# Chorus:an architecture for distributed systems

J.S. Banino, A. Caristan, M. Guillemont, G. Morisset, H. Zimmermann

INRIA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau,
Rocquencourt
BP105 78150 Le Chesnay
France
Tél 954 90 20

Rapports de Recherche

N° 42

# CHORUS:
# AN ARCHITECTURE FOR DISTRIBUTED SYSTEMS

Jean-Serge BANINO
Alain CARISTAN
Marc GUILLEMONT
Gérard MORISSET
Hubert ZIMMERMANN

Novembre 1980

# CHORUS:

# AN ARCHITECTURE FOR DISTRIBUTED SYSTEMS

J.S. BANINO

A. CARISTAN

M. GUILLEMONT

G. MORISSET

H. ZIMMERMANN

## ABSTRACT

The CHORUS project deals with distributed systems; more precisely, it investigates the impact of distribution on operating systems and on execution of applications.

This report is the result of the first step in this work. It presents successively:

- a synthesis of the main advantages and constraints of distribution,
- a model for the execution of a distributed application, where communication, synchronization, control, etc... is based on the exchange of messages,
- a model for the construction of a distributed application which permits to turn distribution to the best account,
- examples which illustrate various aspects of the architecture.

This report presents also the minimal functions required from a kernel of operating system in order to support execution of such distributed applications.

Le projet CHORUS se situe dans le domaine des systèmes répartis; il étudie plus particulièrement l'impact de la répartition sur les systèmes d'exploitation et les structures d'exécution des applications.

Ce rapport est le bilan d'une première étape de travail. Il présente successivement:

- une synthèse des principaux avantages et inconvénients de la répartition,
- un modèle d'exécution pour une application répartie où la communication, la synchronisation, le contrôle, etc... sont basés sur l'échange de messages,
- un modèle de construction d'une application répartie qui permet de tirer le meilleur parti de la répartition,
- des exemples qui illustrent divers aspects de l'architecture.

Ce rapport présente également les fonctions minimales que doit fournir un noyau de système d'exploitation pour permettre l'exécution d'applications réparties conformes à ce modèle.

TABLE OF CONTENTS

# A/ INTRODUCTION

## 1/ Introduction

The world is distributed. Human and industrial organizations are distributed. In the past, data processing has been highly centralized under the rationale of economy of scale and simplification of applications. The bigger the computer, the cheaper each individual application. The closer the various applications, the simpler their relations, co-ordination and communications (common files, common memory, common time, etc...). Despite of the drawback of centralization (vulnerability, unreliability, complexity, distortion of natural organization, etc...), it is only recently that technical and technological advances in large scale integration and in data-communication have made distributed processing economically feasible. However, centralization has highly biased computer sciences and concepts and tools for installing distributed applications are still missing. Distribution is still a topic for research on which several teams all over the world are presently working.

In the area of operating systems, traditional concepts developed for centralized operating systems are largely inadequate for distributed systems. On the other hand, the basic understanding gained in computer communication still needs to be translated into general concepts for distributed operating systems. It was therefore decided that a specific effort should be made on developing such general concepts on the basis of the expertise gained at INRIA both in operating systems [FER 75] and in computer communications ([POU 73], [ZIM 77]). This gave birth to the CHORUS project which was launched

at the beginning of 1979.

The initial effort in CHORUS was put in getting a better understanding of the specificities of distributed applications. It was felt from the very beginning that distributed applications would, eventually, depart largely from traditional centralized applications and that several steps would be required to make the move to distributed systems. Thus the current definition of CHORUS described in this paper is viewed only as the initial step. Implementation and experiments planned with CHORUS are expected to impact its definition. In addition, some aspects of CHORUS (e.g. recovery) will need to be investigated further. This fitback is considered essential to match the real needs of distributed applications.

Section 2 contains a brief discussion of the specificities of distributed systems as currently percieved by the designers of CHORUS, and which oriented their choices. Section 3 and 4 present an introductory overview of CHORUS which is described in details in sections 5 through 7; this description presents a model for the architecture and has been left completely unconcerned with the problems of programming and implementation, i.e. the notions explained describe neither a language nor a machine. Possible usages of CHORUS concepts are illustrated by examples in section 8. Subjects for further investigations are outlined in section 9. Finally, a glossary of terms used to refer to the concepts of CHORUS can be found in section 12.1 and a glossary of primitives can be found in section 12.2.

## 2/ Specificities of distributed systems

Distributed systems offer a number of potential advantages (availability, protection, independence of equipments, increased power, etc...). On the other hand, distribution imposes new constraints (uncertainty about global state, vulnerability of communications, etc...).

Of course, perception of specific advantages and difficulties of distribution has a major influence on choices made by designers of distributed systems. The following subsections give an indication of how distribution is perceived by the designers of CHORUS.

### 2.1/ Advantages of distribution

Distributed systems offer major advantages in the areas of security, reliability, simplicity, flexibility and power.

#### Security

Distribution allows security to be based on physical domains with well defined boundaries, rather than on logical barriers like in complex shared systems.

A physical system can be dedicated to one user or one function or one set of data. It is much easier then to control which programs access which data.

Sensitive data can be physically located close to their owner who can, for instance, physically unplug his system from the others during critical phases (e.g. debugging).

The sophistication of protection can differ from one system to the other to match individual users requirements at minimum cost.

Protection mechanisms can be based upon the existence of physica domains between which communications can be strictly controlled.

In CHORUS, physical domains (sites) can be decomposed furthe into logical domains (actors) between which the same controls can b applied.

Reliability

In a distributed system, individual systems are physicall independent from each other. Domains possibly affected by a failur have well defined physical boundaries; a hardware failure in one o them has no direct impact on the others. If distributed application are designed to fit any physical distribution, reconfiguration an recovery will be easier to handle. Easy reconfiguration o distributed applications, as well as easy check-pointing will b essential to take advantage of potentially improved availability o distributed systems.

In CHORUS, the elementary active entity, called actor, function in successive sequential steps, which facilitates the definition o checkpoints (see 5.3). Moreover, an actor is a local unit o execution with well defined relations with its environment; thi facilitates reconfiguration (see 6.6).

Simplicity

The decreasing cost of logic permits to reduce the degree o sharing. It becomes economically feasible to have individual system dedicated to specific functions or to specific users. Relation between systems tend to match relations between functions which ar therefore kept naturally simple. In other words, distribution sug-

gests simplicity. This trend should be favoured by presenting interfaces between functions in the form of communication interfaces (i.e. interfaces between communicating systems).

In CHORUS, the communication interface is constituted of ports on which actors send and receive messages; these ports reflect the functions performed by the actors rather than their internal organization (see 5.2).

Flexibility

Interfaces between individual physical systems are naturally much simpler than potentially complex interfaces between logical modules within a centralized system. Concepts and tools developed for distributed systems should facilitate modularity of applications.

In CHORUS, the description of a distributed application in terms of communicating modules gives a unified view of communications whether local or remote. This permits to reconfigure a distributed application without any change in the modules already existing (see 6.6).

Power

Distributed systems can provide increased power through parallel processing, provided one has solved the question of decomposing one task into parallel subtasks with minimum communication requirements. The bulk of distributed systems will support applications which do not fall into this category, i.e. most applications will be distributed for other reasons than increased processing power. It is not clear whether the requirements for both types of distributed systems will lead or not to different designs.

In CHORUS, parallel processing is viewed only as a possible side objective, but in no case as the prime one. However the flexibility of the architecture makes it easy to offload some functions on additionnal processors.

## 2.2/ Constraints of distribution

Distributed systems are faced with specific constraints, mainly related to slowness of communication, errors and failure of components, and difficulties of synchronization.

### Slowness of communication

Communications between elements of a distributed system are several orders of magnitude slower than in a centralized one. In particular, no memory can be shared between individual systems. This imposes to minimize communications between modules, and explicitely group together modules which need high communication throughput. Therefore, communications must be given special attention in distributed systems. They must in particular be clearly identified and not be intermixed with processing.

In CHORUS, the ease of reconfiguring applications permits the user to optimize configurations gradually, according to real traffic measurements.

### Errors and failures of components

Distributed systems will include possibly unreliable components (e.g. transmission lines) and the increased number of interconnected components will lower the probability for all of them being simultane-

ously safe. In addition, no central control will exist to monitor the components of the distributed system and uncertainty about the state of the total system will be the rule. Thus, a distributed operating system should minimize dependencies between individual systems and favour decentralized error detection and recovery by providing each individual system with local protection mechanisms not relying on the well functionning of the other systems.

Moreover, a general purpose error control and recovery mechanism would be overcostly for most applications. Therefore, the distributed operating system should initially provide elementary tools which will be used to build a variety of error control and recovery schemes, tailored to each class of application.

In CHORUS, most of the controls are performed by the kernel using procedures written by the users, i.e. the designers of distributed applications; this leaves a high flexibility in designing the most adequate control and recovery policy (see 5.2 and 5.4) for each case.

Difficulties of synchronization

Slowness of communications along with possible errors and failures, make synchronization particularly critical in distributed systems. Synchronization takes time and reliable synchronization between a number of components may require sophisticated protocols.

Moreover the choice of a synchronization protocol is very much dependent on the nature of the application considered. For instance, synchronization requirements are different in distributed data bases and in process control systems.

Therefore CHORUS does not favour any specific synchronization protocol. It rather offers elementary synchronization tools (based on

the exchange of messages) with which various synchronization policies can be implemented.

## B/ CONCEPTS OF CHORUS

### 3/ Orientations of CHORUS

The objective of the CHORUS project is to develop concepts and tools adequate for supporting development, operation and maintenance of distributed applications. In the long term this will impact operating systems, networking and languages as well.

However, much experience must be gained before all aspects of distributed systems can be adequately covered. Thus the initial definition of the CHORUS architecture focuses on the following points:

* Users are primarily interested in running distributed applications. Therefore, developments of concepts in CHORUS start from runtime aspects, i.e. how to execute distributed applications. In particular, the lowest level of CHORUS does not hide distribution but rather makes it convenient to use, i.e allows easy distribution of an application. The concepts provided by CHORUS in order to describe the execution of a distributed application make a distinction between local (sequential) and distributed (parallel) execution.

* A major advantage users expect from distribution is high flexibility through ease of static reconfiguration. This leads CHORUS to make a clear distinction between the description of an application, its configuration and its mapping onto physical resources for execution.

* A third important aspect in distribution is high availability through dynamic error detection and protection. CHORUS proposes user defined control procedures executed by the operating system; these procedures may be modified from a debugging phase to a runtime phase;

they offer decentralized control mechanisms.

* At last, a major concern inherited from the CYCLADES project i the must for heterogeneity. Distributed systems will be made of variety of individual machines: variety in size and in sophistica tion, variety in computing power, in memory capacity and in technolog and variety of manufacturers as well. Therefore, it would b inappropriate to develop concepts which would necessarily have to b supported by specific hardware or would impose the same constraints o all machines. Distributed operating system concepts have to b hardware independent and highly adaptable to a variety of machines The definition of CHORUS presented below intends to match this goal.

This paper describes the initial version of the CHORUS architec- ture; it presents:

* an architecture to support distributed applications, along wit a method to develop them;

* the specification of an operating system kernel (i.e. the kernel level of CHORUS) running on each machine and able to support execution of such distributed applications.

Of course, it is clear that for programming and running distributed applications, several tools (high level languages, software development environment, system services, etc...) have to be developed in order to offer a convenient environment to programmers of distributed systems.

## 4/ Overview of concepts in CHORUS

This section presents a very brief overview of the main concepts of CHORUS. It is simply intended to allow easier understanding of sections 5 through 7.

A distributed application is performed by a set of co-operating actors. Actors are local (not distributed) processing entities equivalent to the usual concept of sequential processes. Each actor can manipulate local objects (i.e. on the site where it resides) provided that it has a link to the object. Actors communicate by exchanging messages through ports.

The execution of an actor is supported and controlled by the kernel of its site. Controls apply in particular to linkage of actors to objects and to exchange of messages.

A distributed system is a set of interconnected computers (sites); on each site an operating system kernel supports system actors and application actors.

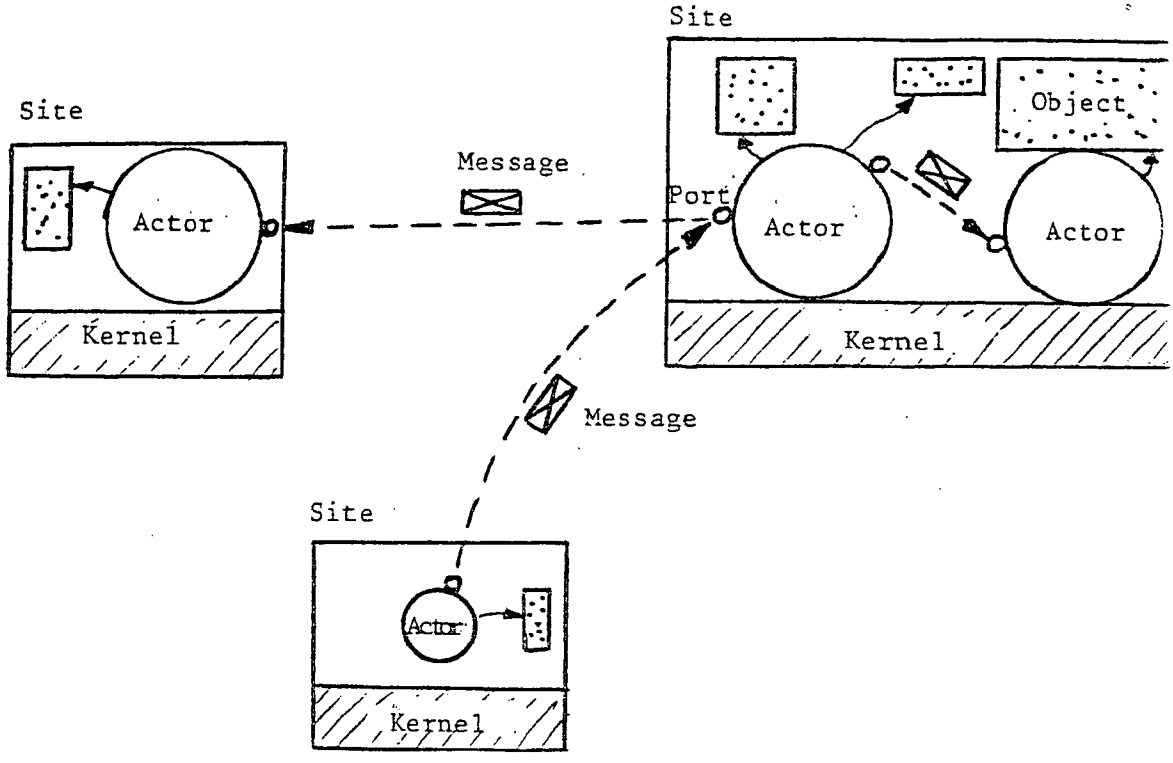



figure 4.1 : Illustration of basic concepts in CHORUS

Reception of a message triggers a processing-step within the receiving actor. Actors perform only one processing-step at a time. In other words, actors are sequential processors of messages.

Actors can be grouped into activities which are the units of distributed processing.

Actors co-operating by exchanging messages according to a protocol are said to be in a colloquy. Colloquies can take place between actors within the same activity as well as between actors in different activities.

In order to provide maximum flexibility, the processing configuration (i.e. mapping distributed applications onto the available set

of ressources) is gradually defined in three steps:

Distributed applications are initially described as a set of modules. Modules are then assembled into models of actor. Finally actors are created from the models, installed on specific sites and linked to local resources.

## 5/ Runtime architecture

### 5.1/ Basic concepts

CHORUS makes a clear distinction between (1) local processing to which centralized and tight control can be applied and (2) distributed processing which involves communications between physically distinct sites thus requiring completely different control schemes: local processing is performed by actors while distributed processing results from the co-operation between communicating actors.

An actor may be viewed as a local sequential process; its execution is started by a message and it can send messages on its turn. Note that the concept of actors in CHORUS is not identical with the concept of actors in [HEW 77].

Actors can perform actions only on local objects. Therefore, actions on remote objects require communications with remote actors residing on the same sites as the objects.

On each site the local kernel of CHORUS supports and controls operation of actors. For the sake of simplicity and reliability the kernel is kept minimum and most operating system oriented functions will be performed by actors. Facilities provided by the kernel are described as primitives which can be called upon by actors in

performance of their task. In the following text, the primitives ar pointed out by "=>".

The conditions under which actors exchange messages and manipu late objects are described in the following subsections.

## 5.2/ Communication between actors

Actors communicate by exchanging messages through ports. message is sent from one port of the sending actor towards one (o several, i.e. broadcast) destination port(s) of the receivin actor(s). Ports are bidirectional i.e. the same port can be used fo sending and for receiving messages as well. This choice results fro the consideration that most communications are bidirectional (there i usually at least a reply coming back after a request) and that actor normally send and receive messages. In addition this facilitate loopback with ports sending messages to themselves which proved to be extremely valuable for tests and maintenance.

Sending of a message is performed as a primitive offered by the kernel:

=> SEND (Source port, destination port(s), message);

The indication of a source port in the SEND primitive has two justifications: (1) it allows to perform appropriate controls on the sent message; (2) it may be used by the receiving actor to identify the source of the message, e.g. for a reply.

Actors have a uniform view of communications whether local or remote. They need not know where the destination port(s) is located.

From the point of view of the kernel, while communication with a local port can be handled directly, communication with a remote port involves protocols with the destination site (and possibly intermediate sites). For handling these protocols, the kernel will call service actors (see section 8.2) but this is hidden to application actors. This uniform view of all communications between actors is essential for easy reconfigurations of distributed activities.

The basic communication facility provided by the kernel is a datagram/lettergram type of facility: execution of a SEND primitive will cause copies of the message to be queued at the receiving port(s) after a variable delay with some probability of success. Additional facilities may be needed within the context of an association between ports (e.g. higher probability of success, flow control, sequential delivery, delivery confirmation, non-delivery diagnosis, etc...); these facilities will be built upon the basic SEND primitive. Again, from actors point of view, these facilities are the same whether with local or remote ports.

In the absence of flow control (basic communication facility), overflow of the receiver(s) capacity will cause additional messages to be discarded. If flow control is requested, overflow of the receiver(s) capacity will cause the sender to be blocked until its SEND request can be satisfied.

The SEND primitive provides a low level of service; a general purpose time-out mechanism will be used in order to cover such cases as loss of a message expected by an actor or failure of a correspondent from which a response is expected:

```
=> TIME_OUT (Port, time);
```

This primitive means: if no message is received on "Port" during "time" (starting now), the kernel sends an error message on Port conversely, if one message is received on "Port" during "time", the time-out is disabled. This primitive provides actors with a simple mechanism which allows them to "wait" for some event but not to be blocked if the event never occurs.

A send control procedure associated with each port permits the kernel to check the validity of messages being sent by the actor. Conversely, a receive control procedure associated with each port permits the kernel to check the validity of messages being received by actors. This decomposition of controls on communications into two parts (sending side and receiving side) matches physical distribution and provides each actor with its own autonomous protection.

However send and receive controls have not exactly the same goal: send controls are intended to control the external behaviour of an actor, i.e. to control that the actor sends only those messages it is expected to send, while receive controls are intended to protect an actor from the outside world, i.e. to control that the actor does not receive unexpected messages. These controls may, for instance, be used to check message types, passwords and so on.

Ports function only when associated with actors, i.e. a message destined to a port not currently associated with an actor will be lost. One port can be associated with only one actor at a time, but an actor can have several ports.

Establishment and release of an association between a port and an actor is performed as primitives offered by the kernel (an actor can open and close only its own ports):

=> OPEN_PORT (Port name);

=> CLOSE_PORT (Port name);

The main reason to introduce the concepts of ports is to decouple organization of communications from organization of processing. A correspondent is primarily interested in the functions being performed behind ports and not in the way these functions are performed. Usage of ports permits for instance to reorganize actors which provide a function through a set of ports without having actors which use the function be aware of this reorganization. This question of decoupling and restriction of visibility will be discussed further in section 8.4.

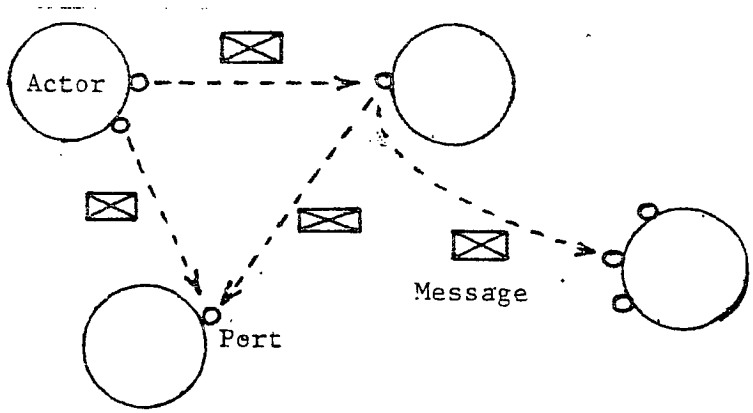



figure 5.1 : Actors exchange messages through ports

## 5.3/ Internal functionning of actors

CHORUS attempts to provide consistent schemes for organizing communications and processing. Messages which are the basic communication units are also the basic processing units.

Actors send messages, receive messages and process messages as well. For the sake of simplicity, each actor processes only one

message at a time, without any possibility of being interrupted at it level; the processing of a message is called a processing-step; processing-step is designated by its entry-point in the actor's code a processing-step must not include any kind of WAIT operation Parallelism, if required, can be obtained by using several actors Actors are simply sequential message processors.

A selection procedure associated with each actor permits the kernel to determine which message, if any, is to be processed next. Parameters of the selection procedure can be modified by the actor in order to influence dynamically the selection policy, using the primitive SELECT. The selection could for instance be restricted to a unique message which the actor is waiting for, or to messages coming from a given port, and so on...

=> SELECT (Port, condition);

where "condition" may be:

- all; means that every message received on Port may be selected.
- a list of ports; means that only messages sent from a port of the list onto Port may be selected.
- none; means that no message received on Port may be selected.

When a message has been selected, a switch procedure permits the kernel to determine which entry-point must be entered to process the message. The SWITCH primitive permits an actor to modify dynamically the selection procedure:

=> SWITCH (Port -> entry-point);

means that messages received on Port have to be processed by the processing-step designated by entry-point.

The end of a processing-step is indicated to the kernel by the primitive:

=> RETURN;

The kernel will then activate the actor for processing its next selected message as soon as available, and so on as long as the actor exists.

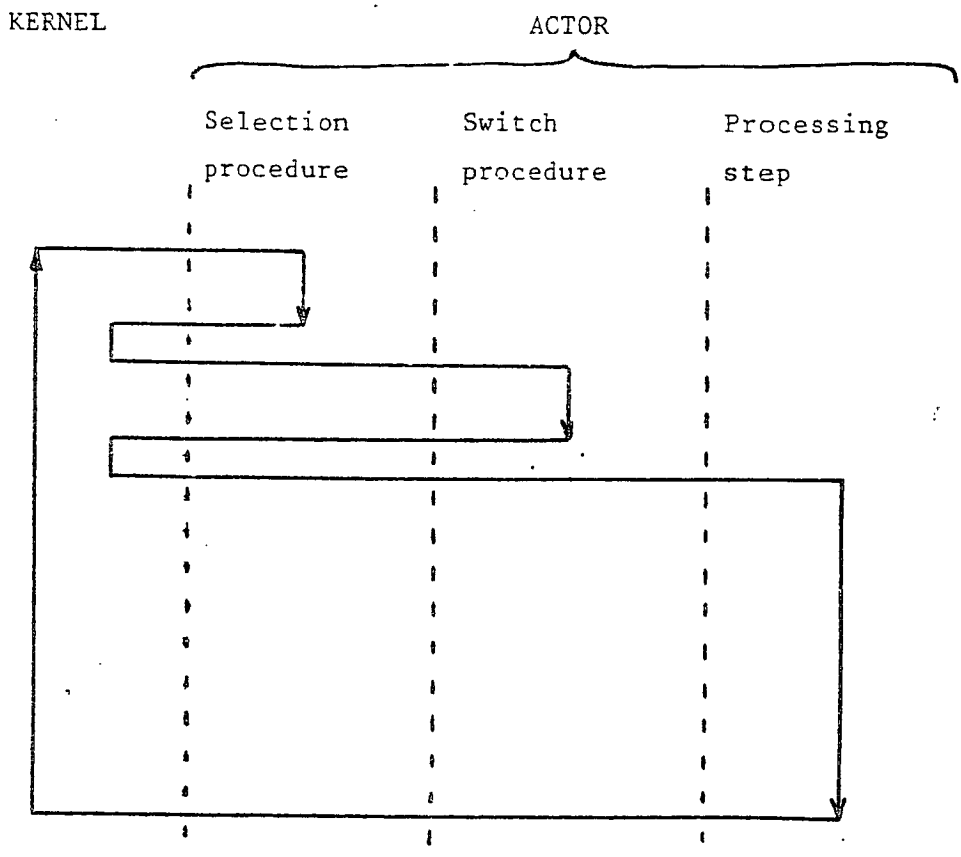



figure 5.2 : Internal functioning of actors

In order to strictly decouple communication from processing, messages sent by an actor are really emitted only at the end of the

current processing-step; in other words, when the actor executes SEND, the message is controlled with the send control procedure (se 5.2) associated with the sending port and then queued in the sendin queue of the actor until the actor executes RETURN; at that momen only, messages are transported to their destination port(s).

With this organization, execution of a distributed applicatio can be viewed as a graph of processing-steps connected by message (see figure 5.3). This should help, for instance, in organizin atomic execution of transactions (an execution which either complete or does not start at all) which forms a basis for error recovery [RE 79].

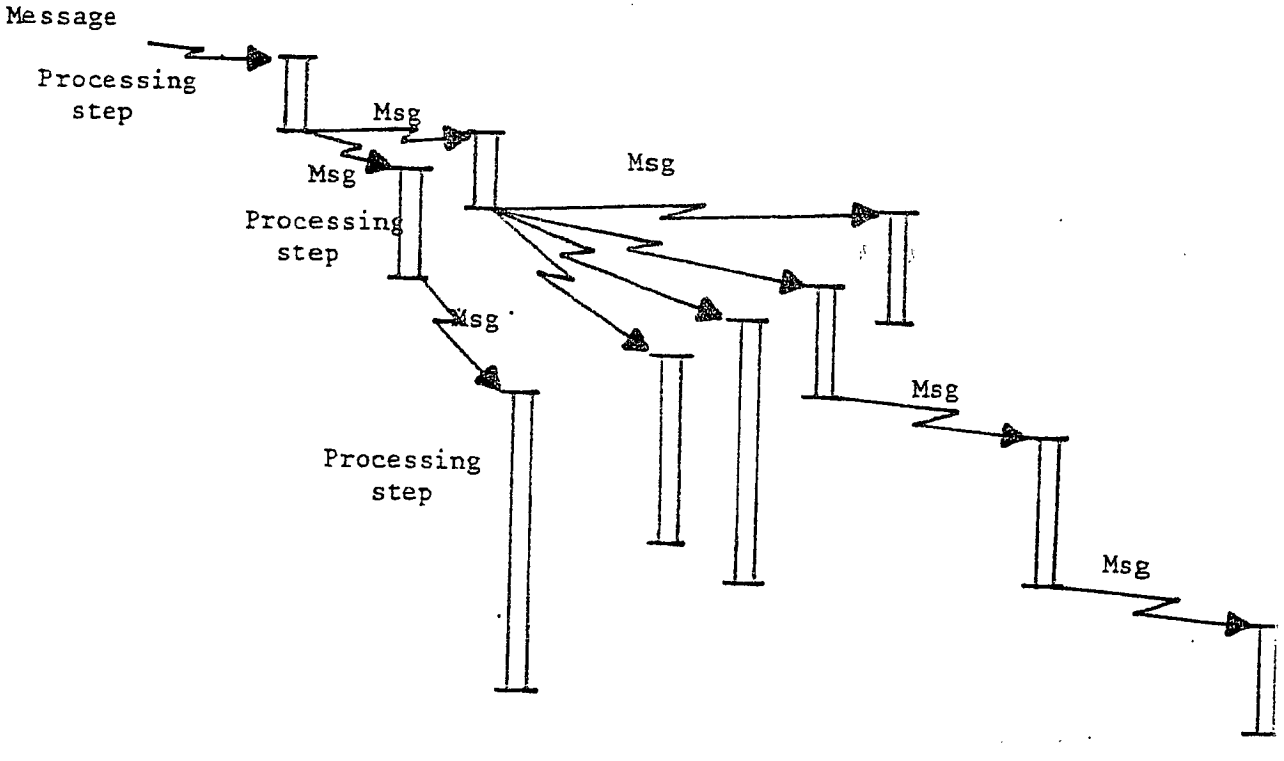



figure 5.3 : Processing-steps connected by messages

This organization of distributed processing as local processing-steps triggered by messages has proved to be very conve-

nient for programming distributed applications [AND 78]. However this transaction oriented organization of processing needs not in all cases be known by the final user, who can program actors in a more traditional way, as illustrated in section 8.3.

## 5.4/ Actions on objects

Objects exist independently of actors. However, objects are passive entities and nothing happens to objects unless acted upon by local actors.

In order to manipulate a local object, an actor must get a link to it. Establishment and release of links to local objects are performed as primitives offered by the kernel:

=> LINK (Object);

=> UNLINK (Object);

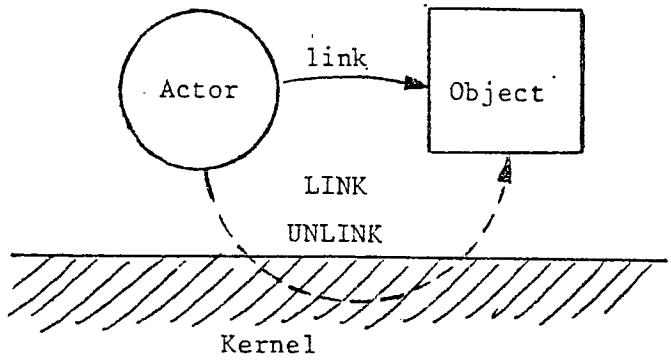
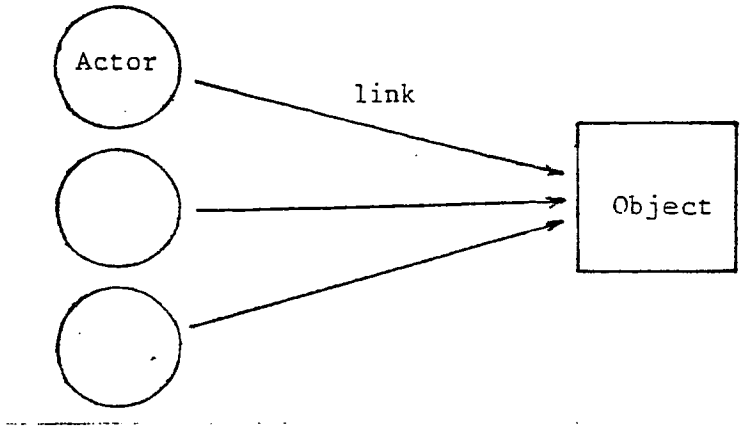



figure 5.4 : Link between an actor and an object

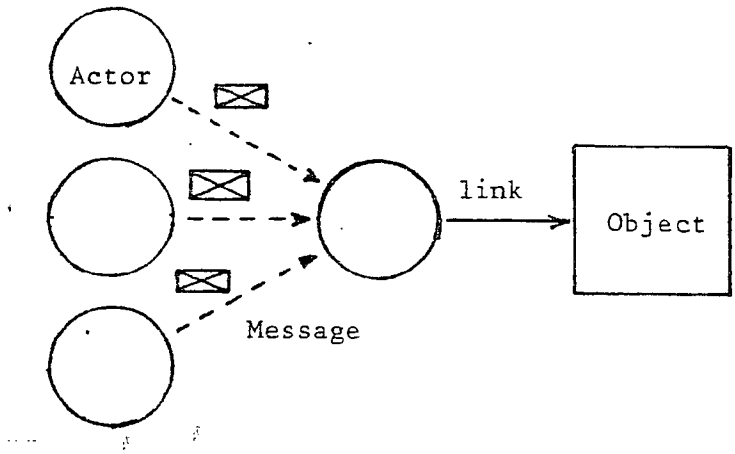
A linkage control procedure associated with the object permits the kernel to determine whether the actor is entitled to establish a link to the object (this is very similar to linking a segment to a

process with associated controls). Once the link has bee established, the kernel does not necessarily control what the acto does on the object (in contrast with capability-machines where eac access is further tightly controlled [FAB 74]); the controls depend o the ability of the machine. The actor as a whole is supposed t behave correctly, i.e. possible errors in the manipulation of the object by the actor are not controlled by the kernel; in other words, the actor is the elementary protection domain.

CHORUS permits objects to be directly shared (see figure 5.5.a) between actors, i.e. one object can be simultaneously linked to several actors (if this is authorized by the link control procedure). The main reason for making direct sharing possible is to reduce overheads in switching of actors by the kernel when the frequency of access to a shared object is high. The alternative scheme (see figure 5.5.b) is to have one single actor responsible for all accesses to the object and communicating with others by messages. One advantage of this indirect sharing is that all actors need not be on the same site.




a : Direct sharing




b : Indirect sharing

figure 5.5 : Sharing of objects between actors

## 5.5/ Activities

To date, two concepts have been introduced in CHORUS to help organizing runtime aspects of ditributed applications: activities which represent groups of actors pursuing collectively a common objective and colloquies which model co-operative relationships between communicating actors.

Just as actors are the units of local processing, activities ar the units of distributed processing. An activity is a group of actor which have been declared as belonging to the activity and collectivel form the activity. Each actor contributes to the objective of th activity. The initial choice, for the sake of simplicity, is that a actor belongs to only one activity.

A distributed job, consisting of actors on various machine performing the local processing required by the job and communicatin in performance of the job, would naturally be identified with one activity.

Actors co-operating in order to adapt routing tables in a packet switching network would form the route adaptation activity.

The objective of the concept of activity is to help organizing administrative aspects of distributed processing:

- Recovery will normally be organized on the basis of activities.
- Under some circumstances, failure of one actor might for instance cause abortion of the complete activity.
- The concept of activity can be used to build protection domains where actors of the same activity grant each other special rights which are denied to actors of other activities.
- Accounting can be done on a per activity basis.

## 5.6/ Colloquies

In a distributed activity, each actor (or group of actors) performs a given function in co-ordinating with other actors (or groups of actors) which perform other functions. Co-ordination is

ensured by exchange of messages between ports.

As discussed in section 5.2 usage of ports permits to decouple the structure of the dialogue between ports, i.e. communication, from the structure of processing by actors behind ports. Through ports, actors or groups of actors have only a restricted visibility of each other, limited to what is visible in their dialogue.

The dialogue, i.e. the ordery exchange of messages which takes place between a given set of ports is referred to as their colloquy. Each port engaged in the colloquy keeps a dialogue with the other ports participating in the colloquy.

The rules governing the dialogue between ports is referred to as the protocol of the colloquy. The protocol is the definition of behaviour of actors or modules behind the ports of the colloquy. The question of how a protocol can be specified is still a matter for research.

In some ways, one can say that a colloquy is an occurence of execution of a protocol as a process is an occurence of execution of a procedure; or, process and colloquy are dynamic notions which correspond respectively to the static notions of procedure and protocol.

Possible usages of the colloquy may be to keep a trace of messages exchanged between actors during a colloquy, or to control dynamically that the actors conform with the specifications of a protocol, and so on.

## 6/ Construction of distributed applications

### 6.1/ From design to execution

CHORUS tends to provide maximum flexibility in installation o distributed applications by identifying three phases where the physical configuration is gradually defined:

(1) A distributed application is initially described as a set o communicating modules;

(2) the configuration of the application is then determined b grouping modules into models of actors;

(3) finally actors (conforming with their models) are installed o specific sites to run the distributed application.

There is some analogy between this installation of a distributed application in three steps and the development of a micro-machine:

(1) A chip realizes an indivisible function; it is defined with its internal resources (processor, memory, etc...) and its interface (pins, signal level, functions);

(2) a circuit-board is then obtained by gathering chips and installing their interconnections, possibly in addition to a power supply, and so on; a circuit-board, in turn, has its own interface;

(3) finally, the micro-machine is obtained by installing various boards on racks and starting the whole thing.

## 6.2/ Modules

Modules correspond to the smallest possible decomposition of an application from the vantage point of distribution. Message processing defined by a module will be performed by one single actor, i.e. on one site and without parallelism.

The description of an application in terms of communicating modules can be viewed as describing the highest possible degree of distribution chosen for the application. Of course, it will be possible to group several modules for execution within one single actor but it will not be possible to distribute one module among several actors.

Therefore, a module contains all information required to build a model of actor, i.e.:

- Code and data describing processing of messages, including declarations of:
  * objects to be linked,
  * ports to be opened with their associated send and receive control procedures,
  * ports of other modules with which communications will take place,
- Identification of associated entry-points,
- A start-up procedure (see section 6.4),
- A selection procedure,
- A switch procedure,

## 6.3/ Model of actor

A model of actor is built by assembling one or several modules It contains the same kind of information (see section 6.2). Al elements except the selection procedure are obtained by gathering the corresponding elements of the modules. Assembly of selection procedures of modules require an additionnal choice since the selection procedure of the model of actor must cover the case where several modules compete for processing each a message.

In addition with these elements, a model of actor contains:
- an actor-creation control procedure,
- an actor-destruction control procedure.

These procedures are used by the kernel to check the validity of the creation and destruction of an actor issued of that model.

Crouping several modules into one single model of actor may in particular reduce overheads if modules have a high degree of interaction: communication between modules of the same actor may, for instance, be performed without send and receive control procedures. In any case, grouping modules into actors reduces the number of actors to be managed by the kernel. An additional refinement (which however needs to be investigated further) consists in transforming exchange of messages between modules in the same model of actor into invocation of procedures.

In order to allow easy reconfiguration by changing the distribution of modules in models of actor, it is essential that each module keeps its own ports; thus the interface between modules remains unchanged whatever be the new configuration.

## 6.4/ Creation and termination of actors

An actor can cause creation of other actors by means of the primitive:

=> CREATE_ACTOR (Site, model, start-up message, name of the actor);

Execution of the primitive causes the corresponding actor to be created and to process its start-up procedure. This may permit initialization of the actor (e.g. establishment of links to objects and opening of ports) and of its dialogue with other actors by sending messages. Once the start-up procedure has been executed, the actor processes the messages it receives, as described in section 5.3.

An actor can disappear by executing the primitive:

=> END_ACTOR;

Termination of an actor can also be caused by another actor by means of the primitive:

=> KILL_ACTOR (Actor);

Creation and destruction of actors are protected by the actor-creation and actor-destruction control procedures (see 6.3).

## 6.5/ Service

In most cases, distributed applications will not be described from scratch. Common functions will be available for building the distributed application and there comes the concept of service.

A service is described as a set of modules performing a well

defined function for the benefit of other modules. Communication between modules using the service and modules providing the service is described as exchange of messages through ports, as any communication between modules. In other words, the modules performing the service need not be defined again when describing the application.

When the application is configured, i.e. modules grouped into models of actor, it may be decided (e.g. for efficiency reasons) to group some service modules with user modules in models of actor. It is of course possible to have the service boundary coincide with actors boundaries.

When the application is installed, it may happen that actors which contain (only) service modules have already been created (e.g. as a permanent function in the distributed system). In this case, only the actors specific to the application need to be created.

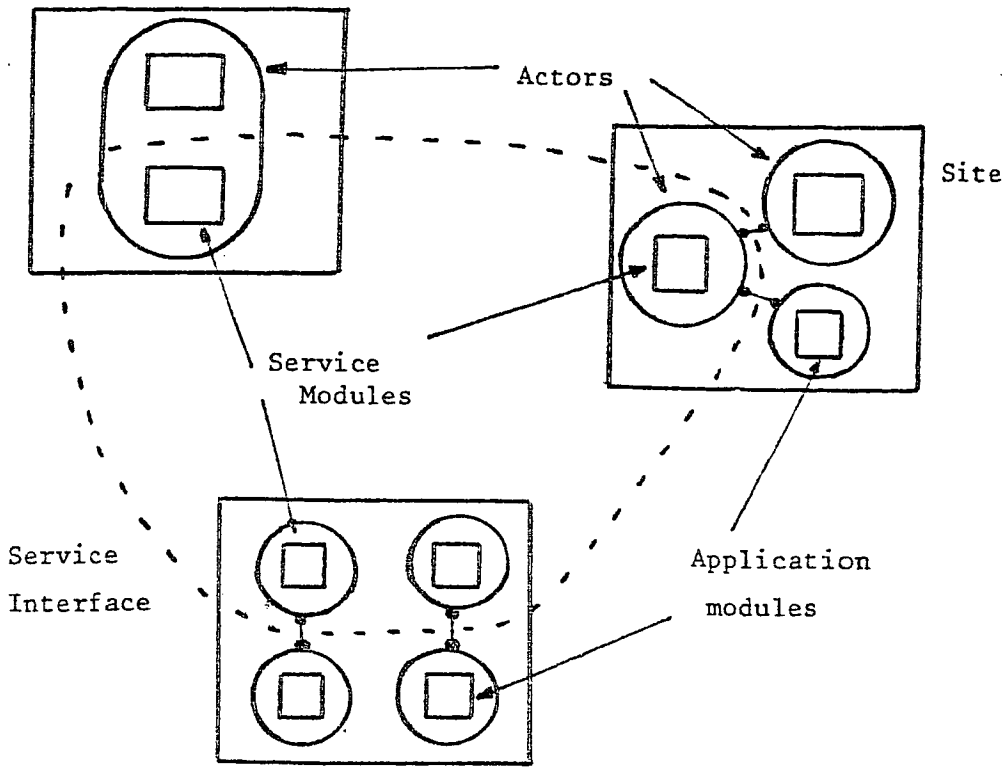



figure 6.1 : Service

It should be noted that the distributed application does not need to know the structure of all modules of the service. It only needs to know those modules which will be integrated in application specific actors and how to interact with the rest of the service (i.e. ports and protocol). If all modules of the service are outside application specific actors, the application need not know anything about the internal structure of the service. This restricted visibility permits to change internal implementation of the service without changing applications (provided the outside visibility of the service remains unchanged).

## 6.6/ Static reconfiguration

The distinction of three steps in the building of a distribute application provides convenient tools for "static" reconfigurations The grouping of modules into models of actor may be changed withou any modification in the modules themselves (as the inter-modul communication is through exchange of messages); the distribution o actors on sites may be changed without any modification in the actors (as the SEND primitive is localization-independent).

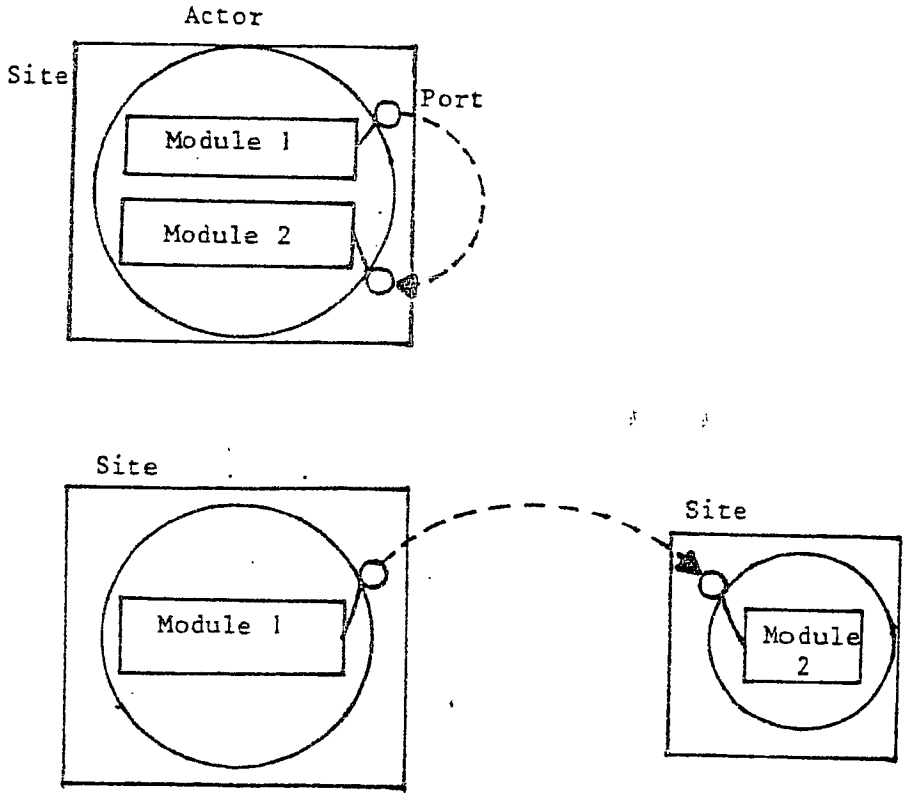



figure 6.2 : Offloading a function by reconfiguring modules

## 6.7/ Description, creation and destruction of objects

Objects are described by models of objects which contain in particular:

- the description of the structure of the object,
- its object-creation control procedure,
- its link control procedure,
- its object-destruction control procedure.

Objects are created by actors issuing the primitive:

=> CREATE_OBJECT (Model of object, parameters, name of the object);

where parameters may be:

- maximum dimension of an array, or a stack,
- password for access,
- indications on the mapping of the object onto physical resources.

The object-creation control procedure is used by the kernel to determine whether the actor is entitled to create such an object.

The link control procedure is used by the kernel to control the establishment of a link to the object (see 5.4).

Objects are destroyed by actors issuing the primitive:

=> DESTROY_OBJECT (Object);

The object-destruction control procedure associated with the object is used by the kernel to determine whether the actor is entitled to destroy the object.

## 7/ Names

In CHORUS, all entities which need to be designated (objects ports, actors, modules, etc...) are identified by names.

Basically, all entities are designated with global names whic have identical meaning wherever they are used in the distribute system. In order to guarantee this identity of meaning in spac despite of synchronization problems, global names in CHORUS are als permanent names, i.e. have an identical meaning through time. Once global name has been created, i.e. has been given a meaning, it keep its meaning forever. In other words, global names are not reused.

All entities can be accessed by means of global names, i.e CHORUS should guarantee proper designation of entities despite o synchronization problems and failures.

In CHORUS, global names are large (e.g. 64 bits): it i therefore easy to have names which are never used twice; this impose to be able to manage large tables of names, e.g. with hash-coding.

In some cases, management of global names and/or access by globa names may induce too much overhead and thus, CHORUS also permits usag of local names. In contrast with global names, the meaning of loca names, i.e. entities they identify, depends on the context (i.e. where and when) in which they are used. Symbolic names (i.e. name used by programmers) are a specific case of local names.

The association of global names with entities should be guaranteed by the kernel. There is a lower level of guarantee about the association of local names with entities since inconsistencies may be introduced by synchronization delays and failures. The various

sets of local names will have their own management mechanisms which may be less costly than for global names, but also less reliable. In case of failure or of ambiguity in the usage of theses names, the global names provide a reliable basis for recovery.

Each entity can be designated explicitly with a specific name which designates this and only this entity. Generic names are used to designate entities with a given functional property (e.g. a port providing a given service). Several entities may have the same generic name.

The concepts of global/local names and of specific/generic are orthogonal, i.e. global names as well as local names may be specific or generic as well.

One entity may have several specific names and several generic names, but each entity must have at least one global specific name, to ensure stability and uniqueness of designation required by CHORUS to manage entities.

# C/ ILLUSTRATION AND DISCUSSION

## 8/ Illustration and discussion of concepts in CHORUS

These sections illustrate and discuss in more details usage of basic concepts presented in sections 5 through 7. Indeed, CHORUS tends to keep its kernel minimum and to let it provide only elementary mechanisms on top of which a variety of functions, services and utilities can be built. The following examples intend to give an indication of how the kernel of CHORUS can be used and expanded.

### 8.1/ Usual controls of execution / Synchronization

CHORUS is intended to support a variety of applications which have not all the same requirements in terms of synchronization. For that reason we adopted, as possible, the simplest options.

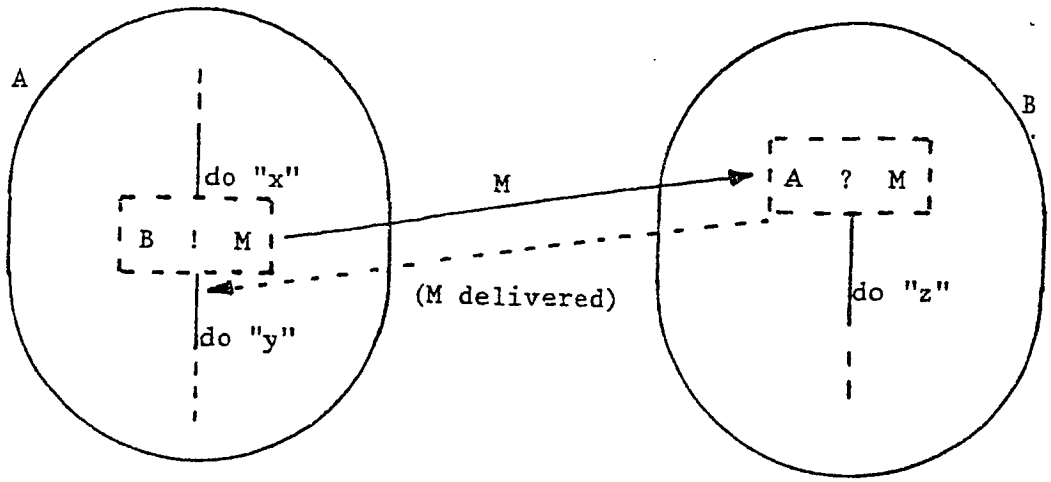
In order to permit implementation of a variety of synchronization schemes, the kernel provides an elementary tool in the form of a SEND primitive (section 5.2) which offers a low level of communication facility; this is a "no_wait send" in the sense of [LIS 79]. Upon this primitive, a user can build more sophisticated communication mechanisms, such as a "synchronization send" in the sense of [HOA 78] - i.e. a primitive which blocks the source actor until the destination actor has begun to process the message - or such as a "remote invocation send" in the sense of [BRI 78] - i.e. a primitive which blocks the sending actor until reception of the response from its correspondent.

The examples below illustrates these possibilities. They present

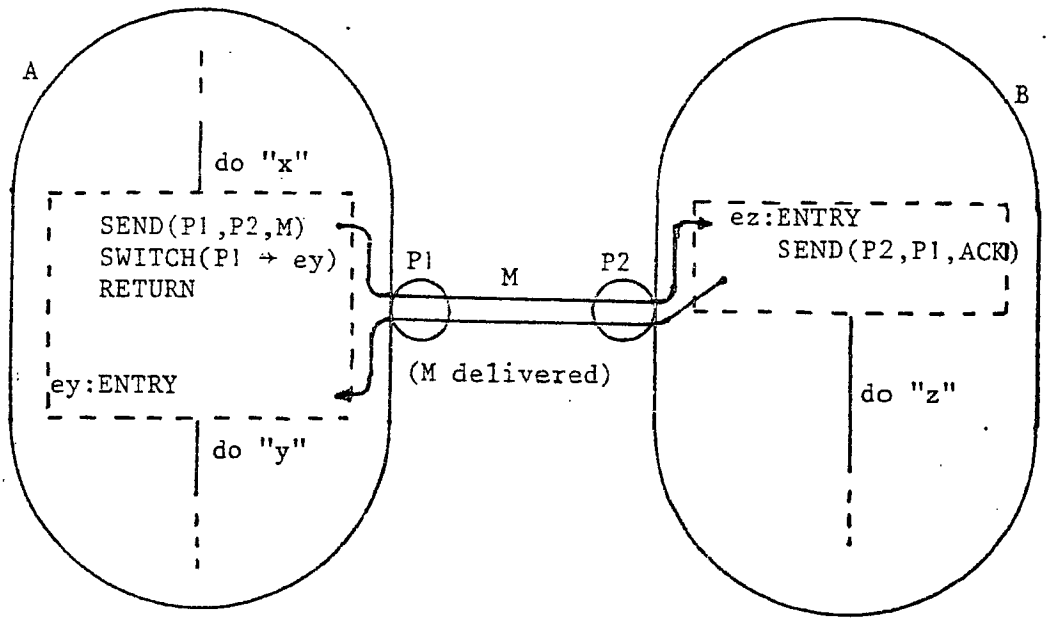
only the case where the SEND primitive works safely; synchronization recovery in case of failure, error, congestion, etc... is a general problem, beyond the scope of this paper.

Example 1: Synchronization SEND

Let be two actors A and B. A sends a message M to B using the "synchronization SEND" and B is waiting for this message.




a : HOARE's notation

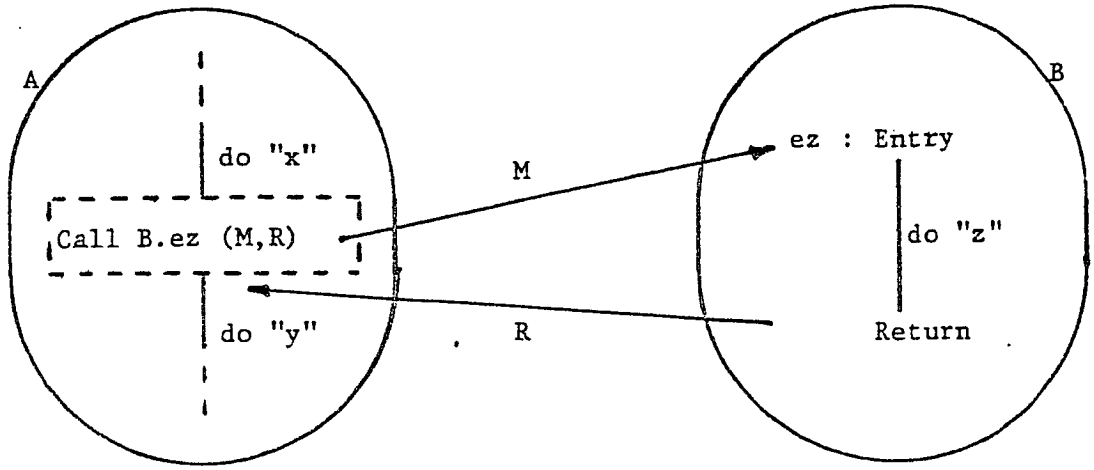



b : CHORUS notation

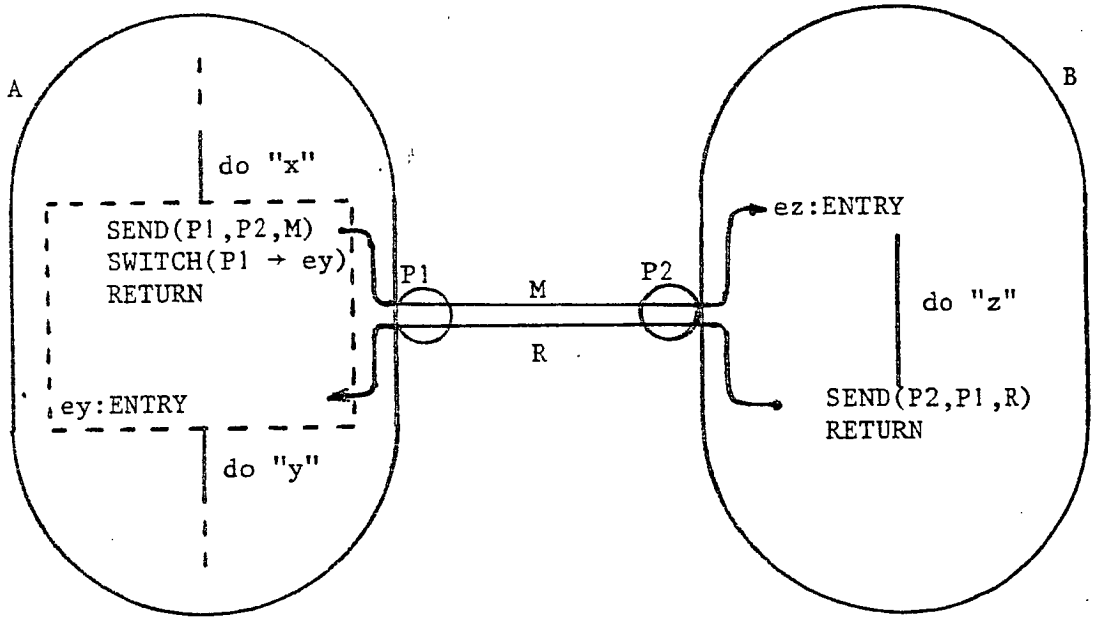
figure 8.1.1 : Synchronization SEND

Example 2: Invocation SEND.

Let be two actors A and B. A sends a message M to B using the "invocation SEND".




a : BRINCH-HANSEN's notation




b : CHORUS notation

figure 8.1.2 : Invocation SEND

Similarly CHORUS can support other usual synchronization scheme such as coroutines or asynchronous parallel processes with fork an join.

Example 3: Coroutines

Two actors A and B can work as coroutines if they are programme in the following way:

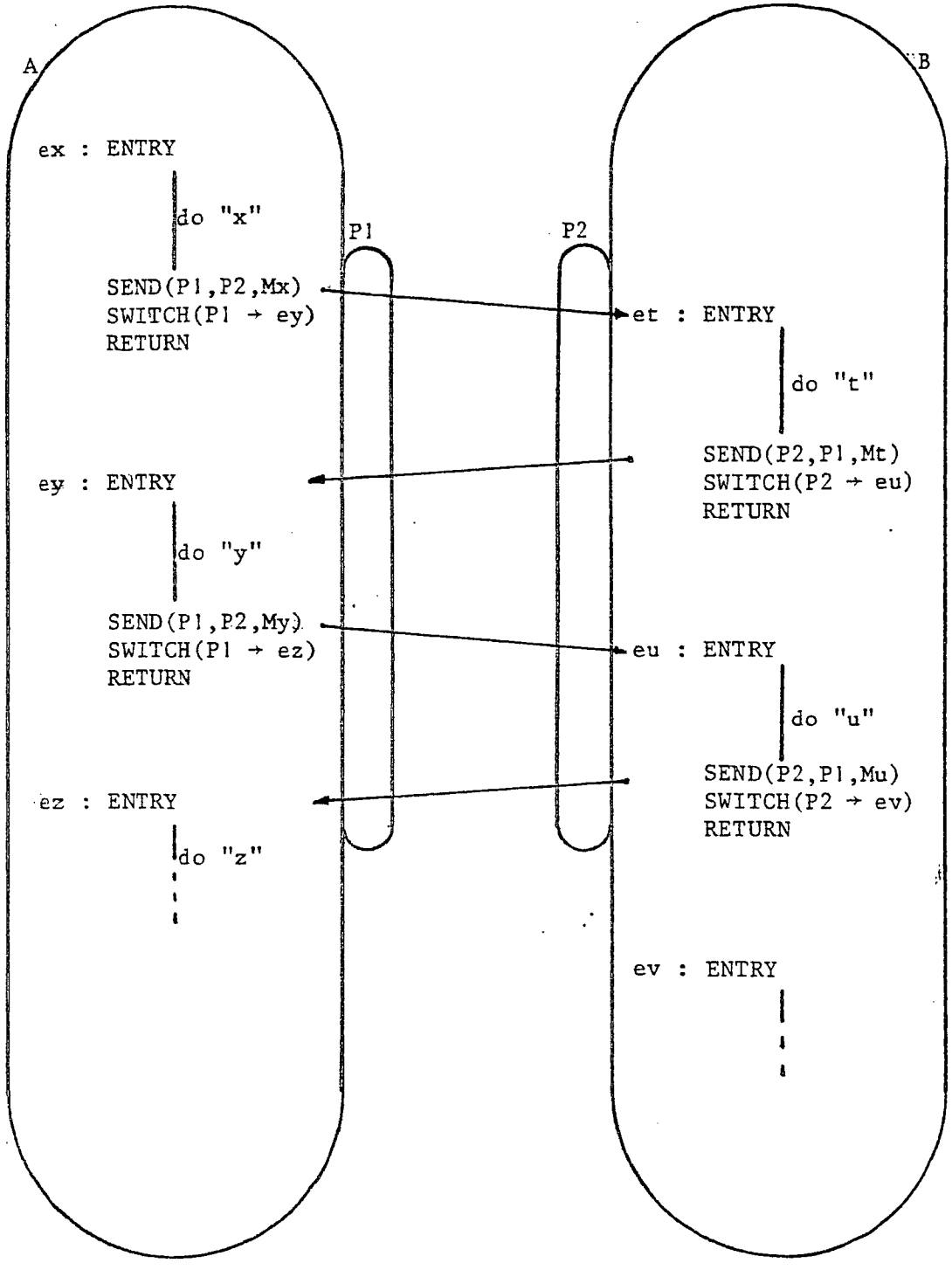



figure 8.1.3 : Coroutines

Example 4: Fork and Join

An actor A can create a parallel actor B and they both can join as follows:

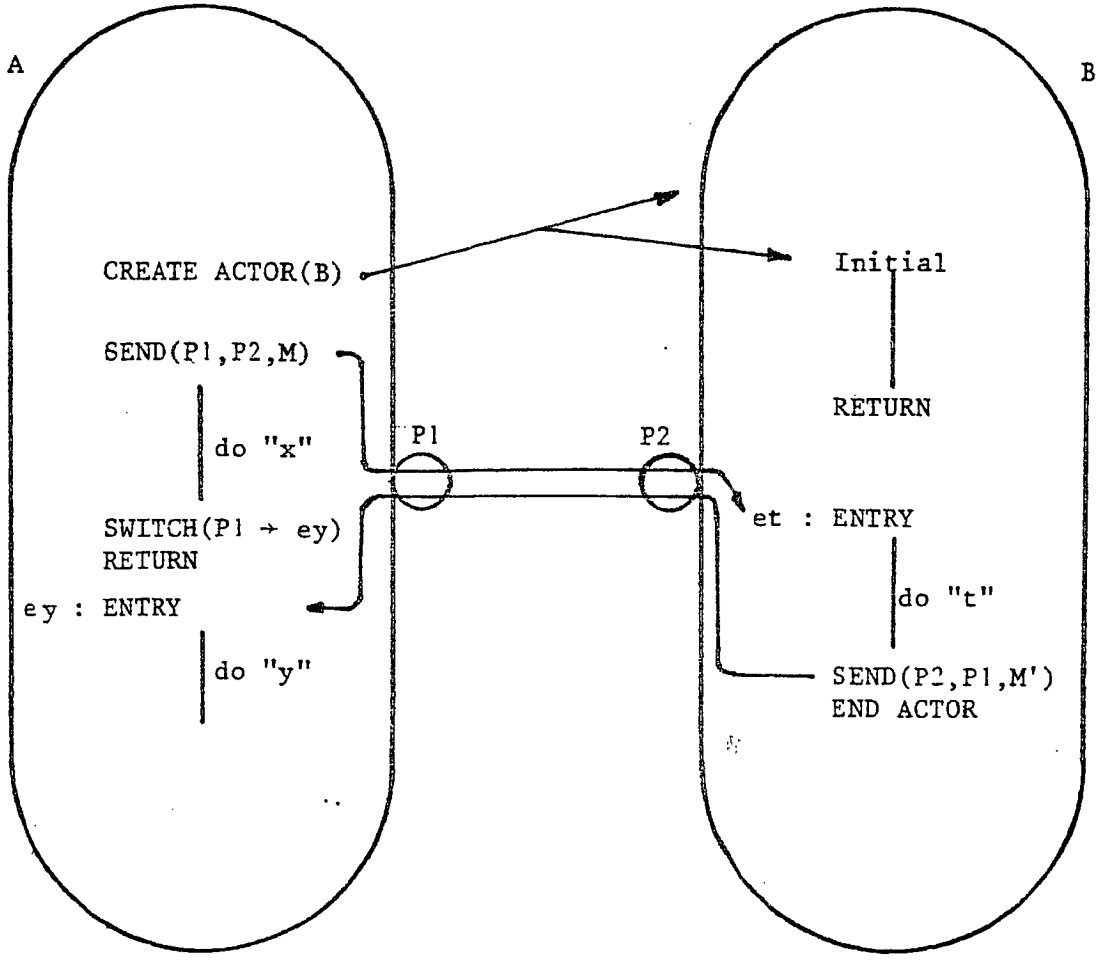



figure 8.1.4 : Fork and Join

The actor A creates the actor B; A sends M to B which processes M and runs in parallel with A. A stops at the end of its current processing until it receives M' from B. When B ends its processing, it sends M' to A. On reception of M', A processes it, i.e. executes the treatment designated by the entry-point ey. Whichever be the

actor which ends first, A will execute the treatment at ey only when both A and B have ended their current processing: A and B join at ey.

To sum up this section, one can see that the combination of the SEND primitive and the internal functionning of the actor makes it easy to realize various usual communication and synchronization mechanisms.

## 8.2/ Message transfer

As presented in section 5.2, CHORUS provides actors with a uniform view of communications, whether local or remote. Local communications can be handled directly by the kernel, while remote communications may involve complex protocols which are above the capacity of the kernel. It is also important that these protocols can be changed (e.g to accomodate new communications media) without impacting neither the kernel nor the applications which use the message transport function. Therefore, handling of communication protocols is subcontracted by the kernel to specific actors, according to the scheme outlined below.

When the kernel gets a SEND request from an actor, it first checks the validity of the request according to the send control procedure associated with the sending port. If the request is valid, the kernel determines whether the destination port(s) is local or remote. In the first case, the message is simply queued at the local destination port where the associated receive control procedure will be executed. If the destination port is remote, the message is passed to a special actor (henceforth called transport actor) on a local port

acting as surrogate of the remote destination port.

The transport actor interacts with other transport actors other sites through communication lines, according to communicati protocols, until the message reaches the transport actor in t destination site. This latter actor then issues to its kernel a SE request to the (local) destination port, from one of its ports actir as surrogate of the remote source port. The kernel recognizes th specificity of this SEND request (i.e. the fact that it comes from surrogate port) and passes the message to the destination port as i it came directly from the remote source port.

The structure of actors performing the transfer of message between ports is not known by the kernel. Instead of having simpl one transport actor on each site, there may for instance be fou actors each corresponding to the four lower layers of the Open System Interconnection Reference Model [ISO 79]. Transport actors collec tively perform the distributed transport service which is viewed b other actors (through the kernel) as identical with local communica tion facilities.

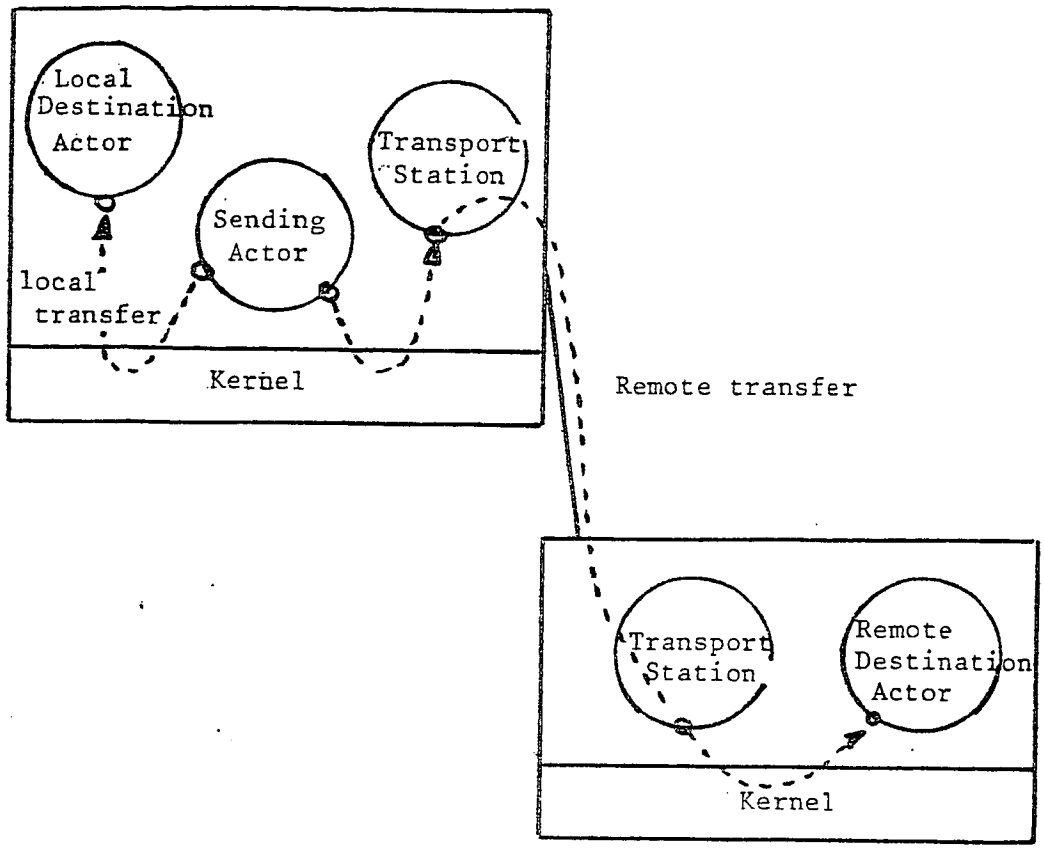



figure 8.2.1 : Local transfer and remote transfer

## 8.3/ Transactionnal and/or sequential programming

According to the scheme presented in section 6, a distributed application is described as a set of uninterruptible treatments started by messages. Those treatments are then assembled into modules which are themselves grouped into models of actor.

The most elementary way of programming a distributed application is therefore to describe it as a set of uninterruptible treatments triggered by messages:

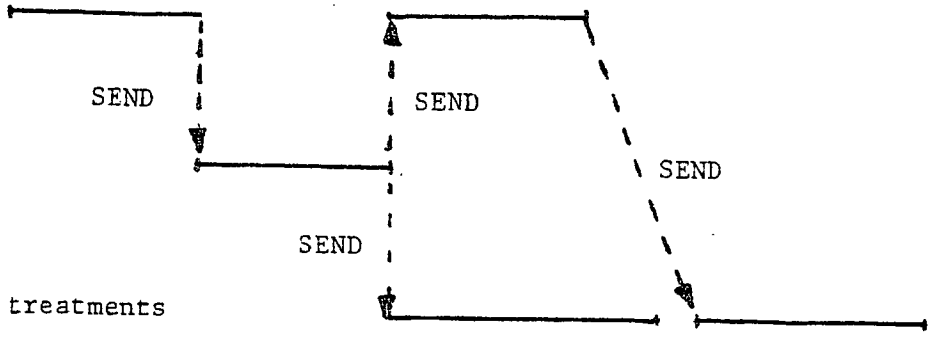



figure 8.3.1 : Basic structure of execution of a distributed application

Experience shows that this unusual way of programming (sometimes referred to as transaction-oriented) can be very well accepted by users [AND 78].

However, there are cases where sequential programming is more adequate or even unavoidable (e.g. existing programs). Indeed, sequential programs can also be run on top of the transaction oriented structure offered by CHORUS (this being of course hidden from the programmer who can write his programs in a traditionnal sequential way) as illustrated below with the example of a GET operation.

In CHORUS, I/Os will be performed by specific actors. The basic scheme for I/O operations is as follows. An actor which needs to perform a READ or a WRITE operation sends an I/O request message to the appropriate I/O actor and receives back later the response in an I/O response message from that I/O actor.

In sequential programming, the programmer is used to use a GET instruction which insures that the result of the READ operation is available when the next instruction is executed:

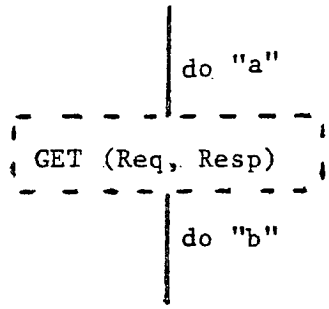



figure 8.3.2 : Sequential programming of a GET operation

Automatically (i.e. the programmer has not to do it himself), this instruction can be transformed in the following sequence:

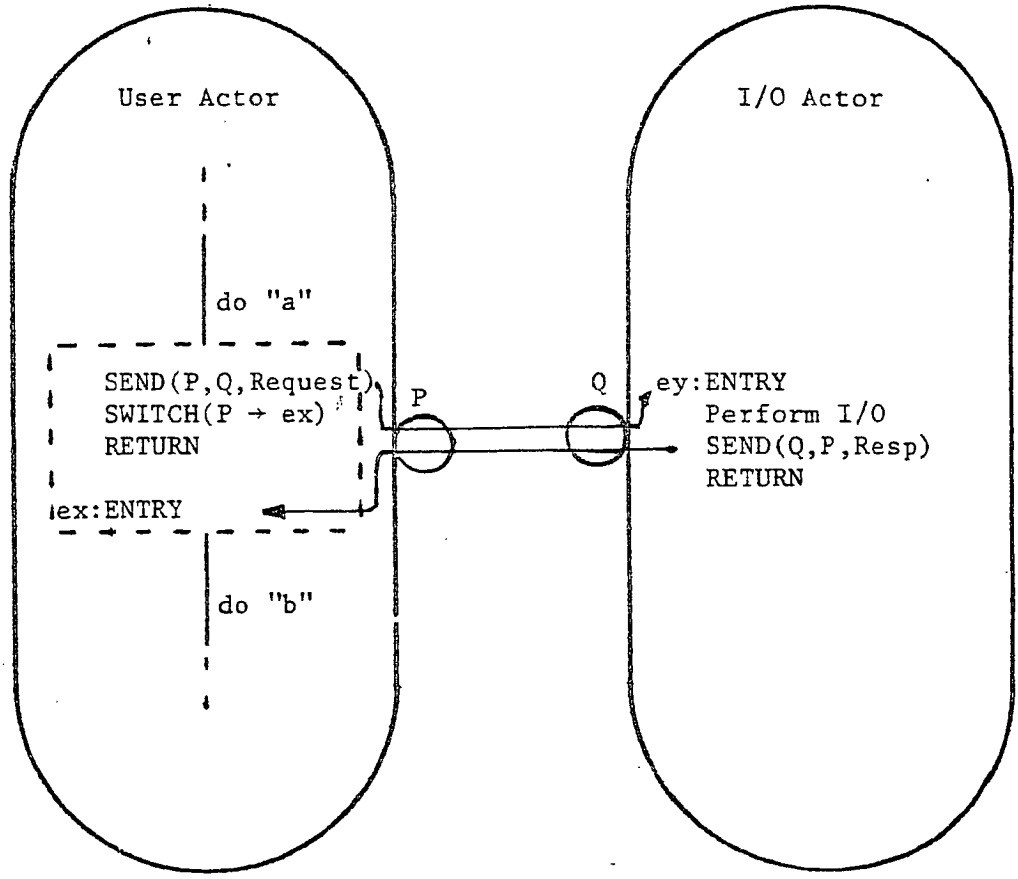



figure 8.3.3 : Transactionnal programming of a GET operation in CHORUS

At execution time, the user actor sends an I/O request message from P onto Q behind which an I/O actor performs the READ operation.

Then the user actor stops. When the I/O actor sends back a response onto P, the message activates the treatment beginning with the entry-point ex.

## 8.4/ Abstract data types

Programming with abstract data types is a well known way to structure programs and to ensure protection [LIS 74]. Indeed, this enforces typed data to be accessed only via procedures associated to their type. The corresponding protection controls can be performed statically, i.e. at compile-time, but in some protection-oriented systems [ENG 74] the controls are performed also dynamically, i.e. at runtime. This is the case with capability-based systems, in which an object type can be realized by a protection domain.

CHORUS permits to construct similar protection arrangements on the basis of actors, as outlined below.

By means of linkage control procedures it is possible to associate one model of actor to an object and to forbid linkage to this object by actors created from other models. So actors of a different model (we could say "of a different type") could access this object only indirectly through an actor of the right model. Usually, the type of an object is defined by the list of operations allowed on this object; similarly, the type of a model of actor could be defined by the list of the entry-points of its modules which realize the operations.

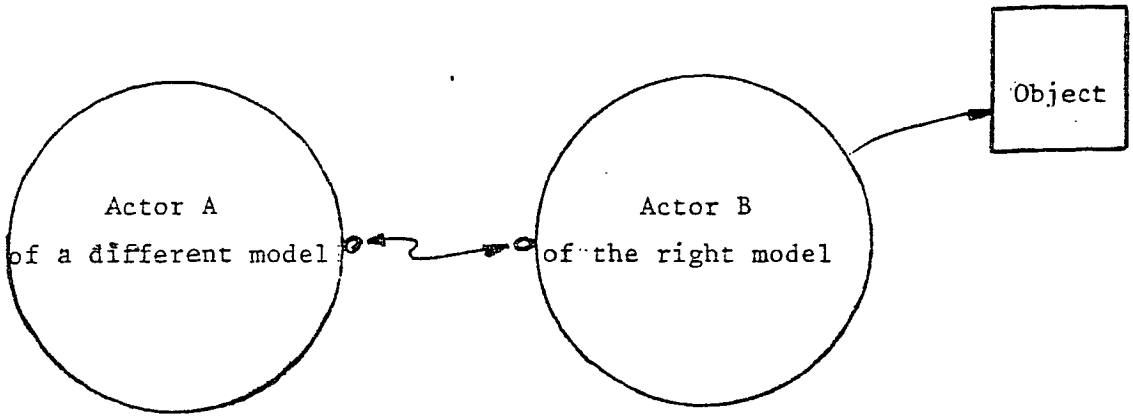



figure 8.4.1 : Access to an object through an actor of the right model

This is similar to the scheme of implementation of an object type in a protection domain [WUL 74].

As seen in 5.4, the typed object can be kept in a single server actor or can be shared by a number of actors of the right model. In the latter case, we have the following scheme:

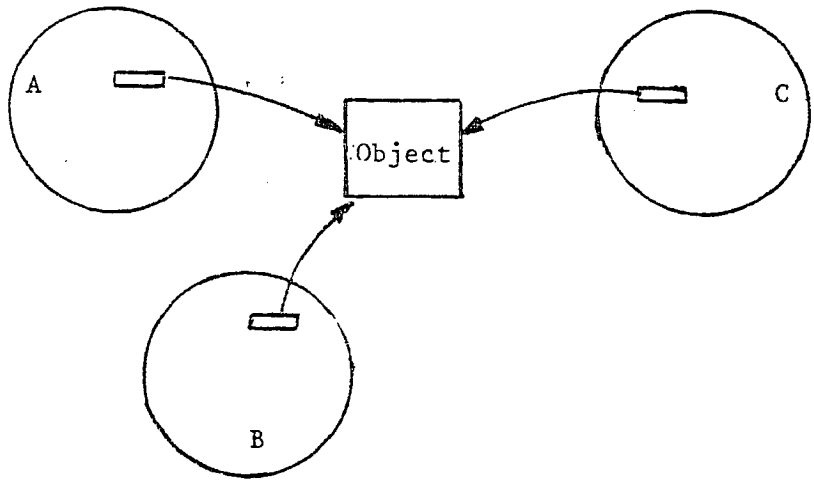



figure 8.4.2 : Several actors of the right model access a shared object

A, B and C are actors issued from the model of actor associated with the type of the shared object O. They are on the same site as O

and can act as servers for other actors anywhere in the system.

For efficiency reasons (minimize number of actors and number c messages exchanged between actors) it may be desirable that a give type of object be associated with several models of actor, provide that each model includes the right modules for the handling of th object. This would of course necessitate validation of several model instead of one, but on the other hand, this would allow various use actors to access directly a common object, without the need of usin server actors (which implies two exchanges of message).

## 8.5/ Distributed abstract data types

The scheme presented above where objects are accessed onl through server actors of the right model can be extended further t ensure protection of constructed objects, possibly distributed or several sites. For instance, consider an object maintained in several copies in a distributed system. Each copy can be handled by one server actor. All these server actors can be assembled into one activity which, as a whole, ensures consistency of the various copies.

From outside the activity which encapsulates it, the replicated object can only be accessed through the interface offered by the activity. For instance, suppose that two distinct operations op1 and op2 are possible on the constructed object; user actors (e.g. A on figure 8.5.1) will know two ports (of a server actor), each associated with one of these two operations.

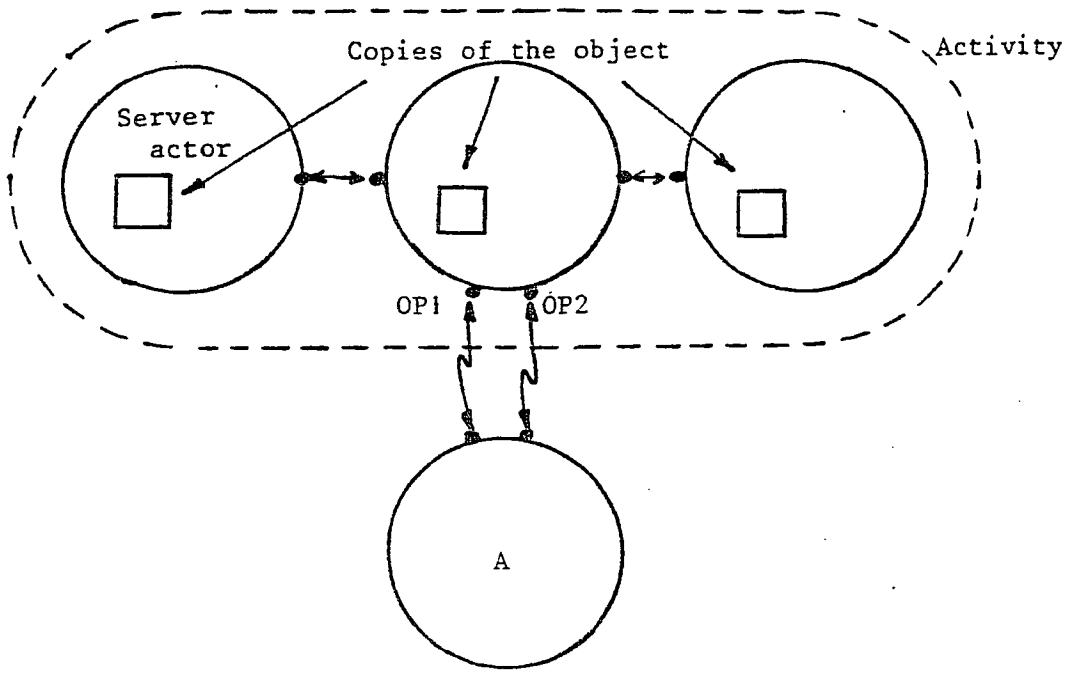



figure 8.5.1 : Management of distributed copies of an object

In the activity formed by the server actors, these operations op1 and op2 will be performed (by means of exchanges of messages between the servers, inside the activity) in such a way that consistency of the set of copies is maintained. In this case, one could consider that the type of the server activity matches the type of the replicated object. However, in the present definition of CHORUS, there is no concept matching directly an "activity type". This can be obtained only indirectly by assembling actors of adequate models.

There may be cases where the interface to the server activity is more complex (e.g. require interactions with several actors on different sites). In this case it may be useful to hyde this complexity with an "interface actor" which will accept messages associated with macro-operations and will translate those into a number of micro-operations within the network of actors:

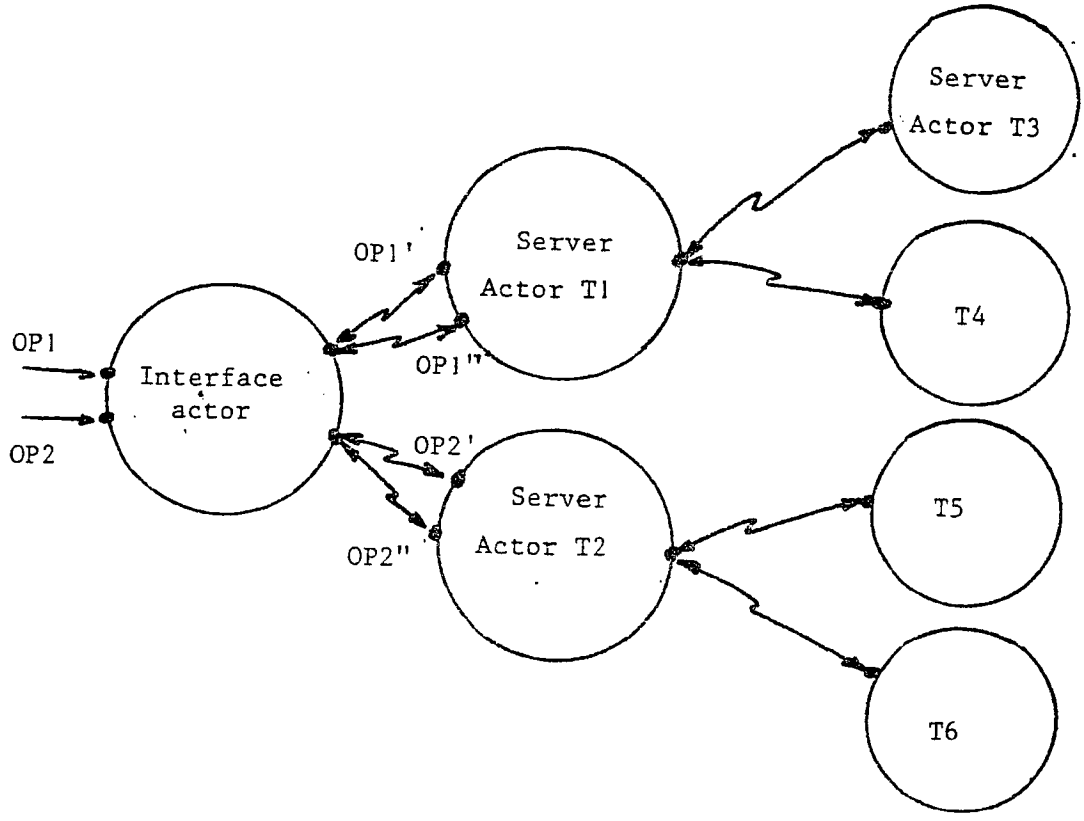



figure 8.5.2 : Interface actor

This scheme offers some similarities with constructed types [FER 76], i.e. an object built of several objects of other types. In our case, operations op1 and op2 on the constructed object are decomposed in more elementary operations op'1, op"1, op'2, op"2 on parts of the constructed object. This scheme can be extended further to more than two levels, as shown on figure 8.5.2.

## 8.6/ Network architecture and protocols

The design of communication aspects of CHORUS is very much influenced by previous (and current) works on computer networks around ARPANET, CYCLADES and more recently within the International Organiza-

tion for Standardization (ISO) for development of standards for "Open System Interconnection" [ISO 79]. A result of this work has been to emphasize the role of communications which of course is the vehicle for co-ordination between elements of distributed processing but also decouple these elements from each other, thus permitting heterogeneity. This led to developments of protocols and network architectures as an essential aspect of distributed systems. Much work remains to be done in the area of protocol and network architecture specification, implementation, debugging and maintenance. CHORUS contributes to this general effort by providing a set of elementary tools which facilitate handling of protocol aspects in distributed systems. This is illustrated below with the example of the network architecture for "Open System Interconnection" (OSI) which is currently being standardized within ISO.

As most network architectures, the OSI architecture is a layered architecture by which, from the point of view of communications, co-operating systems are decomposed into layers of co-operating entities, as illustrated in figure 8.6.1:

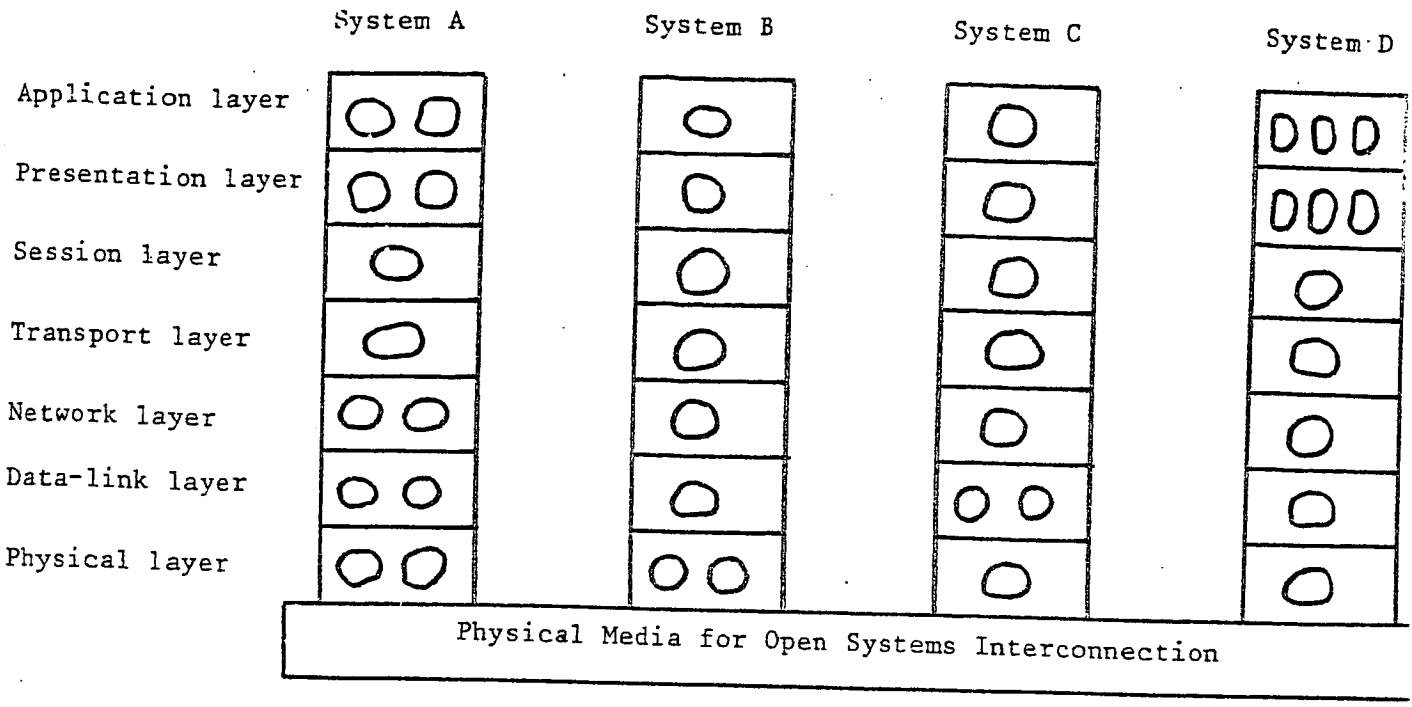



figure 8.6.1 : The OSI layered architecture

Entities in a layer communicate through services provided by the next lower layer. Co-operation between entities in a layer in performance of the function of the layer is ruled by the protocol of the layer.

CHORUS permits to support this type of architecture by simple mapping of ISO concepts as outlined below:

OSI entities can be mapped one to one onto CHORUS modules. Assuming that an OSI system is identified with a CHORUS site, some flexibility is left in the grouping of modules into actors, i.e. the implementation of entities in a system need not be identical, as illustrated in figure 8.6.2:

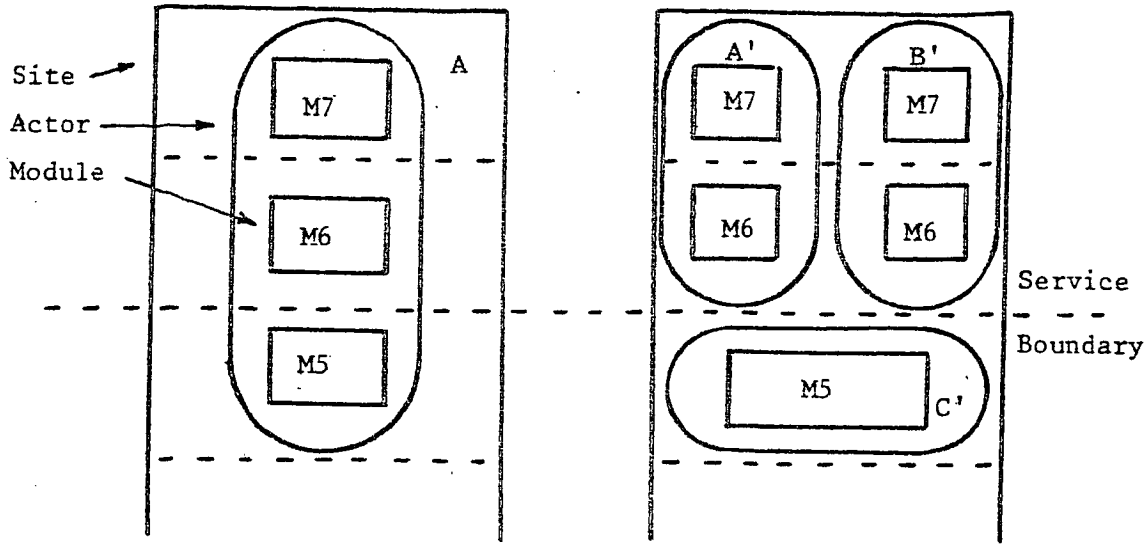



figure 8.6.2 : Different implementations of the same model of architecture

The OSI concept of service provided by each layer to its next higher layer through service-access-points can be mapped in CHORUS onto the concept of service provided by a set of modules to other modules through ports. This is possible without constraining unduly implementations since CHORUS does not impose that service boundaries coincide with boundaries between actors. CHORUS permits implementation of a service request as a procedure call if the service boundary is internal to an actor or as an exchange of messages if the service is provided by another actor, as illustrated in figure 8.6.2 above.

The OSI transport service could be identified with the SEND function offered by CHORUS for communications between ports in different sites. In fact, the SEND function in CHORUS is more general since it covers also communications between actors on the same site, i.e. in the OSI model, communications between entities in adjascent

layers within the same system. This is essential for organizin implementation (one objective of CHORUS) but not for organizin communications (sole objective of the OSI model).

CHORUS permits to specify OSI protocols by specifying module which run these protocols. This technique is sometimes refered to a "specification by example implementation". If modules are describe in a high level language (such as PASCAL, for instance) this technique provides some implementation independence. However, there is clearly a requirement for more powerful tools to specify protocols. CHORUS in its present definition offers the advantage of providing a practica solution for today's needs.

Test of implementation of protocols can be very much aided by CHORUS through usage of send and receive control procedures (see 5.2). These can be used to check that some of the rules edicted by the protocol are actually followed by actors. In addition, CHORUS permits to replace actors behind a set of ports by another set of actors able to perform tests, without having their correspondents be aware of this change.

# D/ CONCLUSIONS

## 9/ Topics for future investigation

Experience gained through implementation and experiments with a variety of distributed applications will permit to refine and validate the initial definition of CHORUS presented above.

However, several topics require further investigation in order to complement the basic set of concepts currently defined in CHORUS. This is in particular felt to be the case for: management of errors, management of names, management of time and language aspects.

### Management of errors

CHORUS provides means to detect errors by means of controls performed at runtime according to user defined control procedures. However, in its present definition, CHORUS does not provide any specific concept or tool to support and/or facilitate recovery. This could be developed in the area of execution structure for recovery procedures, identification of redundant entities and synchronization between them, replay facilities or backtracking, etc...

### Management of names

CHORUS has borrowed its initial naming scheme from CYCLADES [POU 76]. It is very general and, as most other naming schemes, easy to manage (allocation of names, maintenance of access machinery and accesses themselves) as long as entities do not move. Management of names in a more dynamic environment is much more complex and has to be investigated further in the context of CHORUS, possibly on the basis of techniques developed for packet switching.

Management of time

It has often been noted (e.g. [LEL 77]) that there is no concep of "global" or "universal" time in a distributed system. This i quite true, but it has also been shown (e.g. [DAL 78], [FLE 78] o [LAM 78]) that references to local time or to an approximation o global time may facilitate synchronization. This question of manage ment of time will also have to be investigated further in CHORUS ir order to determine which basic tools are adequate for aiding synchronization in distributed applications.

Language aspects

CHORUS is designed bottom up, from physical components to logical concepts, from runtime aspects to description of applications. This choice results from the consideration that low level aspects of distribution will have, at the end, a major impact on high level concepts for distributed applications.

Concepts gradually developed in CHORUS will have to be made available to programmers and to users, i.e. integrated into their languages. Initially, the concepts of CHORUS will be presented in the form of a control language, in order to have more flexibility for evolution. However, one of the objectives in CHORUS is to have these concepts (when stabilized) integrated into a programming language.

## 10/ Conclusion

This paper presents the initial definition of CHORUS, an architecture for distributed systems. It is only a first step in the project, much work is still to be done and several problems have to be investigated further. However, it is felt that some important goals of the project have already been reached:

Communication between actors through ports allows the definition of functional interfaces independently of the realization of the function in actors and of their localization.

Construction of distributed applications in three steps using modules, models of actor and actors offers much flexibility for reconfiguration;

The universal SEND primitive combined with the internal functionning of an actor gives a simple but uniform tool for synchronization independent of the localization of actors.

At last, user defined control procedures provide a powerful tool for development of protection schemes tailored to each class of application.

Initial usage of the concepts of CHORUS for designing distributed applications in the area of Distributed Data Bases, Office Automation and Process Control indicates that, even incomplete and not polished, CHORUS provides a "plus" for building distributed systems.

## 11/ Acknowledgements

The authors would like to thank all friends who contributed CHORUS through many discussions and fruitful comments. Without thei help, CHORUS would not be what it is. Particular gratitude i expressed to C. Kaiser who acts as scientific adviser to the project.

## 12/ Glossary of terms and primitives

### 12.1/ Key-words

ACTIVITY (see p. 23): Distributed and parallel active entity. It i a set of actors.

ACTOR (see p. 13): Local and sequential active entity.

COLLOQUY (see p. 24): Occurence of execution of a protocol.

ENTRY-POINT (see p. 17): Call point for a treatment in a module.

GENERIC_NAME (see p. 34): Name which designates entities with a given common functional property.

GLOBAL_NAME (see p. 34): Name of an entity which has identical meaning wherever it is used in the distributed system.

LINK (see p. 21): Designation in an actor of the access path to an object.

LINKAGE_CONTROL_PROCEDURE (see p. 21): Procedure which permits the kernel to determine whether an actor is entitled to establish a link to an object.

LOCAL_NAME (see p. 34): Name of an entity whose meaning depends of the context in which it is used.

MESSAGE (see p. 14): Information sent from one port onto another

port.

MODEL_OF_ACTOR (see p. 28): Collection of modules. It is used for the creation of an actor.

MODEL_OF_OBJECT (see p. 33): Description of the informations necessary for the creation of an object.

MODULE (see p. 27): Description of an undissociable set of treatments.

NAME (see p. 34): Means of designation for every entity in the system.

OBJECT (see p. 21): Passive entity manipulated by actors.

PORT (see p. 14): Actors send and receive messages through ports.

PRIMITIVE (see p. 13): Name of a function realized by the kernel.

PROCESSING-STEP (see p. 17): Elementary execution of an actor, triggered by a message.

PROTOCOL (see p. 24): Definition of the behaviour of a module in a communication.

RECEIVE_CONTROL_PROCEDURE (see p. 16): Procedure which controls the validity of messages received by an actor

SELECTION_PROCEDURE (see p. 17): Procedure which selects the next message to be processed by the actor.

SEND_CONTROL_PROCEDURE (see p. 16): Procedure which controls the validity of messages sent by an actor

SERVICE (see p. 29): Function realized by one or several modules, possibly on several sites.

SITE (see p. 13): Physical localization unit.

SPECIFIC_NAME (see p. 34): Name which designates one and only one entity.

START-UP_PROCEDURE (see p. 29): First treatment executed by an acto at its creation.

SWITCH_PROCEDURE (see p. 18): Procedure which selects th entry-point which will process the message.

## 12.2/ Primitives

This section presents the primitives of CHORUS. Each primitive is followed by its parameters.

CLOSE_PORT

Name of a port

CREATE_ACTOR

Name of a site on which the actor is to be created

Name of a model of actor from which the actor is to be created

Start-up message

Name of the actor

CREATE_OBJECT

Name of a model of object

Parameters to initialize the object

Name of the object

DESTROY_OBJECT

Name of an object

END_ACTOR

None

KILL_ACTOR

Name of an actor

LINK

Name of an object

OPEN_PORT

Name of a port

Send-control-procedures associated with the port

Receive-control-procedures associated with the port

RETURN

None

SELECT

Name of a port

Condition for messages received on Port to be selected

SEND

Name of a source port

Name of a destination port(s)

Message

SWITCH

Name of a port

Name of an entry-point

TIME_OUT

Name of a port

Value of the time-out

## 13/ Bibliography

[AND 78] Andre E., Decitre P.
On Providing Distributed Applications Programmers with Control Over Synchronization.
Proc. of the Symposium on Computer Network Protocols, Danthine editor, Universite de Liege (February 1978)

[BAN 79] Banino J.S., Kaiser C., Zimmermann H.
Synchronization for distributed systems using a single broadcast channel.
First Int. Conference on Distributed Computing Systems, Huntsville (October 1979)

[BIR 73] Birtwistle G., Dahl O.J., Myhrhaug B., Nygaard K.
Simula Begin.
Auerbach, Philadelphia, Pa., 1973

[BOC 79] Bochmann G.V.
Architecture of distributed Computer Systems.
Lecture Notes in Computer Science, number 77, Springer verlag (1979)

[BRI 78] Brinch Hansen P.
Distributed Processes: a Concurrent Programming Concept.
CACM, vol 21, 11 (November 1978)

[CAS 77] Casey L., Shelness N.
A Domain Structure for Distibuted Computer Systems.
Proc. of the 6th ACM Symposium on Operating Systems Principles (November 1977)

[CER 75] Cerf V.G., Mc Kenzie A., Scantlebury R., Zimmermann H.
Proposal for an Internetwork End-to-End protocol.
Int. Network Working Group, note 96 (September 1975)

[DAL 78] Dalal Y.K., Sunshine C.A.
Connection Management in Transport Protocols.
Computer Networks, vol 2, 6 (December 1978)

[DAN 77] Dang Ng.X.
System and Portable Language Intended for Distributed and Heterogeneous Network Applications.
2nd Distributed Processing Workshop, Brown University, Providence, Rhode Island (1977)

[DIJ 75] Dijkstra E.W.
Guarded Commands, Non determinacy and formal derivation of Programs.
CACM, vol 18, 8 (August 1975)

[ENG 74] England D.M.
Capability concept mechanisms and structure in System 250.
International workshop on Protection in Operating Systems, IRIA, Rocquencourt, France (August 1974)

[FAB 74] Fabry R.S.
Capability-Based Addressing.
CACM, vol 17, 7 (July 1974)

[FAR 72] Farber D.S., Larson K.C.
The system architecture of distributed computer system: The communication system.
Symposium on Computer Networks, Brooklyn (April 1972)

[FER 76] Ferrie J., Kaiser C., Lanciaux D., Martin B.
An extensible structure for protected systems design.
The Computer Journal (November 1976)

[FLE 78] Fletcher J.G., Watson R.W.
Mechanisms for a Reliable Timer-Based Protocol.
Computer Networks, vol 2, 4-5 (September/October 1978)

[GAU 80] Gaude C., Kaiser C., Langet J., Palassin S.
Distributed Processing as a key to Reliable and Evolving Software for Real Time Applications.
Proc. of the IFIP Congress, Tokyo, Melbourne (October 1980)

[HEW 77] Hewitt C.
Synchronization in Actor Systems.
4th ACM Symposium on Principles of Programming Languages, Los Angeles (January 1977)

[HOA 78] Hoare C.A.R.
Communicating Sequential Processes.
CACM, vol 21, 8 (August 1978)

[ING 79] Ingalls D.H.
The Smalltalk-76 Programming System; Design and Implementation.
Fifth Annual ACM Symposium on Principles of Programming Languages

[ISO 79] I.S.O.
The Reference Model for Open System Interconnection.
Document ISO/TC97/SC16 N227 (June 1979)

[JON 79] Jones A.K., Schwans K.
Task forces: Distributed Sofware for Solving problems of Substantial Size.
Proc. of the 4th Int Conference on Software Engineering, IEEE (1979)

[KAH 76] Kahn G., Mac Queen D.B.
Coroutines and Networks of Parallel Processes.
Rapport Laboria Numero 202 (1976)

[KAI 78] Kaiser C., Langet J., Poitvin J.F.
Design of a continuously available distributed real-time system.
Congres FTCS, Toulouse (Juin 1978)

[LAM 74] Lampson B.W., Mitchell, Sathewait
Transfer Control between contexts.
Lectures Notes in Computer Sciences, Springer Verlag (1974)

[LAM 78] Lamport L.
Time, clocks and the ordering of events in a Distributed System.
CACM, vol 21, 7 (July 1978)

[LEL 77] Le Lann G.
Distributed systems: Towards a formal approach.
IFIP Congress (1977)

[LEL 79] Le Lann G.
Algorithms for Distributed Data Sharing Systems which use tickets.
Proc. of the 3th Berkeley Workshop on Distributed Data Management and Computer Networks (August 1978)

[LIS 74] Liskov B., Zilles S.
Programming with abstract data types.
SIGPLAN Notices, vol 9, 4 (April 1974)

[LIS 79] Liskov B.
Primitives for distributed computing.
7th Symposium on Operating Systems Principles, Pacific Grove, California (December 1979)

[MET 76] Metcalf R.M., Boggs D.R.
ETHERNET: Distributed Packet Switching For Local Computer Networks.
CACM, vol 19, 7 (July 1976)

[OUS 79] Ousterhout J., Scelza D., Sindhu P.
Medusa: an Experiment in Distributed Operating System Structure.
Proc. of the 7th Symposium on Operating System Principles, CACM (1979)

[POU 73] Pouzin L.
Presentation and Major Design Aspects of the Cyclades Computer Network.
Proc. of the 3th ACM Data Communications Symposium (November 1973)

[POU 74] Pouzin L.
Cigale, the packet switching machine of the Cyclades computer network.
IFIP Congress, Stockholm, (August 1974)

[POU 76] Pouzin L.
Names and objects in heterogeneous computer networks.
1st Conference of the European Co-operation in Informatics, Amsterdam (August 1976)

[REE 79] Reed D.P.
Implementing Atomic Actions on Decentralized Data.
7th Symposium on Operating Systems Principles, Pacific Grove, California (December 1979)

[SWA 77] Swan R.J., Fuller S.H., Sieworek D.P.
CM*: a Modular Multi-microprocessor.
AFIPS NCC, vol 46 (1977)

[WUL 72] Wulf W., Bell C.G.
C.mmp - a multi-mini-processor.
Proc. AFIPS, Fall Joint Comp. Conf. (1972)

[WUL 74] Wulf W. et al.
HYDRA: The kernel of a multiprocessor operating system.
CACM, vol 17, 6 (June 1974)

[ZIM 75] Zimmermann H.
The Cyclades End-to-End protocol.
Proc. of the 4th ACM/IEEE Data Communication Symposium, Quebec (Octobe
1975)

[ZIM 77] Zimmermann H.
The CYCLADES experience - results and impacts.
IFIP Congress, Toronto (August 1977)