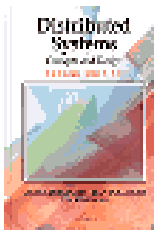


Originally published at pp. 566-79 of Coulouris, Dolllimore and Kindberg, Distributed Systems, Edition 2, 1994.

Chorus

History and architectural overview

Chorus began life in 1979 as a research project on distributed systems at the Institut National de Recherche en Informatique et Automatique (INRIA) in France. The goal of the project was to develop a message-based computational model for constructing a modular distributed operating system. Chorus went through three design phases at INRIA, and three corresponding versions of the distributed operating system emerged. In version 0 a model of communicating processes called *actors* was developed and a prototype implementation was made using a small kernel. In version 1 the previous design was ported from a LAN-based system to a distributed memory multiprocessor. For version 2 a team that had been working on implementing UNIX joined the project, and an attempt was begun to emulate UNIX using the Chorus kernel and re-using some of the code written by the UNIX team. When the project ceased at INRIA, a company, Chorus Systèmes, was set up to continue the development of Chorus on both LANs and multiprocessors. The Chorus kernel (version 3), also called the *nucleus*, and a UNIX emulation built on top of it, Chorus/MiX, are now being produced and developed by Chorus Systèmes [Rozier *et al*. 1988, 1990]. We describe Chorus version 3.3 in this section.

A Chorus system consists of uniprocessor or multiprocessor computers connected by a network. Chorus is architecturally similar to Mach in many ways. The Chorus kernel is a microkernel aimed at supporting subsystems. A Chorus subsystem is a collection of servers which provide a binary emulation of an operating system (UNIX in particular), or which provide some other major service to applications, such as the run-time support for a language. Generic run-time support for object-oriented languages on top of the Chorus kernel is the subject of research [Lea *et al*. 1993]. At the time of writing, the kernel has been implemented on the Intel 80386, Motorola 68030 and 88000 microprocessors, and Transputers, among others. A binary emulation of System V Release 4 UNIX exists for Intel 80386-based and Motorola 88000-based computers, and a BSD 4.3 UNIX emulation is being implemented.

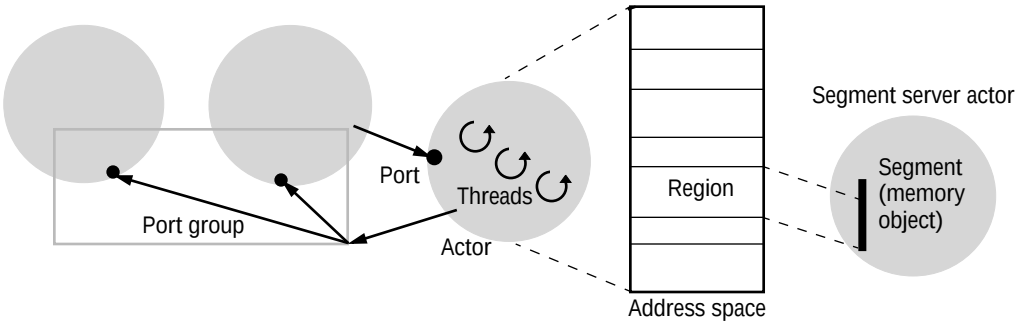
Chorus has been implemented as a basis for real-time process control systems running on embedded distributed memory multiprocessors based on 68020, 80386 and Transputer microprocessors.

Design goals and chief design features

Chorus has the following design goals in common with Mach (see Section 18.1 of edition 3):

- microkernel support for open system services, accessed by message passing;
- support for binary-level operating system emulation (in particular the emulation of UNIX) and other subsystems;
- transparent extensibility of kernel facilities to network operation;
- flexible virtual memory implementation;

Figure 1 The main Chorus abstractions (messages and local caches not shown).



- portability (the Chorus kernel is written in C++ and designed to be modular and split into machine-dependent and machine-independent parts);
- exploitation of shared memory multiprocessors.

Chorus also has the following goals and features:

Dynamically loadable servers: Chorus aims to achieve the same degree of modularity and openness as does Mach. Chorus supports dynamically loadable servers which may execute either at user-level or within the kernel address space.

Enhancement of UNIX: The Chorus design anticipates that users of the UNIX emulation might want to use enhanced facilities provided by the underlying kernel from within UNIX processes, such as multiple threads and the ability to create a new process at a remote computer.

Support for server groups and server reconfiguration: Chorus provides support for server groups in the form of group addressing modes for sending messages, including multicast. Port migration can be used to transfer management of a resource or collection of resources dynamically between servers, and is similar to the transfer of port receive rights in Mach.

Distributed memory multiprocessor operation: Chorus has been implemented on several distributed memory multiprocessors. Processors used in embedded multiprocessor systems may have relatively primitive hardware support for memory management. This has constrained the provision of features that assume the existence of sophisticated MMU hardware.

Real-time operation: The Chorus design aims to support real-time subsystems on the kernel. To this end, Chorus provides for flexible allocation of thread priorities and allows for customized thread scheduling policies; threads executing within the kernel can be scheduled pre-emptively.

Summary of the main Chorus abstractions

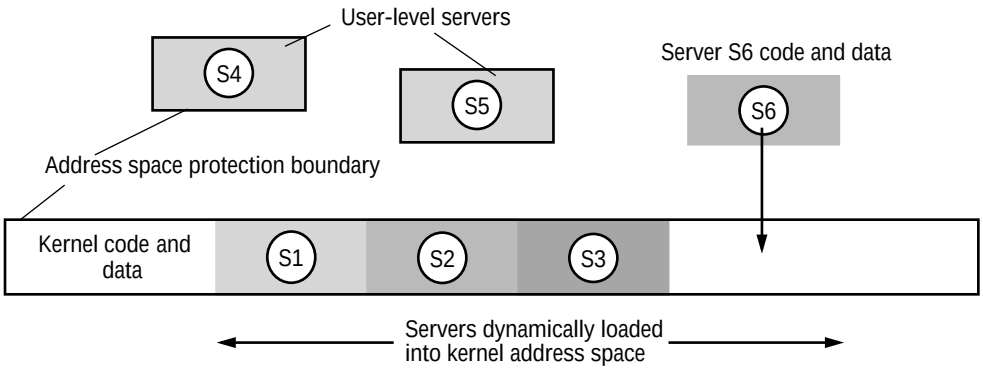
The main abstractions provided by the Chorus kernel (Figure 1) are as follows:

Actors: A Chorus actor is an execution environment equivalent to a Mach task. An actor can have one or more threads.

Ports: A port is a unidirectional communication channel with an associated message queue. Ports can be migrated between actors.

Port groups: Ports can be made members of port groups. A port group is a destination for messages, and there are several addressing modes for sending messages to a port group. Port groups should not be confused with the port *sets* of Mach.

Figure 2 Chorus servers.



Note: The figure shows the code and data for a server being dynamically loaded into the kernel address space, where it will execute. It also shows some user-level servers, which execute within private address spaces.

Messages: A Chorus message consists of a variable length body (limited to 64 kilobytes), and optionally a fixed-size (64-byte) header.

Regions, segments and local caches: An actor's address space is divided into regions, which are as we defined them in Chapter 6. A region can be mapped onto a portion of a *segment*, which is the equivalent of a Mach memory object. For each mapped segment the kernel keeps a *local cache*, similar to a Mach cache object.

The virtual memory design in Chorus is very similar to that of Mach and we shall not pursue this further even though its implementation is interesting [Abrossimov *et al*. 1989]. We now turn to the main design features that do differ from Mach.

Process management model

The basic processing building blocks in Chorus are *actors* and threads. An *actor* is similar to a Mach task and the chief components of its execution environment are an address space and a collection of ports used to receive messages. Actors can be dynamically loaded into the kernel address space, and their threads can execute in the kernel protection domain.

Servers are loaded dynamically at those computers where they are needed and are accessed by message-based communication. Servers are generally run as user-level processes to ensure mutual protection between the kernel and the servers it runs.

The price paid for this, however, is that of the extra context switches that occur in accessing user-level servers, compared to kernel-provided services. The Chorus designers decided that, in the case of some servers, the extra context switches incurred were too high a price to pay. They opted instead for an architecture in which:

- server code and data can be loaded dynamically at a computer as needed, and accessed via message passing, but
- a separate decision can be taken for each server as to whether the server program is added to the contents of the kernel's address space, or run in a private protection domain (Figure 2).

With this scheme, clients are unaware of whether a server with which they communicate is a user-level actor or is executing within the kernel. Clients and servers use the same message passing interface in either case. Moreover, the message passing interfaces can still be used between servers that reside in the same kernel address space. Inside the kernel, the implementation of message passing is designed to be efficient by taking advantage of shared memory to reduce data copying. This illustrates the fact that these servers use the message passing interface only as a convention.

Figure 3 Types of actor, showing address spaces and allowable threads.

Actor type	*Private address space*	*User threads*	*Supervisor threads*
User actor	Yes	Yes	Yes (created externally)
System actor	Yes	Yes (system privilege)	Yes
Supervisor actor	No: shares kernel's	No	Yes

They can perform arbitrary accesses to one another's data if they so choose, or if they contain bugs. Note, however, that actors can be debugged at user-level. If their performance at user-level is deemed unsatisfactory, they can be run later in the kernel, without altering the source code, to gain the performance advantage this brings.

Chorus supports three different levels of privilege for threads accessing local resources. Privilege is sometimes nominally ascribed to actors, but since only threads can take actions with respect to resources, privilege really resides with threads. The three level of privilege are:

user privilege: no direct access to machine resources; cannot invoke certain system calls;

system privilege: no direct access to machine resources, but can invoke all system calls – similar to the level of privilege of a UNIX process bearing the 'root' user identifier;

supervisor privilege: complete access to machine resources.

We now describe these privileges in more detail, in the context of actors and threads.

Actors ◊ Every actor is either a *user actor*, a *system actor* or a *supervisor actor*. System actors and user actors have their own separate address spaces, and may not access that of the kernel. However, the threads belonging to a system actor may make privileged system calls not available to those of user actors. For example, unlike a user actor, a system actor's thread can insert a port into a port group used to provide a system service. System actors can be created only by threads belonging to existing system actors, or by threads with supervisor privilege. At system initialization, one or more system actors are created at each computer, which may create further system actors. The kernel can check whether an actor is a system actor: its status is held securely in a table in the kernel address space.

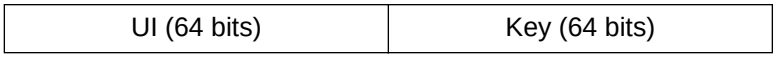
Supervisor actors differ from the other two types in that their code and data reside in the kernel address space. Any program tested as a user-level actor can be run without source-level alteration as a supervisor actor. However, in order to execute within the kernel, the binary code has to be link-edited dynamically. There are two aspects to this. First it is necessary for kernel system calls to be replaced by calls to kernel procedures. Since threads belonging to supervisor actors execute in the kernel, they should take advantage of the cheapest available invocation mechanism. Second, space has to be found in the kernel address space where the supervisor actor's code and data will reside. Neither the server program's address space location, nor the addresses of the kernel procedures that it calls are known in advance. Therefore all absolute program addresses used in the server program must be set dynamically using load-time location information.

Note that not all supervisor actors can be run at user-level – even as system actors. This is because the code for a supervisor actor can, for example, contain privileged machine instructions for manipulating hardware registers. Attempting to execute these instructions at user-level would result in a hardware exception. The three types of actor and their main properties are shown in Figure 3.

Creating actors ◊ All actors are created with only one port – the so-called *default port*, to which operations upon the actor are sent. A brand new actor consists of very little state: it has no threads and, unlike a Mach task, it always has an empty address space. However, some initial characteristics are defined: actors are created as children of existing actors, from which they inherit, for example, their scheduling priority.

The system call to create an actor is *actorCreate*:

Figure 4 A Chorus capability.



actorCreate(actorInit, actorCap, type, status)
creates a new actor as a child of *actorInit*, with type *type* (*USER*, *SYSTEM* or *SUPERVISOR*). A capability for the new *actor* is returned via *actorCap*; *status* specifies whether or not all new threads in the new actor are to be suspended initially.

(The structure of capabilities in Chorus is described below.) A new actor always resides at the same computer as its parent, so there is no need to specify a computer. The creation of actors at remote computers is left to subsystems to implement.

User and supervisor threads ◊ The threads belonging to system actors are privileged in that they may make certain system calls. This does not imply any special privileges with respect to direct access to hardware resources such as physical memory and device registers. However, threads belonging to any type of actor may execute as *supervisor threads*, which execute in the address space of the kernel, and with the processor in supervisor mode. They therefore have unlimited access to hardware resources. Note that all the threads belonging to a supervisor actor have to be supervisor threads, since their code is mapped into the kernel.

The system call *threadCreate*, which follows, is used to create a thread in an actor:

threadCreate(actorCap, privilege, status, priority, entry, stackPointer)
creates a new thread in an actor specified with the capability *actorCap*, with privilege *privilege* (*USER* or *SUPERVISOR*) and with scheduling priority *priority* relative to the process. The initial program counter and stack pointer are *entry* and *stackPointer*.

If *privilege* has the value *USER*, then *entry* and *stackPointer* are addresses in the address space of the given actor, which must be a user actor or system actor. If *privilege* is *SUPERVISOR*, then *entry* must be an address of an instruction in the kernel address space (normally the address of a kernel procedure).

System actors can create supervisor threads. How can a user-level actor come to know the address of a kernel procedure? The address has to be looked up from the procedure's name in the symbol table of the kernel. However, the stack pointer for a supervisor thread is allocated by the kernel, and the value given in *stackPointer* is ignored in this case.

Threads are scheduled pre-emptively by the kernel according to individual thread priorities, but these priorities can be set dynamically by actors. Chorus supports two-level scheduling. Thread priorities are set relative to their actor's priority, and actors are assigned absolute priorities. A thread's absolute priority is the sum of these two priorities. Even supervisor threads can be scheduled pre-emptively, in order to meet real-time demands. By contrast, conventional implementations of UNIX schedule processes executing kernel code without pre-emption.

Naming and protection

Chorus uses *capabilities*, *unique identifiers* and *local identifiers* as basic names for resources.

Capabilities: These are the most general type of resource identifier in Chorus. They are used for identifying and restricting access to resources such as segments managed by servers, and also some resources managed by the kernel itself, notably actors and port groups. A capability consists of a unique identifier, which is normally the identifier of a port, and an additional 64-bit structure called a *key* (Figure 4). The key can be used to identify a resource

from among multiple resources accessed via the same port. Servers choose the key so it is hard to guess, and thus provides a degree of protection against illegal accesses to resources.

Unique identifiers: Ports and other resources managed by the kernel are assigned 64-bit *unique identifiers*. Unique identifiers (UIs) are guaranteed to be unique within a network of Chorus computers, over its lifetime. They are fabricated as bit strings with three components: the type of the kernel resource identified by them (for example, port, actor or port group), the identifier of the computer that created it and a local timestamp guaranteed to be unique over the lifetime of the computer.

Local identifiers: Local identifiers are used to name the threads and ports belonging to an actor. They are 32-bit integers which are valid only within the actor which uses them. An actor's port can have aliases: its local identifier and its (globally valid) UI. Using the local identifier is the most efficient, however: local identifiers are generated by the kernel for fast access to the resources named. They are similar in this respect to Mach's identifiers.

Identifying resources managed by groups ◊ In order that a service can be provided by different servers at different times, or so that a service can be implemented at any one time by several servers, the ports that processes use to receive requests can be collected in groups. Messages can be addressed to port groups as well as ports, in one of several group-addressing modes.

The system calls for creating and manipulating port groups are as follows:

---

grpAllocate
Allocate a capability for a port group.

grpPortInsert
Insert a port into a port group.

grpPortRemove
Remove a port from a port group.

In order to manipulate the membership of a port group, an actor needs a capability for it. Capabilities are allocated dynamically for port groups, via calls to *grpAllocate*. This call can be used either to obtain a capability for a well-known port group, or to allocate a capability for a brand new group. Any actor that knows the group capability can insert a port into the group, using *grpPortInsert*, or remove a port, using *grpPortRemove*.

For sending purposes, only a port group's UI is required, and not the capability. Group identifiers provide a level of indirection, so that an actor using a group identifier does not need to know which ports belong to the group. When a message is sent using a group identifier, the sender selects one of the following addressing modes:

Multicast mode: An attempt is made to deliver the message, unreliably, to all members of the group (this is also known as *Broadcast mode*).

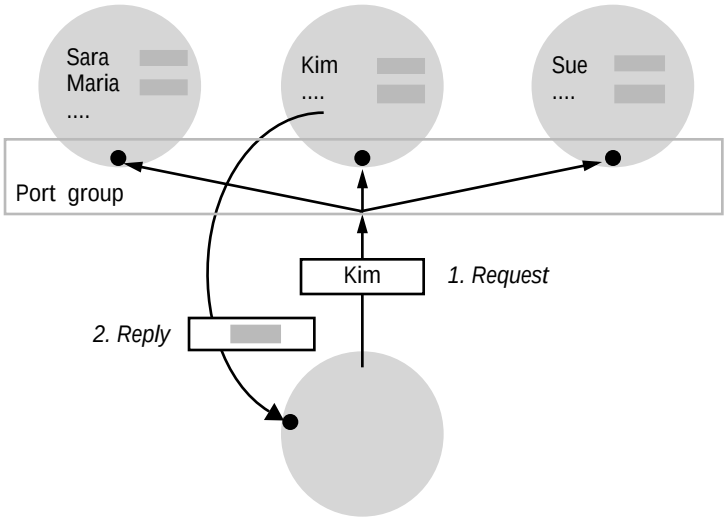
Functional mode: The message is delivered to at most one member of the group, but the member chosen is undefined in advance.

Selective functional mode: In this mode, the sender has to include a UI as well as the group capability. The message is delivered to at most one member of the group, one which exists at the same computer as the resource with the given UI.

We now provide illustrations of how these addressing modes are used to access resources managed by groups of servers.

Multicast mode ◊ Consider the problem of requesting an operation upon a resource that is managed by some member of a group of servers, but it is not known which. The request can be multicast in a message containing a service-specific identifier of the resource. This can be a low-level system identifier, or, for example, a file pathname. Upon receipt of the message, only the

Figure 5 A message multicast to a group.



server that manages the resource will perform the operation; the other servers can ignore the request (Figure 5).

A significant disadvantage of this scheme is that all processes in the group have to receive the message, even if it is not relevant to them. This arrangement would be impractical for the processing of all operations on resources managed by the group. It is preferable for clients to request a capability for the resource initially, using multicast. The capability returned by the server that manages the resource will contain the identifier of a port, and all the client's subsequent requests can be sent directly to the server that owns this port. The initial multicast is effectively a name lookup: a capability is returned when the client presents a resource name to the group.

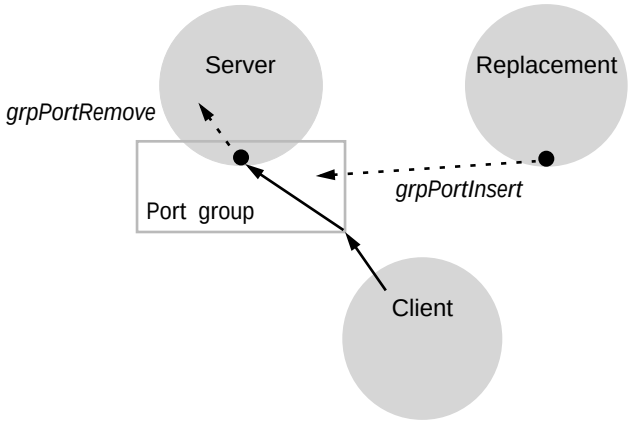
Note that the Chorus multicast service is primitive in that it is unreliable and provides no ordering guarantees. This is a reasonable choice for a microkernel, considering that higher-level requirements are liable to vary, and can be implemented on top of the Chorus mechanism. In Chapter 11 we described reliable and ordered multicast facilities.

Functional mode ◊ To illustrate the use of functional addressing, we now consider the problem of replacing a single server that provides a given service – for example, to replace a server with an upgrade. One solution to this problem is for the service to be provided via a group of server ports, which normally has only one member. All requests are sent to this group of servers, using functional mode addressing. This addressing mode delivers the request messages to one member of the group – and there is only one member. The replacement server can join this group by inserting its port using *grpPortInsert*; the server to be replaced can leave the group by using *grpPortRemove* to remove its port from the group (Figure 6). For reconfiguration transparency to be achieved, care has to be taken to synchronize message processing so that client requests are processed consistently.

Note that message delivery under functional mode group addressing can be implemented as efficiently as unicast message delivery. Once a port belonging to the group has been located, messages can be sent using a unicast protocol to that port until such time as the port leaves the group. At that point, computers sending to the group will receive a negative acknowledgement notifying them that they need to search for another port in the group.

Selective functional mode ◊ The *selective functional* group addressing mode is useful for identifying a particular server from a group of servers. For example, a service that provides information about computer loads could be implemented using an actor at each computer, which monitors activity at that computer. Each such actor places a port in a common group. To find the load at a particular computer, a request can be sent to this 'load monitoring' group, using selective functional addressing with the UI of the required computer included in the address.

Figure 6 Functional mode addressing.



Each computer manages a database of UIs which is searched when attempting to deliver a message using selective functional addressing. Chorus includes in this database, by default, the UIs of the computer, all local actors and all local ports. In addition, an actor can declare a UI of its choice as belonging to its local database using *uiDeclare*, and it can remove it later using *uiForget*. This allows servers to declare themselves as being associated with particular resources for the purposes of selective functional addressing.

Transferring resources between servers ◊ We have already described how port groups are used to implement reconfigurable servers. There is in addition a *port migration mechanism* whereby a port can be removed from one Chorus actor and inserted in another. This transfers from one actor to another the ability to receive messages sent to the port. Using this mechanism, management of an individual resource – or a group of resources – can be transferred from one server actor to another. After a port migrates, all requests sent to it become queued for reception by the new actor. This is so, whether the senders of these requests use capabilities that contain the port's UI, or use the identifier of a group of which it is a member. As one would expect, ports remain members of port groups when they migrate.

This mechanism is equivalent in its effects to the transfer of port receive rights in Mach; it differs only in the fact that the actor at which the port initially resides is not required to take part in the migration. Note that port migration seems at first sight to be equivalent to placing one actor's port in a port group and removing the other actor's port. However, the group mechanism, unlike port migration, can be used even if the original port becomes unavailable due to a crash of its host computer. Under those circumstances, the original port is deemed automatically to have left the group. In favour of port migration, however, note that managing ports requires less memory overheads than managing port groups. Furthermore, the port migration mechanism offers control over continuity of request handling, which cannot be guaranteed using the group mechanism. Messages sent using functional mode addressing that were previously delivered to the original port may be delivered to *any* port in the group after this port is removed – and not necessarily the one intended as the replacement port.

Protection ◊ Chorus port identifiers and capabilities can be propagated freely in messages between processes, without intervention by the kernel. Port identifiers exist in a sufficiently large name space (they are 64-bit UIs) that guessing a port identifier that has been chosen at random is not practically feasible. Capabilities can be made hard to forge through the choice of their keys, and so resource protection can be applied even against actors that know the relevant port identifier.

However, capabilities cannot be used as the sole basis for a scheme to emulate UNIX *user-group-other* protection semantics. For this reason, Chorus undertakes a form of authentication on behalf of service implementors. That is, Chorus is able to identify securely the source of a message to a server that receives it. For example, in order to emulate UNIX, actors implementing UNIX processes can be associated with the equivalent of UNIX user identifiers.

To enable authentication to take place, a *protection identifier* is associated with each actor. An actor's protection identifier (PI) is by default that of the actor that created it; but it can be changed by supervisor threads or system actor threads. When an actor receives a message, it can request the kernel to specify the PI of the actor that sent it. A service could use this mechanism to implement access control for the resources it manages.

Of course, this mechanism provides security only if the association between actors and PIs is itself securely applied. At a single computer, the kernel can easily transmit PIs securely. An authentication protocol is required, however, to provide secure authentication across a network, in the face of possible eavesdropping, tampering and replaying. We discussed authentication protocols in Chapter 7.

The Chorus approach to ensuring that only legitimate servers can provide a service is to make port group membership secure. An actor may only add a port to a group if it possesses a capability for it. Although the capability may contain a well known group UI, the key part is chosen dynamically to be hard to guess. As long as port group capabilities are kept secret, actors cannot masquerade as system servers through the port group mechanism.

Communication model and implementation

Communication system calls ◊ Chorus provides the following main system calls related to communication:

---

ipcCall
Send a request and receive a reply.

ipcSend
Asynchronously send a message.

ipcReceive
Receive a message.

ipcReply
Reply to a message.

ipcSave, ipcRestore
Save/restore current message.

ipcGetData
Receive body of message.

ipcSysInfo
Return information about current message.

portCreate, portDelete
Create/delete a port.

portEnable, portDisable
Enable/disable a port.

portLi, portUi
Translate port's UI to/from local identifier.

Like Mach, Chorus provides primitives for asynchronous message passing as well as for request-reply interactions. *IpcSend* is used to send a message asynchronously. It can be given either a port UI as a destination, or a group identifier with any addressing mode. *IpcCall* sends a request message and awaits a reply. It can be given any type of destination as an argument except a group addressed in multicast mode. The exception is because *ipcCall* cannot support the processing of multiple copies of a request, or the multiple reply messages that may ensue.

Threads receive requests using *ipcReceive*, which can be given either a local identifier of a port as the reception interface, or a special flag signifying that a message is to be received only from

a member of a set of local ports. After receiving a message, a thread can additionally call *ipcSysInfo* to find out information about the message such as the protection identifiers associated with it, before deciding whether to process and reply to it.

When a port is created, using *portCreate*, the kernel allocates and returns a UI for the port. *PortLi* is used to convert the port's UI into a local identifier which can be used with *ipcReceive*. As we explained above, the local identifier is used to gain rapid access to the internal port data structure. Chorus is inconsistent in not allowing the UIs of ports belonging to other actors to be converted to LIs, as might be done for the purposes of efficient message sending. However, the identifiers of these ports may be sent in messages, and only UIs will do for this purpose, since LIs are only valid in the context of a single actor.

By setting an appropriate collection of ports into the *enabled* state, a thread can receive a message from what is the equivalent of a Mach port set. However, there can be only one 'port set' per actor – that is, for all the threads within the actor. To be a member of this set, a port has to be *enabled*. Every port is either *enabled* or *disabled*, and actors can change the state of their ports at will, using *portEnable* and *portDisable*.

Threads reply to requests using *ipcReply*. Normally, the reply is to the last message received by the thread. However, if they are able to request input-output without blocking, threads can postpone replying to a request until the input-output has been performed, and meanwhile respond to other requests. This is useful when, for example, a request arrives for some data that must be fetched from a disk, but when the following request, from a different client, can be serviced without a disk access. Replying out of order is achieved using the concept of *current message*. There is at most one current message per actor at any one time. By default, the current message is the last one received. A thread can save the current message using *ipcSave*, and then receive and reply to further messages. When the thread is ready to reply to the saved request, it can restore it to be the current message, by using *ipcRestore* and quoting a message identifier supplied by the kernel at the time it was saved. *IpcReply* is then used to reply to this message.

Messages ◊ Chorus, unlike Mach, uses simple messages consisting of at most a fixed-size *header* and a variable-sized body. A thread can use *ipcGetData* to extract the body of a message after receiving its header and determining from the data within it the reception buffer to use.

Chorus uses copy-on-write to transfer large message bodies efficiently when *ipcSend* is used, if MMU support for this is available. Unlike Mach, however, Chorus does not generate a new address space region in the receiver as a by-product, but always uses the address range specified by the receiver. There is also an option that can be used with *ipcSend* that causes the pages used to implement the body of a message to be transferred to the receiving actor's address space (assumed to reside at the same computer) – and removed from the sender's address space. This allows the message to be transmitted entirely by page table manipulations. Moving rather than copying data between address spaces in this way is particularly useful where, for example, a message is being forwarded and the sender has no further need for the message body.

The network manager ◊ The Network Manager is an actor that extends the communication facilities of the kernel transparently across a network, and is in this respect similar to Mach's network server. The Network Manager is responsible both for message transport and for port location. When a thread presents a port UI to the kernel to send a message, this UI is looked up on a list of ports known to be local. If it is not found there, then the port's UI is automatically forwarded by the kernel to the Network Manager, which attempts to locate the port by communicating with other network managers. Once the port has been located, messages sent to the port thereafter are delivered directly to a port belonging to the Network Manager, which forwards them transparently. Similarly, the Network Manager is responsible for locating members of port groups residing at computers across a network from the sending computer.

Discussion of main Chorus features

In summary, the Chorus microkernel is aimed at the support of open services, operating system emulation – particularly System V UNIX – and other subsystems in a distributed system. It runs

on network-based distributed systems and on distributed memory multiprocessors. Its scheduling architecture is designed to support real-time systems.

The Chorus kernel provides: multi-threaded processes called actors; communication using ports and groups of ports as destinations; and sophisticated use of virtual memory allowing regions to be backed by external pagers.

The port group and port migration facilities allow for services to be implemented by server groups. However, the level of support for groups is limited to services in which resources are partitioned between servers. Stronger multicast semantics are required when resources are replicated between servers to achieve high availability. A resource or group of resources can be migrated dynamically from one server to another using port migration or port group manipulations. However, although we did not mention this earlier, achieving resource migration transparently in practice involves considerable extra work in transferring resource state between the two servers involved, and synchronizing them.

The communication facilities of Chorus are less dependent than those of Mach upon the existence of sophisticated MMU hardware, and are, by the same token, less flexible in terms of message structure.

The Chorus facility for allowing dynamically loaded servers to execute in the kernel address space is an attempt to improve performance, at the expense of losing hardware protection boundaries between such servers and the kernel. The facility has been utilized in the Chorus/MiX UNIX emulation subsystem, as will be seen in the following section.