if the state of a port changes (open, close), or if the port migrates, it remains in the same groups.

Finally, the system offers some pre-defined groups [Legatheaux 86]: pre-defined groups are neither created nor destroyed, they always exist. The system provides for each user a set of pre-defined groups.

Groups of ports play an essential role in the system. We will discuss our experience on groups use after the presentation of the communication primitives of CHORUS.

4. The CHORUS Inter-Process Communication Facility

A message is a sequence of bytes of arbitrary length; the structure of a message is left to the application level. Messages are exchanged among actors through ports and groups. There is no a priori connection between ports: a message may flow from any source port to any destination port or group, provided that the sender knows a contextual name of the port or the group. The communication mode is the *datagram* or connection-less: once a message is sent, there is no guarantee of eventual delivery.

A message in transit has the following components:

Pe, Pr, Ns, Data

where Pe designates the source port, Pr the destination port or group and Ns the **sequence number** of the message; the sequence number is a global and unique (not reused) stamp that identifies the message forever. The receiver of a message has read-only access to these components of its header, through specific system calls.

We review below two types of communications disciplines: synchronous and asynchronous, and two point of views: the sender and the receiver ones.

4.1. Sender point of view: Asynchronous Communication

putmsg (message, source pgd, destination pgd, addressing mode)

places *message* in the actors transmit queue; the effective transmission of the message is delayed until the end of the current processing-step (see 2.2). The message will be sent from the *source* port (which must be an open port of the actor) to the *destination* port or group.

If the destination is a group, the *addressing mode* parameter is considered. Basically, two values are available:

- **broadcasting**: the message will be sent to all the ports in the group (even if it belongs to the group, the *source* port will never receive a copy of the message);
- **functional**: the message will be sent to only one port in the group, this port being selected in an arbitrary way (the *source* port will never be selected even if it belongs to the group).

If a message M = { Pe, Gr, Ns, Data } is sent to a group (Gr), each receiving port (Pr) receives the message { Pe, Pr, Ns, Data } i. e., groups and addressing modes are transparent to receivers.

4.2. Sender point of view: Synchronous Communication

call (message1, source pgd, destination pgd, addressing mode, delay) → message2

performs a request/reply protocol: *message1* is sent from the *source* port to the *destination* port or group; the *addressing mode* must be such that the message is sent to only one port, i. e., *mode* must be the *functional* one if *destination* is a group. The call returns in one of these cases:

- the reply message has been received (*message2*) - i. e., a message with the same *sequence-number* as the request message (see below);
- *delay* has elapsed and the expected message has not been received.

4.3. Receiver point of view

CHORUS does not provide any primitive for message receiving. When a processing-step ends, the actor implicitly enters in the *receiving state*. The next processing-step will be invoked automatically whenever the next message is selected. This state is blocking, a **time-out** mechanism being used to prevent an endless blocking.

The receiver of a message can send it using the *putmsg* call but, if it is involved in a request/reply protocol, it often uses one of the two following calls:

putfwd (message, destination pgd)

forwards the message, i. e., it has the same effect as *putmsg* except that the source port of *message* is not changed, and

putreply (message)

that will reply *message* to its original sender; this is equivalent to a *putmsg* of *message* where source and destination of the message are swapped.

The *sequence-number* of the message is not modified by these calls and, therefore, this number is like a request/reply transaction identifier.

4.4. Our Experience in using the CHORUS IPC and Port Groups

In systems based on the client/server paradigm, as CHORUS, the request/reply protocol is a very powerful facility. *Call* represents the point of view of the client in such an exchange, *putfwd* and *putreply* represent the point of view of the server in the same protocol. Basically, all the system interface procedures (stub procedures) transform their calls in a request/reply protocol with the server concerned by the system function.

Asynchronous send is mainly used by clients that wish to trigger an asynchronous event, or by multiplexed servers engaged in more complex protocols (see section 6).

CHORUS port/group and file servers use broadcasting to locate migrating or duplicated entities. Even when broadcasting is not reliable, our experience confirms that broadcasting is the simplest way to locate such entities. Moreover, more reliable broadcasting protocols can be easily built upon these simple mechanisms (as proposed in [Cheriton 85]).

UNIX process groups are easily implemented with CHORUS port groups. A group of actors G is implemented as a group of *signal ports* (see 3.2). A signal sent to G is then a *signal message* broadcasted on this port group. In the same way, the system builds the group of all the children of an actor, the group of all the actors attached to the same terminal, etc. These groups are all fully distributed.

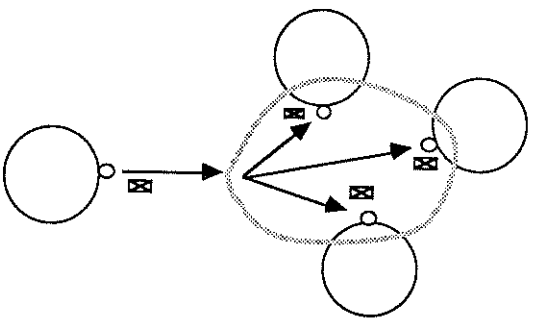

Figure 8 : Signal Broadcasting upon a Group of Actors

Further useful examples of functional addressing use are:

- **Load Balancing:** in CHORUS, load balancing among several equivalent servers can be implemented in the following way: the service is represented by a group of ports (one for each server). The clients address that group in the *functional addressing mode*. Each time a server is selected and receives a message, it removes its port from the group, processes the client's request, replies and re-inserts its port in the group.
- **Binding:** a client wishing to establish a continuous dialogue with any server among a set of equivalent servers first calls the servers group in the *functional mode* and, in the next calls, uses the source port of the first reply.

So, port groups and the *functional addressing mode*, taken together, provide a concept allowing the implementation of communication paths with N senders and M alternating receivers.

5. Naming and Binding

Having presented ports and groups and discussed their use, we show, in this section, how these entities are named and how actors acquire their names.

Generally, operating systems use two kinds of names: symbolic names or user-level names (character strings such as file path-names, user names, etc.) and low-level or internal names (such as process numbers, file descriptor numbers or file internal numbers, etc.). Concerning names, distribution introduces some new requirements:

- naming must be transparent to location; in particular, the rules for structuring the user-level name space must be enforced by administrative and not by location constraints;
- it must be possible for an entity to migrate or be replicated while keeping its name;
- the relationship between a name and the entity it denotes must remain constant even in the presence of faults;
- the naming system must allow the simultaneous use of several sites by one application;

These requirements lead distributed operating systems to build the naming system on top of *global and unique names*, known as UIDs (for Unique IDentifiers), see [Leach 82] for example.

5.1. Naming Layers in CHORUS

CHORUS uses UIDs to designate, at low level, ports and groups. These names have the following structure:

Creation site name	Type	Unique stamp within the site

Figure 9 : The CHORUS UID Structure

Names of sites are statically allocated and are unique within a CHORUS network. The stamp denotes the logical creation time of the entity and is unique within the creation site. These names are globally unique both:

- in space: a given UID always references the same entity even if this entity migrates,
- in time: if an entity is destroyed or lost, its UID will never be re-used to designate another entity even in presence of faults.

In earlier versions of CHORUS, UIDs were directly available to clients. As CHORUS became a full distributed development and execution environment, portability, protection and binding requirements (see [Legatheaux 86]) led us to hide the UIDs behind contextual names. In the current version of the system, actors only know contextual names of ports and groups. UIDs are hidden and only known by the system (kernel and system servers). This contextual address space is protected and typed as the message addressing spaces of Accent [Rashid 81] or DEMOS/MP [Powell 83]. The naming layers present in the system are those of figure 10.

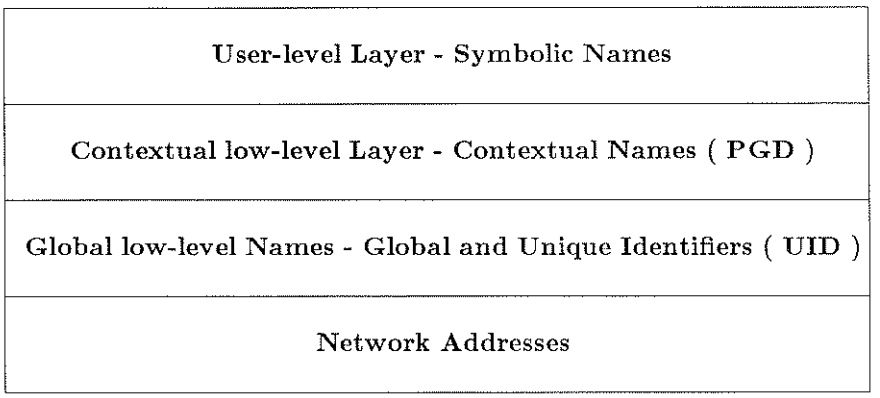

Figure 10 : Naming Layers in CHORUS

In CHORUS, four layers of naming co-exist: network addresses and UID within the system, contextual names within actors and symbolic names at the user level; the system is able to compute all the mappings between names from user level names to network addresses (see the following sections).

5.2. Port Localization Algorithm

For performance and implementation reasons, the dynamic binding to ports and groups representing system services (system servers or kernel services) is achieved by using *static global names* for those ports and groups, i.e. names with stamps known at compile-time. Interface procedures and system servers can compute system port UIDs from their pre-defined stamps and the name of the site where they reside. As these ports never migrate, static port UIDs always contain their current location. This property is essential to avoid recursion in the following algorithm.

In every site a port/group server manages ports and groups. Each of these servers knows the current location of each port that has been created on its site and has migrated. When a network server, or another port server, needs to locate a remote port, it asks the port server that has created the port for its actual location. In order to speed up port addressing, each network server maintains a cache of the location of the last referenced remote ports. Finally, when a site crashes, broadcasting is used to locate the ports it has created.

The protocols used to update the location tables and to request for port location are based on the use of static ports and groups. Moreover, these protocols are very simple because the system considers location information as hints (as a "last solution", broadcast is always available to localize a port).

5.3. Contextual Names, Protection and Binding

Within an actor, **contextual names** reference ports owned by the actor, ports owned by other actors, closed ports and groups. At the same time, two different contextual names known by the same actor always reference two different entities However, as these names can be dynamically acquired or released, the same contextual name may successively reference two different entities. Basically, these names are entries in a table of the actor's context, the message addressing context, that maps contextual names (*pgd*) into global and unique names (UID).

The use of contextual names involves two issues: protection and binding (name exchange). We present below how these two issues are addressed by CHORUS.

Protection, in CHORUS, is based on the following scheme: each actor in the system runs in charge of a given user (generally the user who loaded it); every passive entity (port, group, file, etc.) belongs to an user (generally the user who created it); each entity is associated, at creation time, with a set of protection attributes which specify who may perform which operation on the entity.

Namely, at creation time, a port receives a set of protection attributes which indicate which actors may *open* or *destroy* that port; and a group receives a set of protection attributes which specify which actors may *modify* the group (insert or remove any port and destroy the group) or *self-modify* the group (insert or remove only its own ports).

These protection attributes and the *auto-destruction* attribute are implicitly specified by a contextual *mask* that the actor may set and that is associated with the subsequent created ports or groups.

A CHORUS contextual name differs from a *capability*: its knowledge is required, but not sufficient, to manipulate the entity it denotes. It only shares the properties of a capability under the point of view of the *message sending right*: the knowledge of a contextual name of a port is sufficient to send messages to that port. However, to get the *receiving right* upon one port, an actor must have the right to open it; and to get the *receiveing right* upon a group, an actor must have the right to belong to the group. Port and groups protection attributes are static and can not be acquired or released dynamically.

Let us now examine the binding and exchange of contextual names. In systems with typed messages, as Accent [Rashid 81] for example, this typing may be used to exchange contextual names among processes. In this case, name exchange and binding does not require special facilities. Clients use this general mechanism to

implement their own policies.

In CHORUS, messages are not typed but the system offers some facilities for inheritance and dynamic acquisition of contextual names. We present now these facilities:

1) Automatic Sender Identification

The system records in each message header the UID of the sender. The operation:

msgsrc (message) → pgd

allows an application actor to get a contextual name that designates the source port of *message*.

2) Inheritance of Contextual Names

When an actor is created, it inherits the contextual names known by its parent. The system allows the parent to hide some of these names to its children. This facility, taken together with the possibility for the parent actor to create ports that its children will open, may be used to load and bind together the members of a distributed application; this scheme is, in some way, analogous to the scheme used by the UNIX command interpreter (the *shell*) to load pipe-line commands if pipes are replaced by ports and pipe-lines by nets of actors (see [Legatheaux 86] for details and illustrative examples).

3) Dynamic Binding using Symbolic Names

CHORUS file servers have been extended to support an elementary name service. Ports and groups may be associated with symbolic names through special nodes of the file tree which are called *port/group nodes*. A port or a group may be thought of as representing a service and the associated symbolic name is the user level name of that service.

For instance, the actor in charge of managing a laser printer has a port which receives the message requests for printing; that port may be named */printers/laser*. When this actor starts its operation, it associates that port with this user level name by calling:

setname ("/printers/laser", pgd)

A client of the laser printer may get a contextual name of the same port by calling:

getname ("/printers/laser") → pgd

The association between a port or a group and a symbolic name is independent of the state of the port or the group: for example, an actor may get a *pgd* of a closed port and open it. It is the client's responsibility to ensure the consistency of names.

To sum up, binding can be the result of communication, of inheritance, or of the use of a name service.

The name service is integrated with the file service; which we shall now discuss.

5.4. Distributed Interpretation of Symbolic Names

As we have stated before, distribution requires a naming system with location transparency (i. e., global names) and freedom to structure the user level name space by administrative decisions, independent of location constraints. These requirements are satisfied by systems that offer an unique name space, as for example an unique name tree (Grapevine [Birrell 82], LOCUS [Popek 81]).

Building such a name space requires replication of directories and/or files. To avoid such complication, many systems offer name spaces that are built by interconnecting (i.e. by introducing some kind of special link among) "local" name spaces. The most popular approaches are known as the "network root" approach (Apollo/DOMAIN [Leach 83] for instance) and the "remote mount" approach (NFS [Sandberg 85] for instance). These intermediate approaches are more or less suitable; this depends mainly on the context of use of these systems.

CHORUS also takes an intermediate approach, but is open to evolution: the CHORUS file and naming system is a collection of interconnected local UNIX-like (extended) file systems. In addition to traditional "local" file access, CHORUS offers new facilities which are a sound basis on the way to an *integrated* distributed file and naming system. These facilities are based on two extensions in the UNIX-like "local" file systems: port/group nodes and distributed interpretation of path names based on symbolic forwarding.

As we have already said, port/group nodes allow the registration in the file tree of an access point to a server. When a file server interprets the path-name contained in a client's request message, each time it encounters a port/group node (which needs not be the final name of the path), it replaces the initial path-name by the yet uninterpreted part of it and forwards the client's request message (using *putfwd*, see 4.3) to the corresponding port or group (if the node represents a group, the addressing mode is the *functional* one). If the message is forwarded to another file server, the name interpretation continues in that server and, when finally the path-name is exhausted, the concerned server may execute the client's request and reply (using *putreply*, see 4.3).

One of the results of this distributed name interpretation is to allow the introduction of arbitrary links between a leaf node of a file server and the root node of another one. This is approximatly the function of a "remote mount". This facility is used to interconnect the several "local" naming trees of a CHORUS system in the CHORUS naming forest.

5.5. Naming Conventions of the CHORUS naming forest

This section shows how the above protocol has been used in CHORUS to build a distributed file system allowing access transparency (access to "local" and "remote" files is equivalent), naming transparency ("local", "remote", as well as "global" file names are syntactically equivalent) and providing network-global names for resources. This is achieved through an access method based on the use of ports to interface file servers as well as some precise naming conventions and the replication of directories.

Every request concerning an open file (read, write, etc.) or a named resource (open, creat, getname, etc.) is, in CHORUS, sent to a file server port. For this access, actors always know the file servers ports associated with their root, current

directory or open files [Armand 86]. This access method allows an actor to have its root, current directory and open files managed by different file servers ("local" or "remote"). It also allows, for example, that child actors, even remotly created, share root, current directory and open files with their parents.

In order to allow an actor to access every file, managed by any server, each file server tree contains a directory */fs* (for "file servers") which gathers the identifications of all file servers in the system; these directories are identical in all servers and their leaf nodes are port/group nodes which reference the *root port* , i. e., the port able to interpret path-names starting with "/", in each file server. Therefore, each path-name which begins by */fs* is a network-global path-name: the interpretation of such a path-name will be automatically forwarded to the same server (see the above paragraph), whatever the server which begins the path-name interpretation is, and the path-name will always designate the same entity.

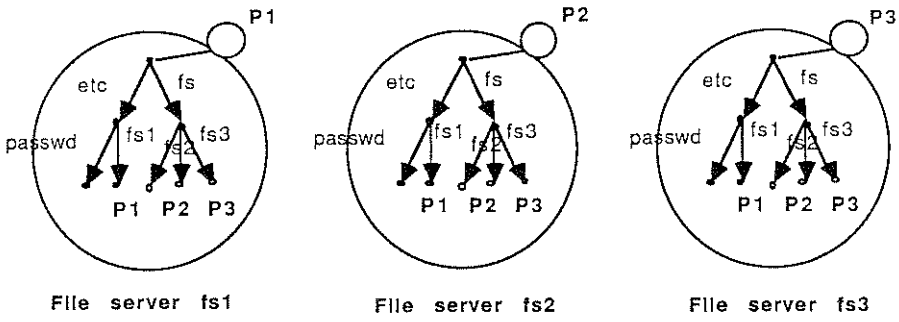

Figure 11 : Configuration of the CHORUS Naming Forest

This forest configuration may be interpreted in two dual ways:

1/ The */fs* directory in each tree contains the directories where other file system tree roots are *mounted remotely*; however, these mounts are automatic and reflect the integrated aspect of CHORUS.

2/ All the naming trees are interconnected through a kind of *network root, /fs* (introduced with no syntactic exception);

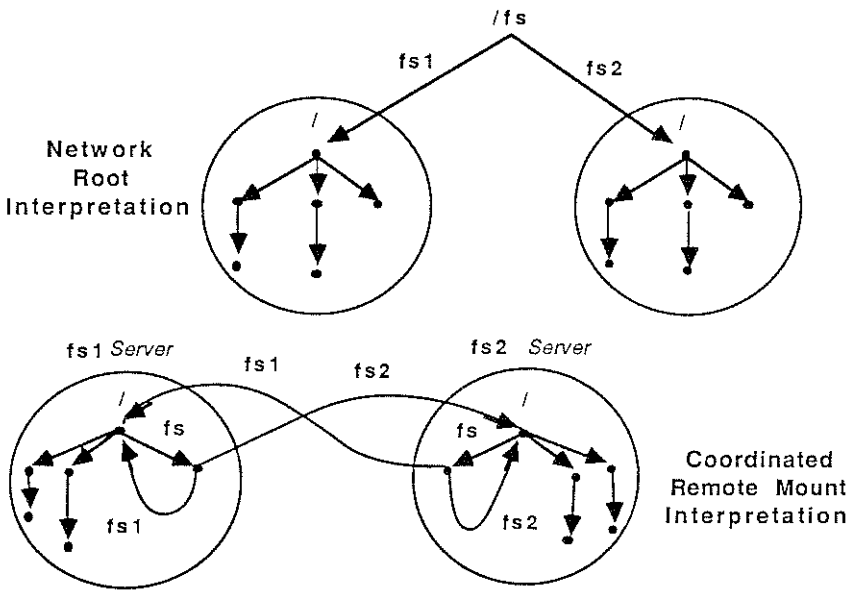

Figure 12 : The dual interpretations of the CHORUS Naming Conventions

This configuration allows the mapping of network-global symbolic names to the name of the port of the server being able to interpret them. Why did we choose it? Because it is rather open to evolution on the way to a truly unique name space (showing an unique name tree). In fact, this scheme is compatible with the introduction of more sophisticated facilities to locate (replicated) named resources as we show below.

Let's name *s-absolute* (absolute for a server) a path-name beginning by "/"; the interpretation of that path-name begins at the root of the tree managed by the file server which interprets it. Two identical s-absolute path-names do not necessarily designate the same entity; they do if they are interpreted by the same server or if the entities referenced by that path-name in two servers are identical, i.e. duplicated. Thus, a s-absolute path-name does not designate one entity but a class of entities with, possibly, several different instances in different servers. By the moment, the equivalence of the different instances is the client's responsibility. However, the client is able to distinguish between the different instances.

For example, in a three sites system (as illustrated on figure 11), the file */etc/passwd* is present in every file server tree. The s-absolute path-name */etc/passwd* designates the class of the files which define the users of the system; this class gathers three instances of the file, respectively designated by */fs/fsm1/etc/passwd*, */fs/fsm2/etc/passwd* and */fs/fsm3/etc/passwd*. On the other hand, */fs/fsm1/etc/passwd* always designates the same file whatever the server which begins its interpretation is.

So, our naming conventions support the evolution to full replication of files and currently they also enable clients to manage (manually) their own duplication or back-up facilities. Moreover, the use of symbolic links (which have also been introduced in the CHORUS file servers) and network-global names allow clients to

rely on a "s-global vision" of names, rather independent of the servers really used. We are also planning to introduce some special-tailored facilities to help the management of "read-only" replicated files as "/etc/passwd" for example.

The conventions introduced may be summarized as follows:

- a path-name beginning with */fs* is a global path-name;
- a path-name beginning with "/" but not with */fs* is a s-absolute path-name, i.e. relative to the server which interprets it;
- every s-absolute path-name of a replicated entity is also a global path-name for the entity; in particular, all the names of the form */fs/server* are global because of their replication.

This ends our discussion on some issues which concern directly the design of the current version of the CHORUS system. The last section of this paper is more dedicated to an experiment that was run on top of the system. This work was stimulated by some expression problems that our experience in distributed system programming has pointed out. Its result is the design - and implementation - of a new structuring concept for CHORUS: the activity [Rozier 86, Banino 85].

6. Distributed Control: The Activity Concept

6.1. Definitions

In a system like CHORUS, a distributed application is basically a set of autonomous processes, with separate address spaces, interacting with each other by message passing. Generally, a complex application is structured into layers of *distributed services*. A distributed service involves a set of *servers* and two kinds of *protocols*. The servers are distributed among the system and manage the *resources* of the service. The two kinds of protocols are the following:

- *access protocols*: the service provides a number of *functions* to the outside world; to each of these functions is associated an access protocol, i.e. the way in which the function is invoked. Such protocols are very simple, and are often "Request/Reply" protocols: the client sends a request to an access point of the service (a port or a group), and waits for an answer. For programming convenience, such protocols are generally embedded into standard procedures, or *stub* procedures (as for the CHORUS system calls - see 2.3).
- *function protocols*: the execution of a function of a distributed service involves the cooperation of different servers. These servers provide the *basic operations* on the resources. The way in which these basic operations are combined to execute the functions is defined by protocols which can be very complex, and are called here *function protocols*.

Let us take the example of our distributed file system. The *resource* is here a set of file trees, distributed over some set of sites. On each site, a local tree is managed by a *file server*. The *functions* offered by this distributed service are for example: open a file, read n bytes from a file, etc. They can be applied to any file in the network. Each of these functions is invoked by a Request/Reply protocol: the *open* system call embeds the sending of an open request to a server (the server which manages its current root or the server which manages its current directory, depending on the structure of the name) and the waiting for an answer. This answer is not necessarily sent by the same server. In fact, the opening of a remote file needs

the cooperation of several file servers (see 5). The *basic operations* provided by a file server can be seen as: look if a file - defined by a pathname - is local, open a local file, etc. The *open function protocol* includes the forwarding of the request message by the different servers.

In that case, the function protocol is rather simple. But such protocols can be very complex; for example, they generally include some parallelism; they often involve heterogeneous servers. In the next sections, we focus on the problem of the expression and execution of such protocols.

6.2. Problems and Objectives

Our experience in the design of several distributed services in the CHORUS system led us to point out three general problems:

- the communication primitives found in existing systems are not powerful enough for many parallel protocols. Let us take the example of a client which needs to invoke several operations in different servers. For efficiency reasons, the invocations may be done in parallel. The different answers must be collected. Moreover, because we are in a distributed system, the client must cope with particular situations: a message may be lost, a server may crash or take a very long time to answer, etc. The client must decide when it stops to wait for answers; it will also have to cope with answers which return after this resuming point. In CHORUS and other systems, the communication primitives are too basic for a good description of such behaviors.
- the description of a function protocol is scattered into the different servers involved in the function. For example, in the distributed file system, the protocol which defines the open function is described in the code of all the file servers, and eventually other servers (name servers,...). When a modification must be done on the protocol, existing servers must be modified and replaced. Moreover, the description of the protocol is a part of the description of the basic operations: these a priori independent descriptions may be affected by the modification. Finally, modification is difficult because the protocol is not globally documented.
- at run time, the execution context of the protocol is scattered into the contexts of all the servers. When we need to trace the progress of the protocol, we need to trace all the servers, and then to modify them.

Our approach to solve these problems is the following: a function protocol must be represented as a new type of system entity, which we call an **activity.** An activity is described as a separate algorithm. This description is independent from the description of the basic operations in the servers. Moreover, the tools used for this description must be adapted to parallelism. An activity instance owns its own context, defining the progress of the protocol. The activity appears at run time as a *process* which migrates in the distributed system to invoke the basic operations in the servers.

In the next section, we show how the activity concept has been introduced into the CHORUS system. The architecture of the system has not been modified at all.

6.3. Activities in CHORUS

6.3.1. Notes on CHORUS

In CHORUS, an **activity** appears naturally: it is a graph of processing-steps executed by actors. The *basic operations* introduced above appear explicitly in the actors programs, as processing-steps. In CHORUS, a processing-step is always triggered by a message. Message-passing is the only control tool offered by the CHORUS architecture. The messages are identified, can be very large and have long lives: a given message can be reused during the different phases of a protocol.

The main idea of our experiment derives from these remarks, and consists of implementing the activity as a message, called the **activity message.**

6.3.2. The Activity Message

Our approach consists of providing a programming method for the activity and mechanisms for the control of its execution. The description of the activity is made in terms of a program, the **activity template.** We will see later the formalism used to program activities. At run-time, an activity is represented by a message: the **activity message.** This message holds the description of the computation, i.e. the path of processing steps to be performed, and its current context. The activity message is transmitted from actor to actor, triggering in each of them a processing-step of the activity. When all of the processing-steps required by the activity has been completed, it terminates.

Logically, an activity message has three distinct parts: control, context and data. The control part contains the description of the path of processing-steps to be executed. The context part holds the information representing the identity of the activity message and the current state of its progress. The data part is the only part of the message which can be seen by the various processing-steps which will be executed in the activity (see below).

6.3.3. Execution Model

When an activity message is received on a port, the actor executes a processing-step, as it does for any message received. The activity message is seen as a normal message: the actor accesses only the part of the data of the activity message which has been structured as a normal message. At the end of the processing-step, the actor only *replies* the activity message. When the message is an activity message, the *PutReply* system call updates the context part of the message and determines the next processing-step of the activity from the control part (*Putmsg* or *Putfwd* has no effect on the activity message). It then forces the activity message to be sent to the corresponding port in the system. This processing (the "interpretation step") is done automatically by the system, and the code of the actor does not have to be modified. This mechanism is shown on Figure 13.

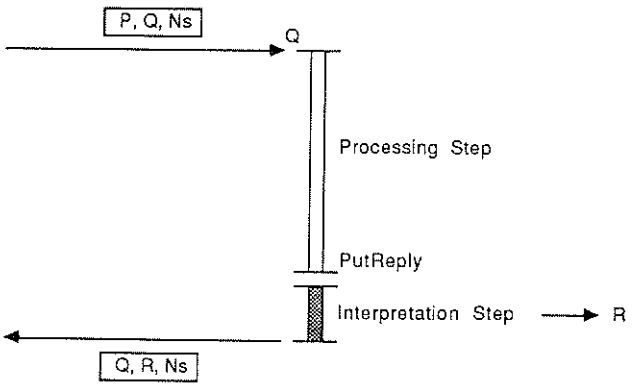

Figure 13 : Activity message processing

6.3.4. Expression Formalism

In the actual implementation, activity templates are written in high level languages (just as are actors). A few specific control structures are added to the language to express activities (see [Rozier 86] for details).

The first additional structure is the basic control structure, the sequential operator. This operator expresses the invocation of a basic operation, i. e., the migration of the activity message to a given port in the system. Its primitive form is the following:

Forward_m (port_name, parameters). This interface procedure causes the activity message to be sent to the "port_name" port in order to trigger the next processing-step of the activity.

Three other control structures, CALL, SPLIT and JOIN are used for the **composition** of activities. They are required to express more complex graphs of processing-steps.

Call (activity_model, parameters) is used to nest activities (the semantics of this construction are similar to a procedure call of another activity).

Split (M, D)
is used to transform an activity into several parallel activities. The syntax of the use of the SPLIT procedure is illustrated on Figure 14.

```
SPLIT (M, Delay) IN
 sub-activity_1 //
 sub-activity_2 //
    ...
 sub-activity_N
ENDSPLIT ;
```

Figure 14 : The SPLIT construction

By the SPLIT construction, the "N" sub-activities are processed in parallel. **JOIN** is used to merge the results of the parallel activities. When a JOIN appears in the code of a sub-activity, the corresponding activity JOINs the main activity. The presence of a JOIN call in the sub-activity programs is optional: some of the sub-activities may have no need to join. In the parameters of the SPLIT call, the programmer gives the minimum number (M) of sub-activities that must JOIN before the main activity resumes and also specifies a delay (Delay) for the success of the "JOIN"s. When all the expected sub-activities have JOINed or when the time-out occurs, the original activity continues its progress. The sub-activities which join after the main activity has resumed are discarded. The structure is associated with a two-way interface scheme: at the SPLIT, the main activity can give some parameters to the sub-activities; when JOINing, a sub-activity may return some results to the main activity.

This SPLIT/JOIN mechanism can be compared to a UNIX fork where the number of children is not limited to one, and where children can communicate some results to the father when terminating. It is a convenient tool for expressing parallel protocols [Rozier 86].

6.3.5. Implementation

An Activity message is implemented as an executable program enclosed in a message. The control part of the message is composed of the code itself. The context part of the message is a subset of the static data of the program and the data part contains the rest of the data for the program. The starting of an activity message consists of the initialization of a message from a compiled activity template.

The current version is implemented in compiled PASCAL. In this implementation, all the features of PASCAL are available to express the control of an activity.

The activity messages are managed in the system by new system actors (these actors perform the creations of activity messages, the split and join operations, etc.). A very important result of our implementation is that the CHORUS system is not affected: the kernel sees activity messages like ordinary messages; activity messages respect the protection rules in CHORUS.

6.4. Conclusion

The activity concept described above provides the system designer with a convenient mechanism for building distributed services. Its most important feature is the two-fold way of expressing a distributed service: on one hand the description of static basic operation in actors, and, on the other hand, the description of the functions as dynamic activities that will invoke these services. The function protocols are described globally as activity templates. The access protocols to the services have a standard form - initialization of an activity message and waiting for the return of this message - embedded into stub procedures. This methodology improves the legibility and flexibility of the distributed applications. It is in particular convenient for programming fault-tolerant applications [Rozier 86, Banino 85a], where complex protocols are added to the functional code of the applications.

In this experiment, we stand up for the idea that a distributed protocol must be described globally (this idea can also be found in [Betourne 85], for example), and be implemented as an active entity in a distributed system. We have exploited the qualities of CHORUS, which offers a clear distinction between processing and communications, and which have proved to be open enough to allow new execution schemes to be defined on its top. Moreover, the implementation is efficient enough to validate the concepts and allow their use for the design of real applications.

Our future work will include a full experimentation of the mechanisms on large applications. We will also focus on the design of higher level formalisms, where actors programming and activities programming would be more homogeneous. Finally, we think that our implementation principle - "intelligent messages" - can be exploited in other fields, like mailing systems or command languages for networks systems.

7. Summary

The main guide-line of the CHORUS design has been to achieve generality with simplicity. A simple message-based architecture led to the design of a complete distributed system which is simple, modular and open to evolution. Moreover, the performances of CHORUS-V2 validate this approach: on a single-processor, they are comparable to those of a native UNIX.

In this paper, we tried to show that our solutions are simple. In fact they are sometimes too simple, leaving some work to the user. But, as we have emphasized, they are rather open to the development of more specific and user-oriented mechanisms. For example, the study of programming formalisms really suited for distribution, initiated with the "activity" experiment, will be a major issue of our future works.

The CHORUS research work is now mature. It is now becoming an industrial product, supported by CHORUS SYSTEMES, an independent company, which will port the system on several machines, enhance it according to the results of standardization and research results and promote it as a basis for the increasing need of interconnection and distribution of homogeneous and heterogeneous computers.

Acknowledgements

Hubert Zimmermann brought many of the original ideas behind CHORUS. Jean-Serge Banino directed the project at INRIA and gave it much inspiration; Gérard Morisset brought the first CHORUS (Version 1) nucleus to real life.

The current version of the system is the result of a cooperative work where individual contributions are intimately mixed. The CHORUS team includes: François Armand, Bruno Deslandes, Michel Gien, Frédéric Hermann, Pierre Léonard, Azzedine Mzouri, Mario Papageorgiou, the authors of this paper, and is directed by Marc Guillemont.

Bibliography

[Armand 86] F. Armand, M. Gien, M. Guillemont, P. Léonard
Towards a Distributed UNIX System - The Chorus Approach
EUUG, Autumn'86. Manchester, (September 1986)

[Banino 82] J.S. Banino, J.C. Fabre
Distributed Coupled Actors: A CHORUS proposal for Reliability
3rd International Conference on Distributed Computing Systems, Miami/FT Lauderdale, Florida, (October 18-22, 1982)

[Banino 85] J.S. Banino, G. Morisset, M. Rozier
Controlling Distributed Processing with CHORUS Activity Messages.
18th Hawaii International Conference on System Science, (January 1985)

[Banino 85a] J.S. Banino, M. Guillemont, J.C Fabre, G. Morisset, M. Rozier
Some Fault-Tolerant Aspects of the Chorus Distributed System
5th IEEE International Conference on Distributed Computing Systems, Denver, Colorado, USA, (May 1985)

[Betourne 85] C. Betourne, M. Filali, G. Padiou, A. Saya
Distributed Control through task migration via abstract networks
5th International Conference on Distributed Computing Systems
Denver, Colorado (May 1985)

[Birrell 82] A.D. Birrell, R. Levin, R.M. Needham, M.D. Schroeder
Grapevine : An Exercise in Distributed Computing
Communications of the ACM, Vol 25, 4, (April 1982)

[Cheriton 84] D.R. Cheriton
The V-Kernel: a software base for distributed systems
Research report, Computer Science Department, Standford University, (April 1984)

[Cheriton 85] D.R. Cheriton, W. Zwaenepoel
Distributed process groups in the V kernel
ACM Transactions on Computer Systems, Vol. 3, No. 2, (May 1985)

[Guillemont 82] M. Guillemont
The CHORUS distributed operating system : design and implementation
International Symposium on Local Computer Networks, Florence, Italy, (April 1982)

[Guillemont 86] M. Guillemont, J. Legatheaux Martins
CHORUS: a new UNIX for the distribution age
Paper submitted for publication. Currently available from the authors at INRIA, (December 1986)

[Leach 82] P.J. Leach et al.
UIDS as Internal Names in Distributed Systems
ACM Symposium on Principles of Distributed Computing, Ottawa, Canada, (August 1982)

[Leach 83] P.J. Leach at al.
The Architecture of an Integrated Local Network
IEEE Journal of Selected Areas in Communications, Vol 1, 5, (November 1983)

[Legatheaux 86] J. Legatheaux Martins
La Désignation et l'Edition de Liens dans les Systèmes d'Exploitation Répartis
(Naming and Binding in Distributed Operating Systems)
Doctorate Thesis, Rennes University I, France (November 1986)

[Popek 81] G. Powell et al.
LOCUS : a Network Transparent, High Reliability Distributed System
8th ACM Symposium on Operating System Principles, Pacific Grove, California (December 1981)

[Powell 83] M.L. Powell, B.P. Miller
Process Migration in DEMOS/MP
9th ACM Symposium on Operating Systems Principles / OSR ACM, Vol 17, 5, (October 1983)

[Rashid 81] R.F. Rashid, G.G. Robertson
Accent : A communication oriented network operating system kernel
8th ACM Symposium on Operating System Principles, Pacific Grove, California, (December 1981)

[Rozier 86] M. Rozier
Expression et Réalisation du Contrôle d'exécution dans un Système Réparti
(Expression and Implementation of Execution Control in a Distributed System)
Doctorate Thesis, INPG, Grenoble, France (October 1986)

[Sandberg 85] R. Sandberg et al.
Design and implementation of the Sun network filesystem
Usenix, Portland (June 1985)

[Zimmermann 81] H. Zimmermann, J.S. Banino, A. Caristan, M. Guillemont, G. Morisset
Basic concepts for the support of distributed systems : the CHORUS approach.
2nd IEEE International Conference on Distributed Computing Systems, Versailles, France, (April 1981)

[Zimmerman 84] H. Zimmermann, M. Guillemont, G. Morisset, J.S. Banino
CHORUS : A communication and processing architecture for distributed systems
INRIA Research Report 328, (September 1984)