MASTER IN COMPUTER SCIENCE

Dynamic Sub-Systems Management in a Closely Coupled Architecture

Vanderperre, Pascal

Award date:
1991

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix

NAMUR

Institut d'Informatique

The thesis

Dynamic Sub-Systems Management in a Closely Coupled Architecture

By Vanderperre Pascal

Promoter

Jean Ramaekers

Thesis presented in order to obtain the title of *Licencié et Maître en Informatique.*

Academic year 1990 - 1991

Abstract

Since the arrival of the distributed operating systems such as MACH and CHORUS with the Loosely Coupled or Closely Coupled multiprocessor systems, a new concept appeared : the server. In the common literature, the designers explain how to access a server service through the existing mechanism of ports and messages. However from the performance point of view, the state-of-the-art relates only few words about the strategies taken into account when (un)loading a server and when (un)binding a server.

In addition, there already was a concept of sub-systems among several operating systems. The first step of the present thesis has been to integrate the server as a kind of sub-system into a sub-systems architecture and into a sub-systems topology.

The scope of this thesis is to elaborate a management of Sub-Systems which permits to exploit the parallelism provided by the servers as efficiently as possible. My management strategies have been designed during a training period at Siemens-Nixdorf on the evolution of the BS2000 operating system to a distributed operating system in a closely-coupled multiprocessor system.

Depuis l'apparition des systèmes d'exploitation distribués tels que MACH et CHORUS pour des systèmes multiprocesseurs "loosely-coupled" ou "closely-coupled", un nouveau concept apparut : le serveur. Dans la littérature courante, les concepteurs expliquent la manière d'accéder à un service d'un serveur avec le mécanisme existant de portes et de messages. Mais d'un point de vue de performance, l'état de l'art ne révèle pas les stratégies considérées lors des (dé)connexions aux serveurs et lors des (dé)chargements de serveurs.

De plus, il existait déjà un concept de sous-systèmes parmi plusieurs systèmes d'exploitation. Une première étape du présent mémoire a été d'intégrer le serveur comme étant un cas particulier de sous-système à travers une architecture et une topologie de sous-systèmes.

Le but de ce mémoire est d'élaborer une gestion de sous-systèmes qui permet d'exploiter le plus efficacement possible le parallélisme offert par les serveurs. Mes stratégies de gestion ont été conçues lors d'un stage chez Siemens-Nixdorf pour l'évolution du système d'exploitation BS2000 en un système d'exploitation distribué à travers un système multiprocesseurs "closely-coupled".

Acknowledgements

First of all, I wish to express my thanks for his helpful and advice to Mr. Jean Ramaekers, computer science professor of the *Facultés Universitaires Notre-Dame de la Paix à Namur* (Belgium) and the promoter of this thesis.

I would like to thank the following members of the SIEMENS-NIXDORF company for giving me the right environment for my studies and for their helpful suggestions : Mr. Nicos Piperakis, Mr. Benoit Hucq, Mr. Bruno Bodart, Mr. Willy Messing and the members of their teams.

I thank also the SIEMENS-NIXDORF company and COMETT II program for giving me the financial possibilities to lead the project.

I dedicate this study to France, my sister.

Table of contents

PART 1

A hardware introduction

1- Parallelism and performance

The parallelism of different entities (e.g. programs, instructions, etc ...) brings about more performance with a multiprocessor system. But the concurrence of shared resources and the coherence of the replicated entities have to be secured ...

1.1- The multiprocessor systems

A tightly-coupled multiprocessor system is a hardware system with several processors, a shared memory and other common input-ouput devices such as the tape disks, the hard disks, etc ...

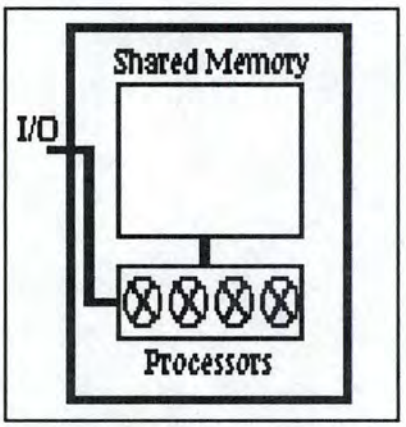

Figure 1 *The tightly-coupled multiprocessor system*

A loosely-coupled multiprocessor system is a hardware system with several nodes interacting through a communication network. A node, also called a computer, may be a uniprocessor system (i.e. IBM PC 80386) or a tightly-coupled multiprocessor system (i.e. Univac 1100/8, BS2000/Z11, IBM 370, CRAY X-MP). A node is always managed by a kernel (VAX-VMS, MS-DOS). It is possible for nodes to have different kernels but there always is a distributed system (i.e. UNIX BSD 4.2, MACH, CHORUS, V-system, AMOEBA) for all nodes. Let's examine the kind of the networks (i.e. ETHERNET, XEROX internet, TOKEN RING) which will allow interaction between the nodes.

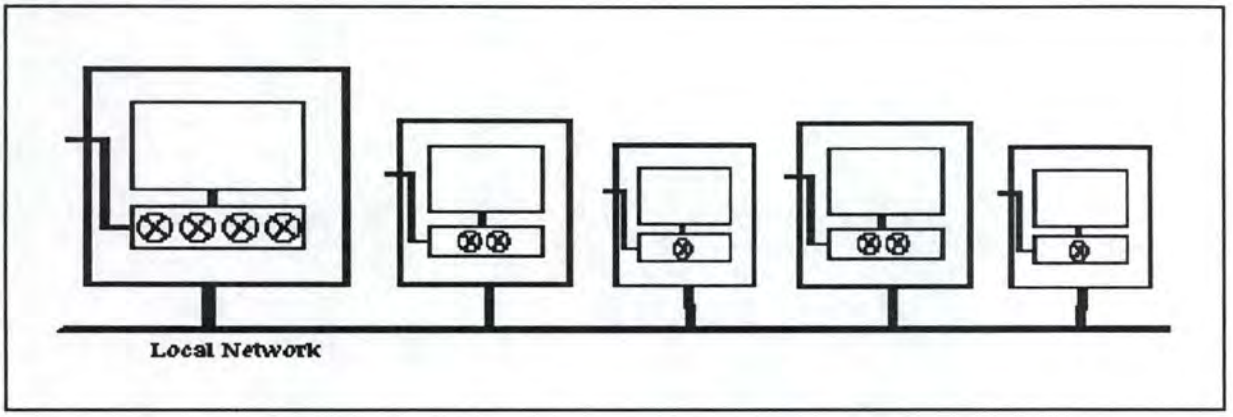

Figure 2 *A loosely-coupled multiprocessor system*

Loosely-coupled systems are characterized by physical separation of processors, low-bandwidth message-oriented interprocessor communication, and independent operating systems. Tightly-coupled systems are characterized by close physical proximity of processors, high-bandwidth communication through shared memory, and a single copy of the operating system. The intermediate approach is the closely-coupled structure also called cluster (i.e. VAX-Cluster [KRON]) : a cluster has separate processors and memories connected by a message-oriented interconnect, running separate copies of the same distributed operating system, a close physical proximity of all resources, a single security domain (physical and logical), shared physical access to disk storage, and a high speed memory-to-memory block transfer between nodes.

1.2- The level of parallelism

In the centralized operating system there are several levels of parallelism. The intra-instruction level results from the capabilities of the modern pipeline computer which are able to treat several instructions at the same time by using scalar and vector pipelines. The inter-instruction level allows the parallelism of instructions of a program which is written in a high level language. The multitasking level makes it possible to execute several programs (e.g. the user programs and the local sub-systems) at the same time.

In the distributed system there are two new levels of parallelism : the multithreading level which can perform several procedures of a program at the same time, and the nodes level which allows the parallelism of programs interacting through a communication network (e.g. the user programs and the servers).

2- Performance and cost

2.1- The failure of Grosch's law

Grosch's law stipulates that the performance is proportional to the square of the cost [COUL]. For example, a system of 2 million $ is two times more efficient than two 1 million $ systems and a network. Thus a tightly-coupled multiprocessor system is better than a loosely-coupled multiprocessor system for the same price. But this law doesn't apply to the new high speed networks, the cheaper memory and the cheaper processors. At present some designers put forward the following hypothesis : the performance is nearly proportional to the logarithm of the cost. In addition, users need more and more complex applications. The advantage of the loosely-coupled multiprocessor system is that it is able to offer a lot of heterogenous nodes, each node being specialized in a certain kind of application (e.g. a workstation for the interactive or graphical applications which are not made for an oriented batch node).

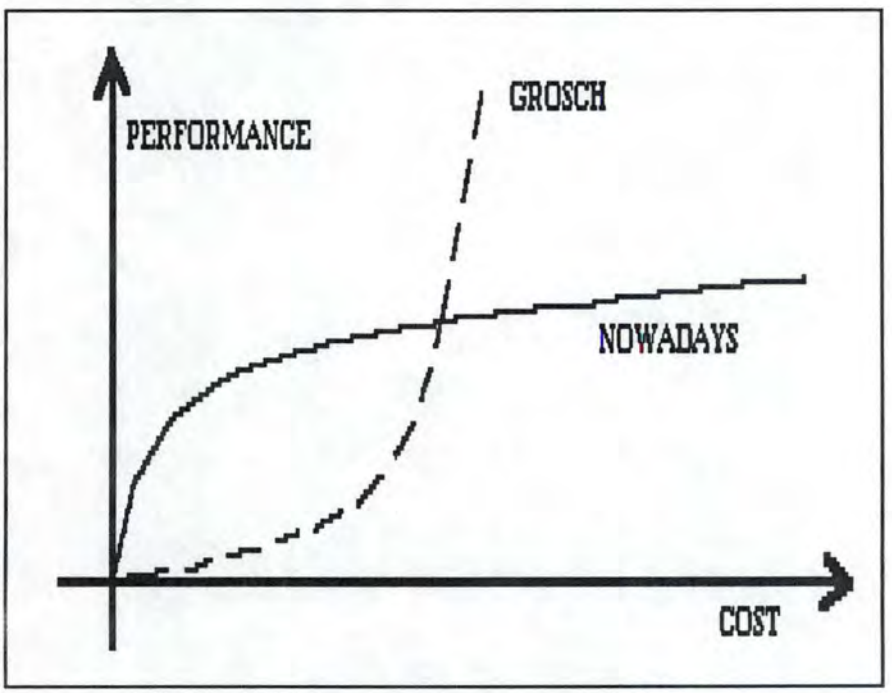

Figure 3 *The performance according to the cost*

3- Management of servers in a Closely Coupled Architecture

To obtain the optimal response time of the services of servers, the (un)loading and the (un)binding strategies have to be found. Part 3 of this thesis attempts to answer two fundamental questions regarding the Closely Coupled Architecture : Which is the node where a server may be loaded at a given time ? What is the copy of the server to be chosen during a binding ? It is advisable before tackling part 3 to read explanations about the Remote Procedure Call, the local and global services, the local or global call mechanisms, the Client/Server model, the Client/Sub-Systems Architecture, the topologies of Sub-Systems, some principles about the distributed operating systems ... Those concepts will be explained in detail in part 2 of this thesis.

PART 2

Services and Sub-Systems in a Distributed Operating System

Chapter 1 : Some goals and principles of the DOS

This chapter gives the main goals, principles and properties of a particular case of Distributed Systems : the Distributed Operating Systems in a Closely Coupled Architecture.

1- Definition of a Distributed System

A distributed system is a multiplicity of general purpose resource components which have a physical distribution interacting through a communication network [SEIF]. The Distributed System is the manager of the software as well as the hardware resource components.

2- Kinds of operating systems in distributed systems

From a functional point of view, the operating system is split into a kernel and a set of sub-systems. The kernel comprises a lot of basic services such as the processor allocation and the memory management. The sub-system is another kind of set of services such as the file backup management. The server is a special kind of sub-system (the server services may be used whatever the localization of the user may be).

2.1- The Network Operating Systems

Each node has its own kernel and there is a common software package to be added to each host in order to communicate.

2.2- The Distributed Operating Systems

Each node has its own kernel. The Distributed Operating System is able to manage the resources through the network by using the servers. In fact, it is this kind of operating system this thesis will examine, but in the context of a closely-coupled multiprocessor system.

2.3- The decentralized operating systems

They are the same as the distributed operating systems but here the functions of the kernel are themselves distributed. In other words the services of the kernel are split among all nodes.

3- Goals of the DOS

Obviously, the main goal of the Distributed Operating Systems is to share a lot of resources (i.e. the files, the printers, ...) whatever the node they are associated with, but it has to be carried out by providing the transparancy of the service utilization (the single image machine) and by insuring the effectiveness of those services.

The single image machine principle of DOS

The users should have a view of the system as a whole, they should not normally need to be aware of the location of hardware and software components from which the system is constructed.

4- Properties of the DOS

4.1- The transparancy

A user uses a distributed service, and thus a server, without knowing its location, its possible replication, its possible migrations on other node, its possible failures and recoverabilities, its possible existing activities and without knowing the configuration of the physical components and its possible extensions.

4.2- The performance

The DOS has to optimize the weight of each node with the minimum network traffic rate because that implies a greater performance for the response time per service. It also has to furnish some transport communication services which are to be more efficient because they are more lightweight.

4.3- The reliability

Because DOS is very dependent on the network it has to recover the network failure and has to make the whole system reliable.

4.4- The fault tolerance and the recoverability

DOS has an inbuilt automatic recovery system for hardware or software failures. For example, a node may crash yet that doesn't hamper the users of other nodes.

4.5- The security and the extensibility

There are no interruptions of the user activities if new hardware or software components are added. This property is very interesting when we speak about the open system. But it also increases the risk of getting undesirables components such as spies or something like that. Therefore DOS has to protect the whole system against accidental or intentional violation of access control and privacy constraints.

4.6- The coherence

There are often several copies of a logical entity (i.e. a server, a file, ...) and so DOS has to secure the coherence of this entity.

4.7- The concurrency

As one of the primitive goals of Distributed Systems is to share the resources (i.e. a server, a file, a printer ...) among several users, DOS has to manage this concurrency.

Chapter 2 : How to achieve services in DOS

In a Distributed Operating System there are several ways to carry out a service for a client : there are those of the traditional operating system and in addition the new ways required by Distributed Systems. Generally speaking the sub-system concept is a set of services and the client concept is an entity which has need for a service. The aim of this chapter is to provide the basic elements of the Distributed Operating System in order to explain how to achieve the service.

1- The representation of the client or of the sub-system

In the distributed operating systems a client or a sub-system is represented by a particular kind of process [1]: the task.

The task

The tasks represent virtual central processors competing for the possession of the central processor. The tasks are submitted to the task scheduling according to priorities, even though two or more tasks may be executed simultaneously when two or more central processors are available. The program components, an address space and additional resources are assigned to the task. The task is the support of a set of performing units. The task is privileged or not depending on whether the current performing units are privileged or not. A task may be a user task or a system task. A system task is a task for the operating system and in particular for the sub-systems. The user task is a task for the user programs. The tasks management (the (des)activation, the initiation, the creation, the terminaison, the preemption and the scheduling of the task) belongs to the kernel.

---

[1] In the BS2000 operating system there are several kinds of processes such as the special process, the task and the activity. "Processes represent virtual central processor and thus compete independently of one another for possession of the central processor" [SIM1]. Only the tasks are the support for the clients and the sub-systems, therefore the clients and the sub-systems are submitted to the multitasking. The only kind of process in UNIX BSD 4.3 is the task with only one thread. The subject of process types is very specific and depends on the operating system and so we do not here introduce it here because this chapter wants to stay as general as possible. More over we speek here about the inter-task communication instead of the inter-process communication throughout the thesis for referring to the theorical concept, MACH or BS2000 naming.

2- The representation of service

There are several kinds of services used by a client : the kernel, the local and the global services. In the distributed operating systems those services are represented by different kinds of program-runs. The program runs are the execution units for a program within a task. Their context is a subset of the task context. There may be several program runs within the same task. A program run is privileged if its code contains at least privileged instructions. The program run management belongs to the kernel.

2.1- The kinds of services

2.1.1- The kernel services

The Kernel services are the generic or basic services. The kernel services are the services provided by the traditional operating system and they also exist in the distributed operating system. They are included within the kernel. In addition the kernel has some new services for securing the distribution such as the services of the transport management, the management of distributed files, etc ... Thus a kernel service may be called with the SuperVisor Call by the clients which are on the same node of the corresponding kernel. Concerning the kernel, services or their interfaces are the only things which will be examined in more detail in this thesis. More over, the kernel services have to be submitted to a deeper analysis when the distributed system is decentralized ...

2.1.2- The local services

The local services are the services provided by the traditional operating system and they also exist in the distributed operating system. They are included within the local sub-systems. Therefore, a local service may be used by the clients which have to be on the same node of the corresponding local sub-system. The local sub-systems may be loaded or unloaded depending on the utilization of their services.

2.1.3- The global or distributed services

The global services are included within the distributed sub-systems (the servers). Therefore, a global service may be used by all of the clients wherever they are. The servers may be loaded and unloaded according to the utilization of their services.

2.2- The types of program-runs

2.2.1- The Program Control Bloc runs

The PCB runs are used for synchronous processing within a task. The PCB run has no address space of its own and it also shares all of the other important resources with all of the other program runs of its task. Two or more PCB runs of the same task may be executed alternately only and two or more PCB runs of different tasks may be executed simultaneously when the tasks are executed simultaneously.

2.2.2- The threads

The threads are used for asynchronous processing within a task. The thread has no address space of its own and it also shares all of the other important resources with all of the other program runs of its task. Two or more threads of the same task may be executed simultaneously. The parallelism of threads within a task is also called the multithreading. For the purpose of bounding this thesis we shall put forward the following hypothesis : the process manager of the kernel is able to treat the threads and the procedures but not the PCB runs because we don't know beforehand whether it is possible to manage those two different kinds together, or in other words whether the threads and the PCB runs are compatible.

2.2.3- The procedures

The procedure is a code which is still performed in the context of the caller (the client). This is possible because the distributed operating system defines a shared code in system address space for all of the tasks, and also defines a common memory pool between all of the tasks. The common memory pool and the shared code are available only in the context of the node. In that case the procedure may only represent a local service. The common memory pool is almost the same things as the shared code but there is still an "undirection" of address when a client makes a branching to this common memory pool. Indeed, the common memory pool permits the opening of a window for the caller address space in the callee address space. In other words the code is loaded in another address space but the caller has the impression that the code is in its own address space. The management of the address "undirection" belongs to the kernel. There never is an "undirection" of address when a client makes a branching to the shared code in a system address space.

2.3- Context of execution

When a client requests a service from a sub-system and if the corresponding program run is a procedure then that program-run is performed within the task corresponding to the client. When a client requests a service from a sub-system and if the corresponding program run is a thread then the program-run is performed within the task corresponding to the sub-system.

3- The service call mechanisms

3.1- The local call mechanisms

The local call mechanisms are used to carry out a local service or a kernel service between a caller and a callee on the same node. The principal notions of those mechanisms are the address and the registers.

3.1.1- The SuperVisor Call

The supervisor call is used for calling a kernel service wich may be a service of the thread management, of the transport of message management, etc ... (e.g. a thread calls by using the SVC of another thread within the same task). The kernel service is responsible for saving and for restoring the registers which represent the current environment of the caller. The registers are the ties of communication between the caller and the callee. The caller and the callee may exchange the address of the arguments and of the results of the service. The kernel service is identified by a SVC number. The kernel service may be performed in the context of the caller or of the kernel.

3.1.2- The direct branching

The direct branching (BALR) is used for calling up a procedure according to its address. The caller routine or the callee routine are responsible for saving and for restoring the registers which represent the current environment of the caller. The registers are the ties of communication between the caller and the callee. The caller and the callee may exchange the address of the arguments and of the results of the service. In addition, one of the registers has to contain the return address.

3.1.3- The bourses and commissions

The bourses and commissions mechanism [SIM1] is a mechanism of local Inter-Task Communication. The bourses help to cope with situations where two or more tasks access a shared resource or are in a Client/Server relationship with each other on the same node. The bourses and commissions are used for exchanging a little message between to tasks of the same node. The bourses are shared resources with mutual exclusion for accesses. There also is a queue which is associated with the bourse for receiving the requests or the responses. The clients and the sub-systems may exchange commissions. A commission is a very small message of almost eight words which increases the transaction speed. The using of bourses and commissions is somewhat complex but it offers great results.

3.2- The global or distributed call mechanisms

The global call mechanisms are used to carry out a global service between a caller and a callee whatever their location may be. The principal notions of those mechanisms are ports and messages.

3.2.1- Ports and messages

The port and message mechanism is a mechanism of Inter-Task Communication. It is used for exchanging messages between two tasks (the client and the server) wherever they are. If those tasks are on the same node then the common memory pool or the shared code will be used as a communication support by the transport manager. The port of a server is a queue for receiving messages while being associated to a task corresponding to the server. The server is responsible for the creation and the deletion of its port. The message is the smallest communication entity between tasks. The message is a set of typed data. The transport manager (responsible for the message transport throughout the network) is included in the kernel. The presentation of the message may be a convention between the client and the service.

4- Call Modes

From a dynamical point of view there are mainly three modes for calling a service : the Normal Mode, the Interactive Mode and the SVC Mode.

4.1- The Normal Mode

It is the executable program of the client which directly uses the local or global call mechanism to obtain the service of a Sub-System. To do this it has to know the port or the address of the corresponding service.

4.2- The Interactive Mode

If a user wants to use the execution of a service in an interactive mode then he has to use the exec command : */EXEC command,parameters list*
In this case it is the program run corresponding to the command which uses the local or global call mechanism. This program run belongs to the user or client task.

4.3- The SVC Mode

The client may still use the service of a sub-system by using the supervisor call mechanism, with the SVC number to replace the service name and by using the registers to give the address and the length of the parameters. Here it is the Supervisor routine which uses the local or global call mechanism to obtain the service of the server. The supervisor routine is performed in the context of the client task.

Chapter 3 : Clients/Sub-Systems Architecture

It is important to define an architecture in which the Sub-Systems may be performed and to know the reasons of their existence. This architecture is not specific of a particular operating system.

Historical approach

At the end of the sixties the operating system designers had decided to construct the new operating systems using the abstract levels because they needed more structure for easier design the maintenance. Those levels define a hierarchy starting with the "use" relationship, each level containing a great number of modules. The modules are called sub-systems with kernel [1] at the first abstract level. The reciprocal actions between the modules are defined by a common protocol. This protocol involves the interface definition, the possible sequence of interface calls, etc ... In addition, this kind of construction gives a dynamical configuration of sub-systems thus allowing the operating system to evolve. The distributed system amplifies these necessities because the trend for the single image machine is to avoid the concept of the shutdown. It is possible to consider the server as the only kind of sub-system but it is not very realistic because an operating system architecture is often constructed in the incremental way...

Introduction

There are two kinds of sub-systems : the local sub-systems and the global sub-systems also called servers or distributed sub-systems. A client has access to local sub-systems through a local call mechanism such as the direct branching with the shared code but never with the global call mechanism of ports and messages. This implies that the local sub-system has to be on the same node as the client. The client has access to the servers through the global call mechanism of ports and messages.

[1] An other name for the kernel is the nucleus.

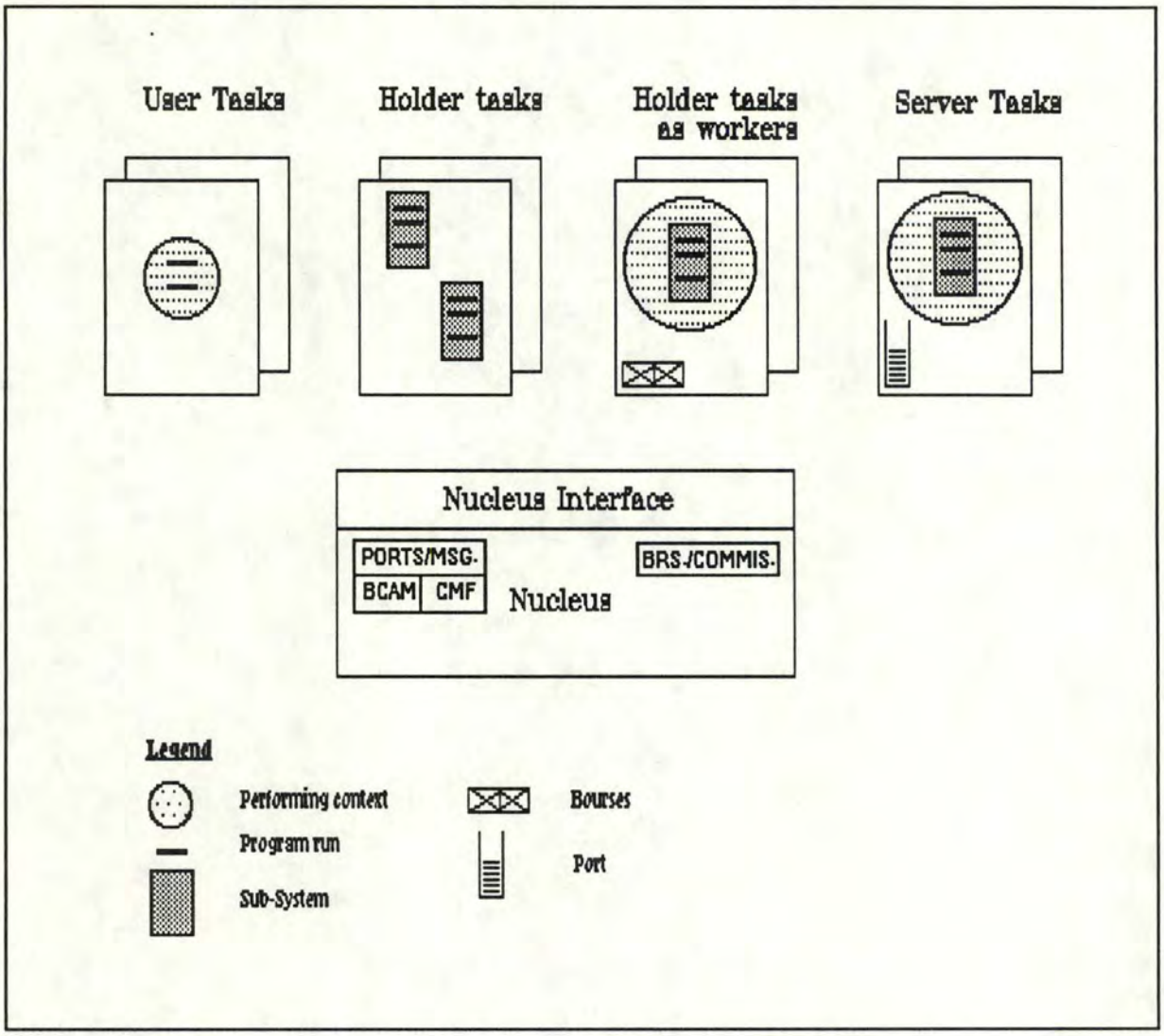

Figure 4 *The Clients/Sub-Systems Architecture*

1- Definition of the architecture components

1.1- The user task

It is a task allocated to the user program. The user program may be dynamically split into several threads. Actually the threads which represent the services provided by the local sub-systems may be executed in the context of the user task by using a local call mechanisms such as the direct branching with the common memory pool or with the shared code.

1.2- The holder task

This task may include several local sub-systems, their threads (services) are executed in the context of the user tasks. These local sub-systems have no need of a personal context or personal resources for their executions. It is the reason they are stocked within a same task.

1.3- The holder task as worker

This task includes a single local sub-system, their threads (services) are executed in the context of their own task. This local sub-system needs a personal context or personal resources for its execution. It is never called by the direct branching or by a supervisor call but it is always loops by manipulating the bourses and by treating the commissions. This kind of sub-system is called an autonomous sub-system.

1.4- The server task

This task includes only one server and has its own port , its own resources and its own context for the execution of threads which represent the services of the server. The server is always loops by manipulating the ports and by treating the messages. The server also is an autonomous sub-system.

1.5- The Kernel

There is one kernel per node because it contains all low level functionalities that are required for performing and managing all devices and all basic resources. The generic services are for example the virtual memory management; the thread scheduling and (dis)allocation; the processor allocation; the inter-task communication; the event handling (svc; interrupt; exception);... This part of the hierarchy is very dependent on the hardware. The Ports and Messages module is the transport manager or the support for the ports and messages mechanism between the clients and the servers. The Ports and Messages module is based on two other tools : Basic Communication Access Method (BCAM) is a tool of communication between the nodes and Cross Memory Facility (CMF) is a tool for sharing a certain address space between tasks of the same node. The Bourses and Commisions module is the support for the bourses and commissions mechanism between clients and local sub-systems which are autonomous.

1.6- The memory space

The memory is often split into 2 main parts : the user space and the system space. Often there also is a subdivision into classes, each of them has its own meaning. The system space is a memory space reserved to the kernel services or to the services of sub-systems. That does not exclude these services may be performed in the context of the user task. The user space is available for the user programs or for the services of sub-systems.

2- Local sub-systems or servers ?

The choice between a local sub-system and a server depends on the synchronization aspects of the execution of their services. The services of a non autonomous local sub-system may be rivals within the same context of the client task (multithreading) and different clients may use the same service of the same local sub-system at the same time. If the client has to wait synchronously for the terminaison of service of sub-system then it is better to perform this service in the client context because it is more efficient as regards the multitasking aspects (execution in the same micro time slice with a replication of the service for each client). If the client doesn't have to wait synchronously for the terminaison of the service, then the service may be performed in parrallel with the execution of the client task. Thus, the service is included within a autonoumous sub-system. It is performed in the sub-system context because it is more efficient as regards the multitasking aspects (parallelism of tasks which allows the client to go on with its activity). The service may be shared by several clients. It may also be replicated several times but never in the context of the clients. It is better to choose a server instead of an autonomous local sub-system because the server takes advantage of the parallelism of nodes. The autonomous local sub-system takes advantage of the localization knowledge of the client (on the same node) but the failing benefit of the nodes parallelism may not be recovered. In general a service may be asynchronous if it needs to share resources (data, printer, disk, ...) with other services of the same sub-system. In that case it has to be performed in the context of the sub-system which is obviously autonomous.

Examples of local sub-systems

- Editor (non autonomous local sub-system) : The services provided by an editor are used in an interactive mode with the user and they may not be performed in parallel. They have no need of a sub-system context.

- Creator : The services of the (un)loading of the local sub-systems may be performed in parallel with the client but they have to know the localization of the caller. The management of the local sub-systems requires its own context to allow the sharing of the local sub-systems.

Examples of servers

- Spool server : The printing of files may be performed in parallel. The management of the printer requires its own context in order to share the printer with a lot of clients.
- Archive server : The archiving of files may be performed in parallel. The management of the archiving requires its own context in order to share the tape disk.

Chapter 4 : Sub-System topologies

The aim of this chapter is to define the logical and physical configurations based on the functional dependencies between the sub-systems.

1- The logical configuration

A sub-system may require the presence of other sub-systems but that does not inevitably involve their calling [1]. It is in fact the functional "use" relationship that constructs the hiearchy also called the logical configuration of sub-system or the logical topology. The nucleus may be considered as the module of the first abstract level. Here each sub-system is represented by a point and their relationships by arrows, the constructed graph represents the logical configuration.

1.1- The loop extension

In this graph there cannot be any circuits because the logical configuration is defined as a hiearchy. This avoids the recurrence of calls. It would have been possible to tolerate it but it would bring about a lot of new problems.

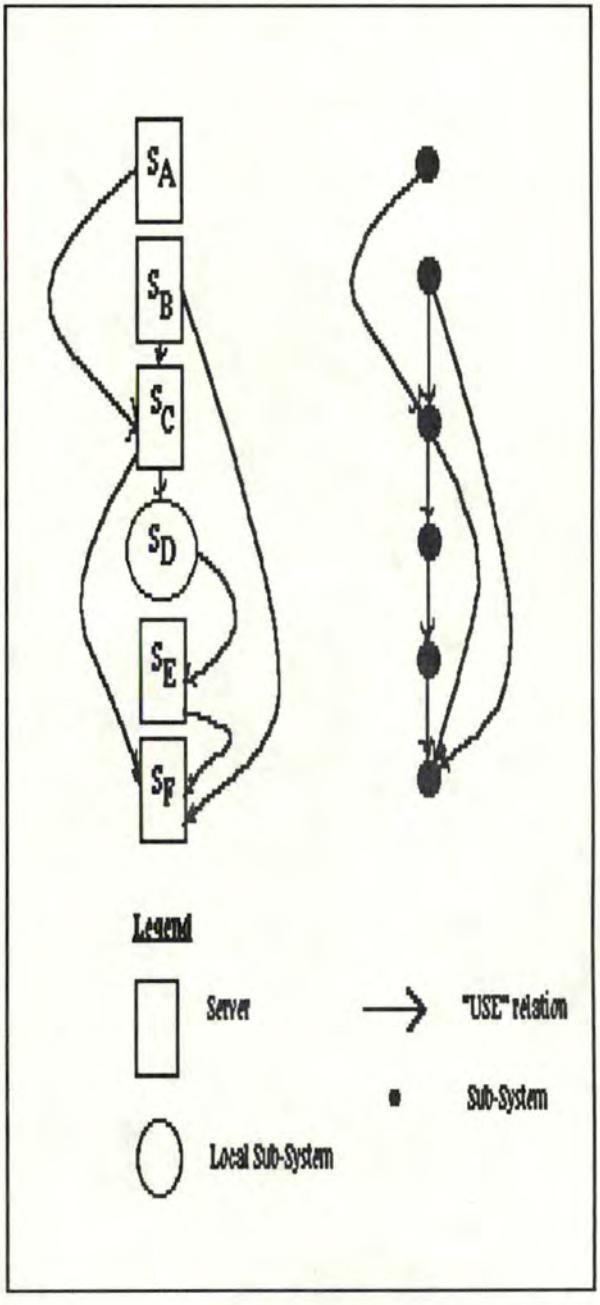

Figure 5 *The logical topology*

[1] The conditional "if" structure according to the received arguments may influence it.

For example, if a service i of a server j always calls asynchonously a service k of a server l which always calls asynchronously a service of the caller and those services are replicated within a task then a deadlock would appear after the space occupied by the replicated services has filled the task space up wherever their execution contexts are.

1.2- Examples of the "use" relationship

- The servers require the presence of the binder and the loader because they manage the (dis)connections and the (un)loading between the clients and the servers.
- A sub-system i insures the portability to the sub-system j for the different operating system versions. Consequently the sub-system i will have to be present each time the sub-system j is in that configuration.
- There is no practical example for the absence requirement ...

2- The physical configuration

Several copies or instances of the same sub-system can exist on different nodes but for efficacy and complexity reasons there always is at most one instance of the same sub-system on the same node. But the principle of the single image machine implies that the client has to see only one entity regardless of the instance situation and its possible replication. If the instances of the same server need to use a local sub-system, then it has to be located on the same nodes as those instances. The physical configuration or topology is the same as the logical configuration but it takes into account the instances on each node. Thus, for a given logical configuration there may be a lot of different physical configurations.

2.1- The coherence of the physical configuration

If a sub-system i of the physical configuration requires the presence of the server j, then the server j must also be in the physical configuration.
If a sub-system i of the physical configuration requires the presence of the local sub-system j, then the local sub-system j also is in the physical configuration and they are on the same nodes.

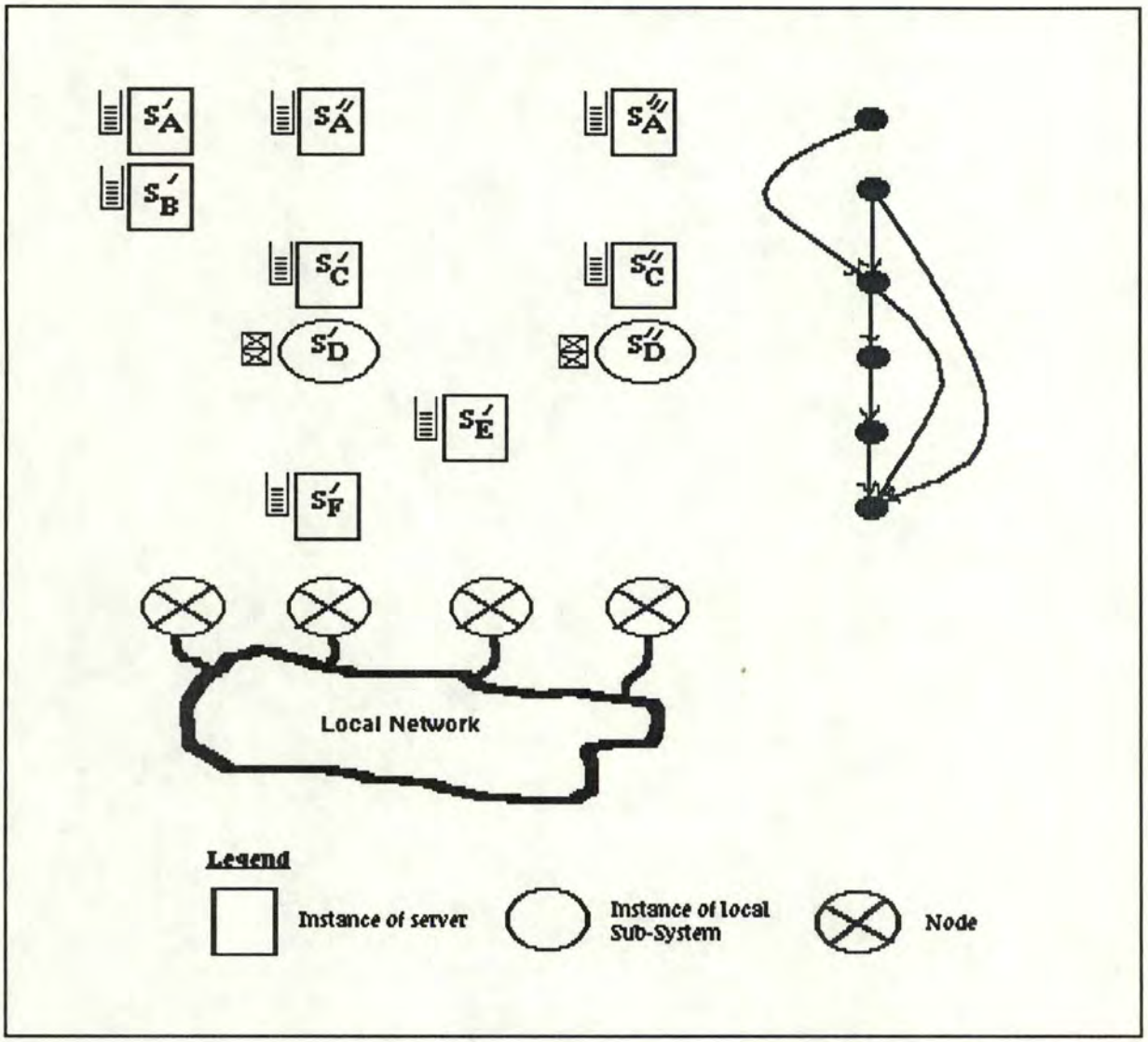

Figure 6 *The physical configuration*

3- The startup configuration

After each startup phase [1] there always is the same logical configuration and so it is statically predefined. The corresponding physical configuration may evolve with time, using the (un)loading of sub-systems. The loader has to maintain the coherence of the physical configuration. Thus, the functional dependence has a direct impact on the (un)loading validation.

[1] The installation of this initial configuration involves a "temporal" serialization of loading and some different states of the physical configuration (e.g. system ready; system not ready; before DSSM loaded; after DSSM loaded; ...) [SIM2]. But it is not the objective of this thesis to bring that out.

4- The states of instances

Not present

The instance is not present in the physical configuration (not loaded).

Not ready or present

The instance is present in the physical configuration. There are no connections established with it.

Ready

The instance is present in the physical configuration. There are connections established with it. Perhaps a service of the instance is being performed.

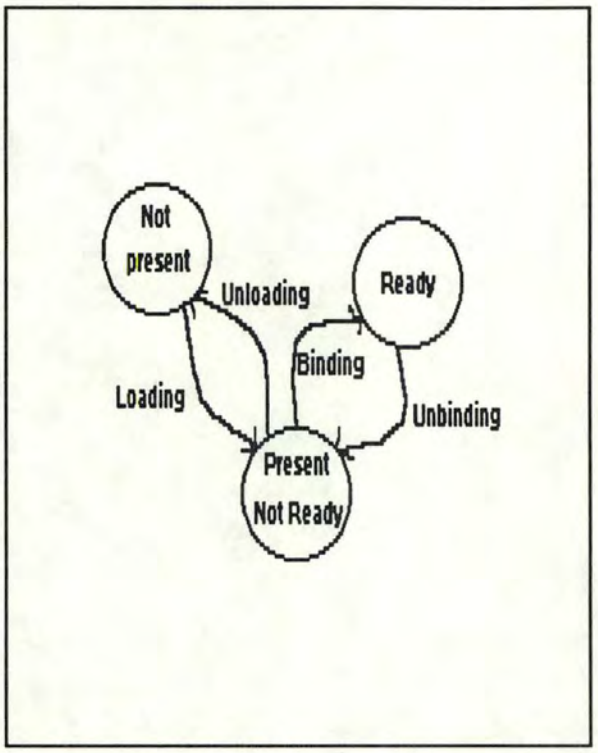

Figure 7 *The instance state*

5- The states of sub-systems

Not present

The sub-system is not present if all of its instances are in a "not present state".

Present

The sub-system is present if at least one of its instances is "in present state" or in "ready state".

Chapter 5 : Client/Server model

The aim of this chapter is to explain the Client/Server model because it is the conceptual or theorical background of communication in a Distributed Computer Environment. The scope of the model itself is from a functional point of view to explain the communication between two entities which are being performed in parallel, the first entity having need for a service which is performed by the second entity wherever they are.

1- The Client

It is the entity that asks another entity, which is called the server, to perform a service. The client carries it out by sending the arguments of the service in a message to the port of the server and by receiving the results of the service that are in another message. The sent message has to include the name of the service for the purpose of identifying the required service. The port of the server is a server queue where the messages are accumulated. The client has to make a connection corresponding to the service name to know the port of the server and to increase the number of connections of the server. To decrease the number of connections of the server the client has to make a disconnection.

2- The Server

It is the entity which performs the services required by the clients. The server carries it out by treating the messages which are accumulated in its port and by replying the results in a message to the corresponding client. The server has only one port. A server may be the client of another server. The server includes a set of different kinds of service. The functional designing of a server is a methodological problem that requires the know-how of a software engineer.

The functional equivalence between server and service

The server S is a set of services f1, f2, f3 and fc, the function fc is the dispatcher for the coordination of f1, f2, f3. The servers S1, S2, S3 respectively have the services f1, f2, f3, each server has its own dispatcher. We shall call this set of servers (S1, S2 and S3) : O.

Now we have : O <==> S from a functional point of view

2.1- The dispatcher of the server

Each server has a special function called the dispatcher which dispatches the services according to the service name. In fact, it is the dispatcher which receives the message, unmarschalles (or "splits") it into a service name and arguments, calls the service with the arguments, marschalles (or "encapsulates") another message from the results of the service and replies this message.

2.2- The functional server configuration

A server may be the client of another server, so there are relationships between servers. Those relationships define a servers configuration which may be representated by points and arrows, the points being the servers and the arrows the relationships.

3- The message

The structure of the sent message includes :

- the service name
- the arguments of the service

The structure of the replied message includes :

- the results of the service

4- The managers

4.1- The service name manager

It is an entity which is responsible for the connection between a client and a server. It manages the relationships between the port and the service names so as to permit the dynamic reconfiguration of instances. In addition, it also updates the number of connections of the server. It provides the clients with two interfaces : the connection (Connect) and the disconnection (Disconnect). Sometimes the words (un)binding are used instead of (dis)connection.

4.2- The presentation manager

It is an entity which is responsible for the representation of the arguments or of the results within a message. The parameters are given by value and never by reference. It provides two interfaces to the clients or to the servers : the message marschalling (Marschalling) and the message unmarschalling (Unmarschalling).

4.3- The transport manager

It is an entity which is responsible for the transport of the message between a client and a server. The transport manager carries it out by identifying the client and by recording the all of the necessary information such as the identification of the client, the port of the server and the identifier of the transport connection. It is possible that for a given transport connection the connection identifier given to the client and the connection identifier given to the server are not the same. Nevertheless, the transport manager has to record this data couple. In this case we have two identifiers for the transport connection. It provides two interfaces to the clients : the message sending (Sendmsg) and the message waiting (Waitmsg). And it provides two interfaces to the servers : the message receiving (Receivemsg) and the message replying (Replymsg).

5- The interfaces

5.1- For the client

Connect(servicename;port)

Input parameters :

servicename : the name of the service

Output parameters :

port : the port of the server corresponding to the service

Marshalling(servicename;serviceargument;msg)

Input parameters :

servicename : the name of the service

serviceargument : the arguments for the service

Output parameters :

msg : the message

Sendmsg(port;msg;connectid)

Input parameters :

port : the port of the server corresponding to the service

msg : the message

Output parameters :

connectid : the identifier of the transport connection

Waitmsg(connectid;msg)

Input parameters :

connectid : the identifier of the transport connection

Output parameters :

msg : the message

Unmarshalling(msg;serviceresults)

Input parameters :

msg : the message

Output parameters :

serviceresults : the results from the service

Disconnect(port)

Input parameters :

port : the port of the server corresponding to the service

5.2- For the server

Receivemsg(connectid;msg)

Input parameters :

connectid : the identifier of the transport connection

Output parameters :

msg : the message

Marshalling(serviceresults;msg)

Input parameters :

serviceresults : the results from the service

Output parameters :

msg : the message

Unmarshalling(msg;servicename;serviceargument s)

Input parameters :

msg : the message

Output parameters :

servicename : the name of the service

serviceargument s : the arguments for the service

Replymsg(connectid;msg)

Input parameters:

connectid : the identifier of the transport connection

msg : the message

Remark

Obviously for each interface there is a return code which informs about the validity of the output parameters.

6- The asynchronous aspects between client and server

Clients and servers are performed in parallel and when a client needs a service of a server then the client doesn't inevitably have to wait for synchronous response of the service. The client receives the response when it wants.

7- The parallelism of server services

The server may treat two or more messages at the same time. In addition, the treatment of a message can be finished before the treatment of another message which has been received earlier from the port even if those services are of the same kind.

8- The inter-client communication as a particular case of service

If two clients want to communicate then they have to use the special communication services which are included in a special communication server. In particular when two clients do not have not their own port. The aim of this kind of communication is not the performing of services but the exchange of a lot of data. For example, we could define a connectionless protocol by using two communication services such as the data sending (Senddata) and the data receiving (Receivedata). In this example the sender and the receiver are free to choose

their own symbolic name and they are free to communicate or not with one another but there are no special services which are able to indicate if the data has been received or not by the receiver.

A simple example of communication interfaces

Senddata(Symbolicnamesender;Symbolicnamereceiver;data)

Input parameters :

Symbolicnamesender : the symbolic name of the client (the sender)

Symbolicnamereceiver : the symbolic name of the client (the receiver)

data : the information given from the sender

Receivedata(Symbolicnamesender;Symbolicnamereceiver;data)

Input parameters :

Symbolicnamereceiver : the symbolic name of the client (the receiver)

Output parameters :

Symbolicnameemitter : the symbolic name of the client (the sender)

data : the information given from the sender

Remark

Obviously for each interface there is a return code which informs about the validity of the output parameters.

9- The stochastic paradigm

The stochastic model may be associated with the configuration of servers in order to evaluate or to approximate a great number of statistical values such as the response time for each service, the average number of messages in each port, the troughput of each server, etc ... But each arrow has to be weighted against a probability of call and the service rate of the services or of the servers has to be defined. Thus, the server configuration permits to describe a stochastic model.

Chapter 6 : Remote Procedure Call (RPC)

The aim of this chapter is to explain the Remote Procedure Call which is well know by designers of the distributed operating systems. The Remote Procedure Call is a mechanism which allows the programs to use global services. The programs have access to the global services by using the remote procedure call instead of the port and message mechanism. Please note that the meaning of the procedure in this chapter is very different from the meaning of the procedure when it is a kind of program run.

1- Procedures and programs

Before explaning the remote procedure call we would like to recall the concepts of procedures and programs. Performing the procedures and the programs requires different phases of construction.

1.1- The edition phase

The edition phase is the writing of sources. The sources are the texts which specify procedures and programs. A procedure is composed of a body and a head. The body of a procedure is an algorithm, which is in fact a set of instructions. The head of a procedure is composed of the definition of the local variables and of its interface. The interface of a procedure is composed of the name of the procedure and the definition of the input and output parameters. A program is made up of a body and a head. The body of a program is an algorithm or a set of instructions such as the procedure call, the loop instruction, the conditional instruction, etc ... The head of a program is made up of the input and output parameters, the definition of the global variables, the name of the program, the internal procedures, and the interface of the external procedures.

1.2- The compilation phase

The compilation phasis is the compilation of the sources for getting the machine codes. The procedure code of internal procedure is included in the program code. The procedure code of external procedure is not included in the program code but its interface has to be specified in the program. The library is a set of procedures codes which are specified in the user programs as being external.

1.3- The statical linking phase

The statical linking phase is the linking of the program code to the external procedures to build an executable program. All of the external procedures are merged with the program code.

1.4- The loading phase

The loading phase is the loading of an executable program which becomes program runs in a task. A procedure may become a program run. In the performing phase several threads may represent a same procedure within the same task.

2- The relationship between the services and the procedures

The difference between a procedure and a service resides in their different call mechanisms, also the services are never statically linked to the program code of the client. The management of sub-systems carries out this dynamical linking (or binding) of the services and the loading of sub-systems. The purpose of this thesis is not to explain in detail the use of the local services because it has already been analysed for the traditional operating systems.

Services performed within the client context

Those services are locals and they are included in local sub-systems. They are never performed in parrallel with the client task. A client may access them with a direct branching or by using the replacing macros.

Services performed within the sub-system context

Those services may be locals or globals. They are still performed in parrallel with the client task. Thus the sub-systems which contain those services are designed as programs. For the local services they are included within local sub-systems. A client may access them with the bourses and commissions mechanism or by using the replacing macros. For the global services they are included within a server. A client may access them with the ports and messages mechanism or by using the replacing macros. Indeed, the procedures which correspond to the global services are called the remote procedures.

3- The Remote Procedure Call

The client program calls the client-stub procedure instead of directly using the mechanism of ports and messages to call the remote procedure. Likewise, the server program calls the server-stub procedure instead of calling directly the remote procedure because the remote procedure does not use the ports and messages mechanism. The body of the server program is called the dispatcher [COUL].

3.1- The client-stub procedure

The client-stub procedure is an external procedure for the client program. Thus, for each service there is a corresponding remote procedure and a corresponding client-stub procedure. The library which contains the client-stub procedure is called user package. The client-stub procedure has to establish the communication between the client and the server for the client program.

Example of an algorithm of the client-stub procedure

The name of the service is Pascal with arg1 and arg2 as input parameters and arg3 as output parameter. The name of the corresponding client-stub procedure is Nestor1.

```
BOOL  Nestor1(arg1,arg2,arg3)
struct ARG1TYPE * arg1;
struct ARG2TYPE * arg2;
struct ARG3TYPE * arg3;
{
struct PORTTYPE port;
struct MSG msg;
struct CONNECTID connectid;

    if (Connect("Pascal",port) == 0)
    /* the service is accessible */
    {
    Marshalling("Pascal",* arg1,* arg2,msg);
    Sendmsg(port,msg,connectid);
    Waitmsg(connectid,msg);
    Unmarshalling(msg,* arg3);
```

```
        Disconnect(port);
        return(0);
        }
        else
        {
        return(-1);
        }
}
```

3.2- The dispatcher

The dispatcher is the body of the server program. It has to manage the port (the receipt of the message by using a strategy like FIFO, the deletion and creation of the port itself, etc ...). By managing the procedures parallelism, it can dispatch the message to the corresponding server-stub procedure according to the service name.

Example of an algorithm of the dispatcher

The name of the server is Servera. One of the service name is Pascal. The name of the corresponding server-stub procedure is Nestor2.

```
main Servera
{
struct PORTTYPE port;
struct MSG msg,msg2;
struct CONNECTID connectid;
char * endofserver;
extern void nestor2(struct CONNECTID;struct MSG);
.../... /* other definitions of external server-stub procedures */
        .../... /* creation of port */
        endofserver = "no";
        while (endofserver == "no") do
        {
          Receivemsg(connectid,msg);
          Unmarshalling(msg,servicename,msg2);
```

```
        switch (servicename)
        {
        case "Pascal" : nestor2(connectid,msg2) &; /* in parallel */
        case "end" : .../... /* terminaison of the server */
                    endofserver = "yes";

        .../... /* other server-stub procedures calls */
        }
        }
        .../... /* deletion of port */
}
```

3.3- The server-stub procedure

The server-stub procedure is an external procedure for the server program. Thus, for each service there is a corresponding remote procedure, a corresponding server-stub procedure and a corresponding client-stub procedure. The server-stub procedure has to unmarschall the message into the input parameters, to call the remote procedure, to marschall the message from the output parameters and to reply the message. The library which contains the server-stub procedures is called the server package. The dispatcher and the server-stub procedure have to establish the communication between the client and the server for the server program.

Example of an algorithm of the server-stub procedure

The name of the server-stub procedure is Nestor2. The name of the corresponding external remote procedure is Nestor with arg1 and arg2 as input parameters and arg3 as output parameter.

```
void   Nestor2(connectid,msg)
struct MSG msg;
struct CONNECTID connectid;
struct ARG1TYPE * arg1;
struct ARG2TYPE * arg2;
struct ARG3TYPE * arg3;
extern void Nestor(struct ARG1TYPE *;struct ARG2TYPE *;struct ARG3TYPE *);

        Unmarschalling(msg,* arg1,* arg2);
        Nestor(arg1,arg2,arg3);
```

```
    Marschalling(* arg3,msg);
    Reply(connectid,msg);
}
```

3.4- The Interface Specification Language and compiler

The client-stub procedures, the dispatcher and the server-stub procedures are developped by a specialist of sub-system management. In this way the service names and the mechanism of ports and messages are hidden from the application programmers. By specifying only the name of the service name, the parameter kindd and the name of the client-stub procedure to be generated the client-stub procedures may be easily generated automatically. The dispatcher and the server-stub procedures may be automatically generated by specifying the name of the server, the service names, the parameter types, the name of the remote procedure and the name of the server-stub procedure to be generated. Those specifications may be written in an interface specification language and compiled by an interface specification compiler. The compiler would check the equivalence of the parameter types between the remote procedure, the client-stub procedure and the server-stub procedure. The statical linking of the dispatcher, the server-stub procedures and the remote procedures brings about the executable server program. The statical linking of the client program code and the client-stub procedures brings about the executable client program.

3.5- RPC semantics

Some problems may occur when using the Remote Procedure Call. For example, the request message may get lost, the reply message may get lost or the server may crash.

At-least-once call semantics

The client can be sure the procedure has been performed at-least-once when it receives the reply. The client has to wait for the reply and if after a certain timeout there is no reply then it sends the message once again. It repeats it until it receives a reply. Therefore, it is possible the procedure is performed more than once.

At-most-once call semantics

The client may be sure the procedure has been performed at-most-once when it receives the reply. The client has to wait for the reply and if after a certain timeout there is no reply then it

sends the message once again. It does so until it receives a reply and it is not possible for the procedure to be performed more than once.

The maybe call semantics

There is no reply for the client request. The client does not know whether the procedure has been performed ...

3.6- Kinds of RPC protocol

There are several kinds of Remote Procedure Call protocol according to the RPC semantics.

The "R" protocol : the request protocol

The client does not wait for the reply to its request. Here we have the "maybe" call semantics.

The "RR" protocol : the request/reply protocol

The client waits for the reply to its request. Here we may get either the "at least once call semantics" or the "at most once call semantics". For securing the "at most once call semantics" the server performs the service only once. But it has to keep the messages.

The "RRA" protocol : The request/reply acknowledge-reply protocol

The client waits for the reply to its request. When it receives the reply it sends a reply acknowledgement to the server. Here we have the "at most once call semantics". To secure the "at most once call semantics" the server performs the service only once. Here the server doesn't have to keep the messages but it has to wait for a reply acknowledgement.

Chapter 7 : Concurrency of services

The concurrence of services comes from the parallelism of the performing units. But several services may not be performed at the same time. The aim of this chapter is to provide some management mechanisms for the concurrency of services.

1- Multithreading

The threads are performed in parallel within the same task, they share all resources of the task. Thus, several services may be performed in parallel in a client or an autonomous sub-system task (i.e. the loading and the unloading of the same instance may not be performed at the same time, two loadings of the same sub-system on the same node, ...). A common variable permits the mutual exclusion of rival services (i.e. the state of the instance prevents the unloading and loading of the same instance at the same time). A monitor also permits the mutual exclusion of rival services (i.e. the dispatcher may serialize the requests of the same service : if two loadings of the same sub-system on the same node occur at the same time at least one loading fails).

2- Multitasking

The tasks are performed in parallel on the same node or on different nodes. Thus, several services may be performed in parallel in several clients or several autonomous sub-system tasks. There is a concurrency of services because there is a replication of instances or when several sub-systems share the same resource (i.e. the loading and the unloading of the same instance may not be performed at the same time by different instances of the loader, the connection and the unloading of the same instance may not happen at the same time, two loadings of the same sub-system on the same node by different instances of the loader, ...). The mutual exclusion of those services is carried out by using the inter-task communication or by using a shared file.

2.1- Mechanisms for managing the concurrency

2.1.1- The inter-task communication

When there is a change of information for one instance then it has to communicate this change to the others. This may be done by a kernel service or by the port and message mechanism. The main problem arises when there are two attemps of changing the same information in two instances at the same time. To solve this problem a master/slave protocol with the polling adressing may be implemented. For example, the master may be the first instance loaded. The disadvantage of this is the implementation of this protocol and its negative effect on the efficiency.

2.1.2- The shared and distributed file

Another solution is the use of the shared and distributed file containing the common information with the exclusive accesses for writing or updating the data. It is an easier solution because it does not imply a particular implementation, also, the absence of the inter-task communication with a protocol is a good way for increasing the efficiency. The distributed file manager also insures the transparancy of access to this kind of file.

2.1.2.1- The catalog file

The catalog file is a set of sub-system declarations. The declaration of a sub-system is made up of all of the permanent information about a sub-system. The permanent information of a sub-system is the information which stays unchanged during the changes of the sub-system state. The management (creation or modification) of the sub-system declarations is insured by a sub-system called the generator.

2.1.2.2- The configuration file

For example, to avoid the operations (the binding and the unloading) made by the loader and the binder on the same instance occuring at the same time. A common information helps to solve this problem. The state of this instance has to be included in a shared and distributed file. This file is called the configuration file. The configuration file is a set of sub-system information which may be modified when the sub-systems are in present state.

PART 3

Dynamic Sub-Systems Management in a Closely Coupled Architecture

Chapter 1 : Binding management

In the traditional operating system there already is a module called the linker which manages the (dis)connections between the clients and the local sub-systems [SIM2]. The distributed operating system needs another similar module or an extension of the linker. This extension or this additional module is called the binder. The binder is a sub-system which establishes the (dis)connections between the clients and the server instances. This chapter only examines the (dis)connection between the clients and the servers because the management of the local Sub-Systems already exists in the traditional operating systems.

1- The (dis)connection or (un)binding phases

1.1- The oriented service connection

To avoid the client having to know about the relationship between the server and their services, the binder furnishes the concept of the connection to a service. A client wants to make a connection with a service (i) of a server. It calls the service of connection provided by the binder to get the port of an instance of the server which includes the service (i). The service of connection of the binder has to treat the request. This service of connection has to know the physical configuration and the relationship between the servers and their services. If there isn't an instance of this server in the physical configuration (the "Not Present state" for each instance of this server) then the binder calls the loading service furnished by the loader to obtain an instance in the physical topology and to receive its port. The server has to be specified with the auto-load attribute = «YES». The connection request to a "not present" server may only be made by a client which is a user task (not another sub-system task) because the physical configuration always has to be coherent. If there are several instances of this server in the physical configuration (the "Present" or "Ready" state for several instances) then the binder has to select only one instance. After the selection it has to reply the port of the server instance which includes the service (i). Also, the binder increases by one the connection number of the server instance. Please note, that zero as a connection number value indicates whether the instance is in a "not ready" or "not present" state. The binder has to change the state of the instance if it was in "not ready" state previously. In short, the service of connection has three phases : the validation, the auto-loading and the selection. The client calls the service (i) then it makes a disconnection with the corresponding instance. Thus, there is always one (dis)connection for each service calls. The Client/Server model and the RPC have adopted the oriented service connection.

1.2- The oriented server connection

A client wants to make a connection with a server. It calls the service of connection provided by the binder to get the port of an instance of this server. The service of connection of the binder has to treat the request. The scheme is nearly the same than that of the oriented service connection but the binder doesn't have to establish the relationship with the server and their services. When the client receives the corresponding port it is free to call several services of this instance during the same connection, but it has to know the relationship between the server and its services ... It is also free to make a disconnection when it wants to.

1.3- The disconnection between a client and an instance

A client wants to make a disconnection with an instance. It calls the service of disconnection provided by the binder. The service of disconnection of the binder has to treat the request. This service of disconnection has to know the physical configuration. Also the binder decreases by one the connection number of the server instance. Please note that zero as a connection number value indicates whether the instance is in "not ready" or "not present" state. The binder has to modify the state of the instance when the value zero has been reached.

1.4- The coherency of the service names of the server

S(i) as server.
s(ij) as service.
s(ij) belongs to S(i) for each j.
S(i) not = S(k) if i not = k.

If the service name of s(ij) is not = the service name of s(kl) when i not = k or j not = l, then the service names of all servers are coherent.

This coherency is checked by the generator which installs the declarations (specifications) of the server in the catalog file. In this case, the catalog is a set of server declarations and the server declaration is a set of values corresponding to the attributes of a server.

1.5- The explicit connection

The explicit connection would be a connection between the client and the server of a given node. But that does not correspond with the principle of the single image machine because the explicit connection is not a transparant mechanism. Therefore, the clients cannot be given this option.

1.6- The problem of the connection to the binder

The client cannot connect to the service of (dis)connection of the binder to receive its address or its corresponding port because, to obtain this, the client has to know the address or port of this service. Thus, the client directly calls the service of (dis)connection by using the SuperVisor Call (SVC Mode of Call) because the nucleus knows the reference of the binder and the SuperVisor routine may call the service of (dis)connection of the binder. From a dynamical point of view the supervisor routine is executed in the context of the client task. The call of the supervisor may be hidden by using the commands or the macros ($connect and $disconnect).

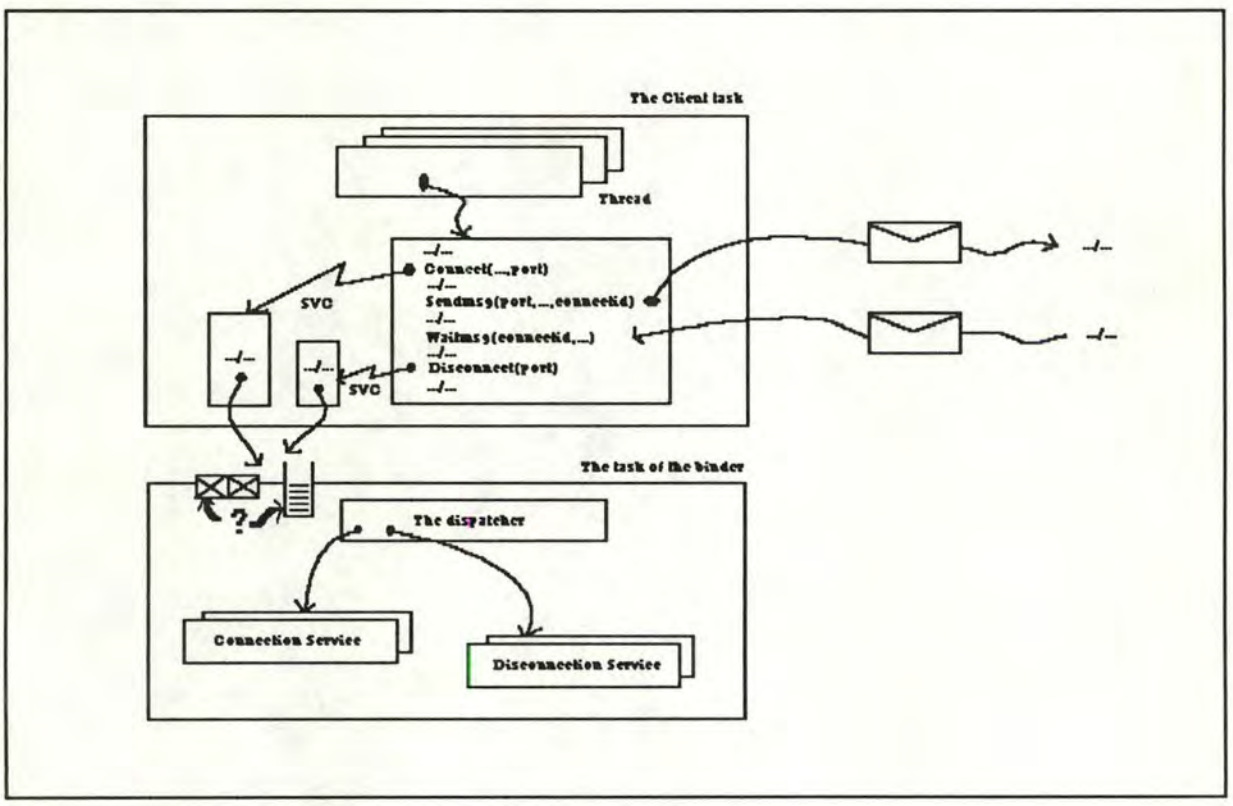

Figure 8 *The dynamical components*

2- The dynamical components

The call of the (dis)connection services to the binder is performed in the context of the client task. When the client task is composed of client-stubs then it is the client-stub which calls the (dis)connection services. The services of the (dis)connection are performed in the context of the binder task and they may be threads.

3- The interfaces of the binder

There are three basic services provided by the binder : the binding to a server, the binding to a service and the unbinding to an instance.

ConnectServer(server-name,server-version,port,ret-code)

Input parameters :

server-name : the server name.

server-version : the server version.

Output parameters :

port : the port of an instance of the server if the return code equals ok.

ret-code : the return code of the operation (ok or fail).

ConnectService(service-name,port,ret-code) <==> Connect(servicename,port,ret-code)

Input parameters :

service-name : the service name.

Output parameters :

port : the port of the instance of the server wich includes the service if the return code equals ok.

ret-code : the return code of the operation (ok or fail).

DisconnectInstance(port,ret-code) <==> Disconnect(port,ret-code)

Input parameters :

port : the port of the instance to unbind.

Output parameters :

ret-code : the return code of the operation (ok or fail).

4- The choice of the system-server instance

When the binder replies a port of a server to the client it is in fact a port of an instance of this server. In the case where there are several instances of the same server the binder may choose the connection at random but the response time of the service may suffer from this approach. The problem for the connection is to choose the instance that offers the best response time for the client. There are a lot of different criteria such as the number of task per node, the CPU occupied rate per node, the response time of the network, the physical configuration, the probability of a connection between sub-systems, the probability to call a connected instance, the existing connections between instances, the number of connections per instance, the weight of the instance port, the localization and the replication of the binder, the service rate of the instances, etc ...

4.1- The number of tasks and the CPU occupied rate

A principle of the Distributed System is that each node has for efficiency reasons the same number of tasks and the same CPU occupied rate. Otherwise there are in terms of performance «bad» nodes which would be suppressed. These balanced nodes have to be insured by the task loader and the loader of sub-systems. We consider this principle of balanced nodes as an axiom during the (un)binding phases. The periods between two successive time slices of any instance have to be the same from a task scheduling point of view.

4.2- The considerations about the network and the physical configuration

The time requested by the network [1] activities may be included in the response time of the service, thus the binder has to minimize the response time by avoiding the connections that imply more exchanges on the network. Several sequences of services of other sub-system instances may be used for carrying out a service and each of them may send a certain number of messages through the network. Therefore, the physical configuration has to be taken into account.

---

[1] The network considered here has a fully interconnected architecture [DASS] that implies there is no time difference when a node i sends a message to a node j rather than the node k. In addition the messages have the same length.

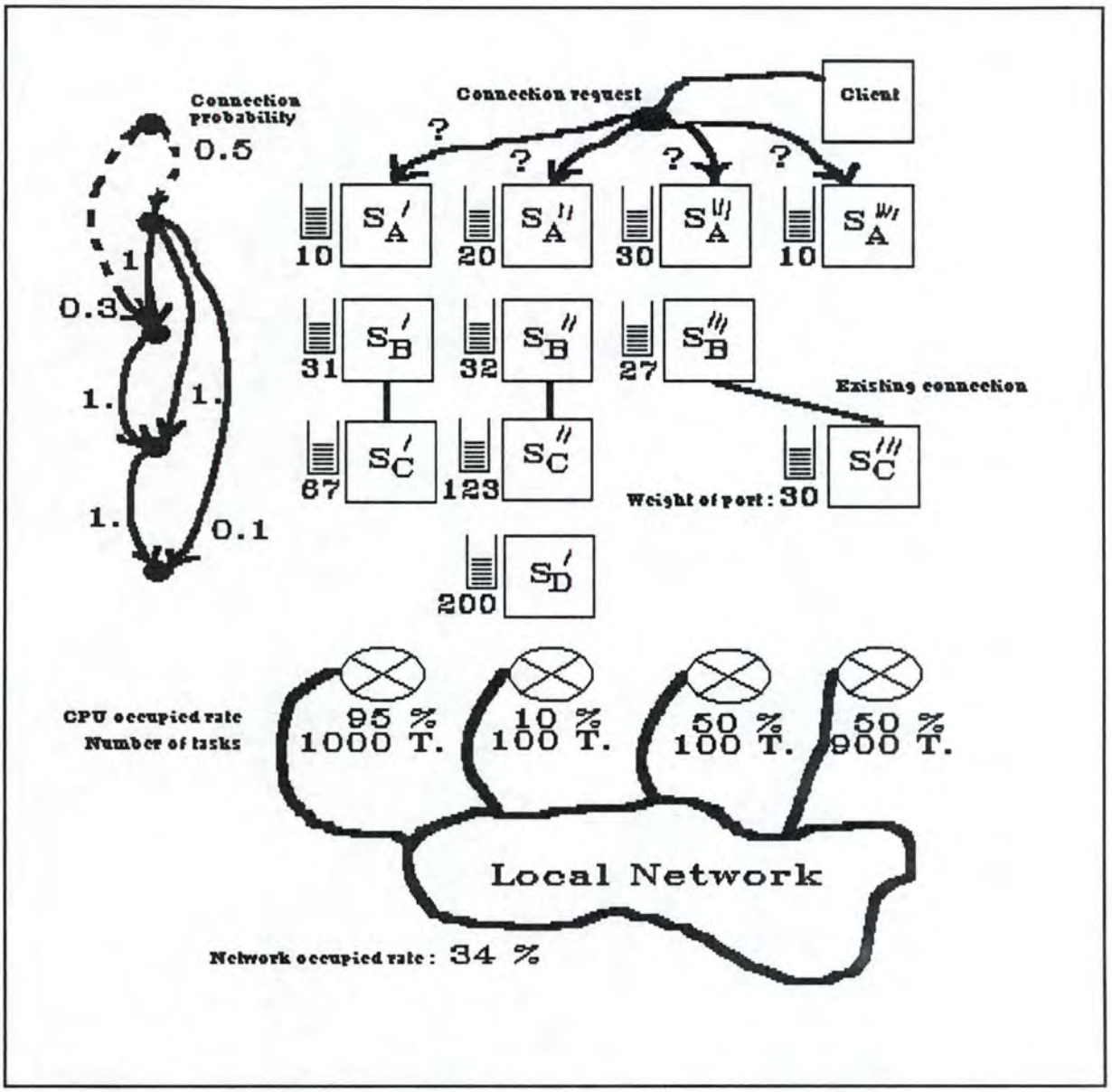

Figure 9 *The choice of the best instance during a connection ?*

When a "use" relationship between two sub-systems is specified it doesn't necessary mean that they will establish a connection. This is the concept of connection probability [1]. There also is a statistical distribution of the number of called services per connection of an instance. Those probabilities have to be weighted according to the level of the sub-system in the logical topology.

[1] Indeed, there is a probability of using a service of a given sub-system rather than another. The connection probability is calculated from those service probabilities.

The localization and the redundancy of the binder instance have an influence on the response time of the service because the time requested by the connection may not be negligible. The problem of the choice of an instance in a given configuration could be mathematically defined and solved but there is some dynamical information the binder doesn't know (i.e. the number of messages of each connection, the utilization of the existing connections by a server rather than the creation of new connections). Moreover the time requested by the calculation has to be taken into consideration ...

4.3- The number of connections per instance

Another solution to the problem is more stochastic, as we will see. The dispatching of connections permits the best performance of the response time for each service.

4.3.1- Hypothesis

The hypothesis is that all utilizations of a server by clients have a behaviour of a Poisson distribution with the same arrival rate (LAMBDA). In other words, each connection (oriented server connection) to instances of a given server implies that its average number of requested services (LAMBDA) is the same. Other kinds of statistical distributions could be imagined ... The evaluation of the Poisson distribution rate may be specified in the declaration of the server. The designer has to evaluate this Poisson rate per connection. To calculate the port size it has to evaluate the maximum number of connections per instance. It may be useful to modify this rate to use some statistical measures given by the kernel about the port of the instances. The arrival rate of requested services per instance equals the sum of the Poisson rate of each connection per instance or equals the Poisson rate of one connection multiplied by the number of connections of the instance.

This means :

$LAMBDA(i) = LAMBDA * N(i)$
such N(i) is the number of connections of the instance i.

The service rate (MU(i)) of the instance i must be greater than the rate of message arrivals per instance so as to not saturate the instance and must have an exponential distribution.

This means :

$MU > LAMBDA(i)$.

4.3.2- The strategy : the dispatching of connections

For example if you have two instances of the same server and the first has four more connections than the other then the next connection for a client which is on the same node as the first instance, has perhaps to be made with the second instance because its response time doesn't include the additional time brought about by the four additional connections.

The additional time of an instance i is the number of messages in the port divided by the service rate of the instance i (MU(i)). It is necessary to evaluate this additional time because the binder does not know the number of messages in the port.

The stochastic formulas (M/M/1) [SAUE] for the instance (i) are following ...

$$\textit{The utilization rate of the instance } i = TETA(i) = \frac{LAMBDA(i)}{MU(i)}$$

$$\textit{The average number of message in the port} = P(i) = \frac{TETA(i) * TETA(i)}{1 - TETA(i)}$$

$$\textit{The average waiting time in the port} = T(i) = \frac{P(i)}{LAMBDA(i)}.$$

$$\textit{The response time of the instance } i = R(i) = T(i) + 1/MU(i) = \frac{1}{MU(i) - LAMBDA(i)}.$$

The additional time is evaluated by T(i). The additional time is nil if there is not an existing connection for an instance (and in fact we have T(i) = 0 / 0). This doesn't happen often because if there is a replication of the server then it is because this server has a great [1] demand. The parameters of the response time of an instance are the number of connections and the service rate of the instance. But the response time of the service also depends on the localization of the client which requests a connection.

If the client is not on the same node as the instances in "present" or "ready" state, then the connection is made with the instance of the server which provides the minimal response time.

---

[1] The redundancy of server increases the number of ports and thus the global capacity of the server.

The response time of the service equals the response time of the instance plus the additional time required by the local network to exchange the messages. But the binder doesn't have to calculate this additional network time because the network has a fully interconnected architecture.

This means :

Client on node(k)
Instance(k) is in "not present" state
There exists at most an instance(i) which is in present or ready state such i not = k

If R(j) = Min{R(i) : for each i such the instance(i) is in present state}
Then the connection is made between the client and the instance(k) on the node(k)

If the instance(i) in "present" or "ready" state is on the same node(i) as the client and this instance has the minimal response time R(i) compared to the response time of the other instances of the same server, then the connection has to be made with this instance and the client.

This means :

The instance(i) and the client are on the same node(i)

If R(i) = Min{R(j) : for each j such the instance(j) is in "present" or "ready" state}
Then the connection is made between the client and the instance(i)

If the instance(i) in "present" or "ready" state is on the same node(i) as the client and another instance(k) of the same server in the "present" or "ready" state has the minimal response time R(k), and, if the time requested by the message exchange through the network TN plus this minimal response time R(k) is greater than the response time of the instance(i), then the connection has to be made with this instance(i) and the client otherwise the connection has to be made with the client and the instance(k).

This means :

The instance(i) and the client are on the same node(i)
TN is the time requested by the messages exchange through the network
R(k) = Min{R(j) : for each j such the instance(j) is in "present" or "ready" state} and k not = i

If R(k) + TN => R(i)
Then the connection has to be made between the client and the instance(i)
Else the connection has to be made between the client and the instance(k)

4.3.3- Localization of the client

Two methods are available to know the localization of the client : the first is that the binder is an autonomous local sub-system which is always on the same node as the requester of the connection (the client), and the second is that the supervisor routine of the connection request mechanism establishes a protocol with the binder for indicating the position of the client. In this thesis the first solution is chosen, therefore there are instances of the binder on each node where the clients are.

4.3.4- The time requested by the calculation

The whole information for calculating the response time of an instance has to be known by the binder. The response time of an instance is calculated by the binder which is on the same node.
The calculation of the response time of an instance is always made after the modification of its number of connections. The response time of an instance(i) in "not ready" state equals 1/ MU(i). The variables are : the number of connections (N(i)), the time required by the message exchange (TN) and the service rate of the instance (MU(i)). Obviously, the designer may also change the evaluation of the Poisson rate of the arrivals per connection (LAMBDA) in the declaration of the server. The time requested for the evaluation of MU(i) and of the evaluation of TN may be quite long thus these evaluations may be made sometimes but not each time we have a connection to an instance.

4.3.5- The service rate of the instance

A sub-system may contain several services which have different service rates but thanks to the stochastic approximation [SAUE] we replace these service rates by the service rate MU(i) of the instance(i). Nevertheless, a programming methology may improve the situation. For example each service must have its own queue if there is only one thread per service, because two requests for the same service may block the dispatcher. Another solution is to create a new thread for each request.

Thus, the multithreading or the programming methodology improves the service rate of the instance; the real response time is better than the predicted response time.

The method for the evaluation of MU(i)

The binder sends a message to the server which contains a wrong service name. At the same time it takes the time T1. When it receives the reply it takes the time T2. The difference between the two is in fact the response time of the instance. With this response time the binder can calculate the service rate of the instance.

This means :

The calculated response time : DELTA(i) = T2 - T1

$$\textit{The service rate : } MU(i) = \frac{1}{DELTA(i)} + LAMBDA(i)$$

When an instance becomes "present" state after having been in "not present" state, the evaluation of MU(i) equals 1/DELTA(i). But the only executed thread of the instance is the dispatcher for treating the wrong service name. Thus, the evaluation of MU(i) is better when there is a great number of connections because different kinds of services within the instance are in execution. The first calculation is made with zero as the number of connection. The indicator limit of the instance is switched to zero. Each time the number of connections exceeds the indicator limit then the binder calculates the DELTA(i) again, so as to evaluate a better MU(i). The indicator limit is switched to the value of the number of connections. Thus, the MU(i) is calculated according to the maximum number of connections the instance has got.

The variable service rate according to the number of connections

The response time R(i) of an instance(i) increases if MU(i) decreases or LAMBDA(i) increases. Indeed, if LAMBDA(i) increases it is possible that MU(i) decreases because the used sub-systems have a greater response time. This means that the service rate of the instance is variable according to the number of connections. In this case we have new formula.

The stochastic formulas with a variable service rate are following ...

$$TETA(i,N(i)) = \frac{LAMBDA^{N(i)}}{MU(i,1)*MU(i,2)*...*MU(i,N(i))}.$$

with MU(i,j) the service rate of the instance(i) which has j connections

$$R(i,N(i)) = \frac{TETA(i,N(i))*TETA(i,N(i))^{N(i)}}{(1-TETA(i,N(i)))*LAMBDA}.$$

Concerning this hypothesis the binder has to remember all MUs(i,j) for each i and for each j. Each time a change of state of an instance appears in the physical configuration, the binder evaluates MU(i,j) and R(i,j) for all different number of connections (j).

4.3.6- The time requested by the message exchange through the network

The binder has to evaluate TN regularly because the traffic rate of the network can change often. About the additional time of the network brought by the exchange of messages equals two times the time necessary for exchanging a message of a fixed length between two nodes. This time depends on the traffic rate of the network. Certain performance tools of the kernel have to be available for providing this time.

4.3.7- Different Poisson distribution rates of the connection

In this case, the average number of the service arrivals per instance for all connections equals the sum of the Poisson rate of each connection and not the Poisson rate multiplied by the number of connections. If an instance has connections with a low rate of arrivals and another which has connections with a big rate of arrivals then the first instance is not fully used with the strategy of the dispatching of connections. For example, an instance with thousands of messages per second and another instance of the same server with ten messages per hour. The problem here is that the binder does not know the rates of the client connections. The designer can make an approximation of a common Poisson rate per connection if he considers the loss of time is negligible.

4.4- The weight of the instance port

4.4.1- Introduction

Either there are different Poisson rates of the request arrivals per connection with a not important loss of time or it is not possible to evaluate the distribution rate of the arrivals or it is not possible to settle the distribution of the arrivals. In these cases, a new strategy has to be found ...

4.4.2- The cooperation protocol

The instance of the server knows the message number of its queue. To know the message number in the port of all instances of the same server it is necessary to establish a cooperation protocol [NEH2] between them. Obviously, the instances of the same server will have the same protocol. This protocol informs about the message number in the port and about other parameters such as the response time of each instance. It also permits the redirection of messages from an instance to another. With this protocol we have in fact the principle of the dispatching of ports of instance. The negociation of the message treatment is dynamical whatever the established connections may be. In fact, the binder establishes the connection between the client and an instance which are on the same node. Or it selects the instance which has the minimum number of connections.

4.4.3- The strategy : the dispatching of ports

For example, if you have two instances of the same server and the first has seven more messages than the other then the next message received by the first instance, will perhaps have to be treated by the second instance because its response time does not include the additional time brought about by the seven additional messages.

The additional time of an instance (i) is its number of messages in its port (N(i)) divided by the service rate MU(i).

This means :

$$\textit{The additional time of the instance(i)} = T(i) = \frac{M(i)}{MU(i)}.$$

such M(i) is the number of messages in the port of the instance(i)

$$\textit{The response time of the instance(i)} = R(i) = T(i) + \frac{1}{MU(i)} = \frac{M(i)+1}{MU(i)}$$

Here, each instance may calculate the additional time and thus there is no need for an evaluation to calculate it. The localization of the client is not important because an instance has actually received the message. The redirection of a message is necessary when the response time brought about by the instance is greater than the minimum response time of another instance added to the additional time brought about by the message exchange through the network.

This means :

Time required by the message exchanges on the network = TN
R(i) is the response time of the instance(i) which has the message
R(k)=Min{R(j) : for each j when the instance j is in "present" or "ready" state} and k not = i

If R(i) > R(k) + TN
Then the instance(i) makes a redirection of the message to the instance(k)
Else the instance(i) itself treats the message

4.4.4- The time requested by the calculation

The whole information about the response time of the instances is known by all instances. Each instance has to calculate its own response time and all of the instances have to communicate their response time to one another. The calculation of the response time of an instance is always made after the execution of a service of the instance. The response time of an instance(i) in "not ready" state equals zero. The variables are : the number of messages in the port (M(i)), the time required by the message exchange (TN) during the redirection and the service rate of the instance (MU(i)). The time requested for the evaluation of MU(i) is short. But the time of the evaluation of TN may be long, therefore this last evaluation may only be made sometimes and not each time we have a treated message of the instance.

4.4.5- The service rate of the instance

A sub-system may contain several services which have different service rates but thanks to the stochastic approximation we can replace those service rates by the service rate MU(i) of the instance(i). But here, it is the designer of the instance which chooses the distribution of this service rate (e.g. Erlang, Exponential, Cox, ...) and has to implement its evaluation in the server program. Nevertheless, it could approximate the service rate with the average time of the executed services as explained below.

The method for the evaluation of MU(i)

The dispatcher takes the time T1 before the execution of the service and the time T2 after the execution of the service. The difference DELTA(i) between those times is added to the sum of differences which is also the time TS(i) necessary to perform all services. TS(i) receives zero as initial value when the instance becomes "present" state after having been in "not present" state. The instance also increases the total number of requested services TM(i). TM(i) has zero as initial value when the instance is in "present" state after having been in "not present" state. The performing time TS(i) is divided by the total number of requested services TM(i) in order to obtain the service rate with the instance MU(i). With this service rate the instance can calculate the response time.

This means :

The performing time of the service = DELTA(i) = T2-T1

The performing time of all services = TS(i) = TS(i) + DELTA(i)

The total number of requested services = TM(i) = TM(i) + 1

$$\textit{The service rate} = MU(i) = \frac{TS(i)}{TM(i)}$$

4.4.6- The implementation of the cooperation protocol

The designer of the server has to implement the coordination of all instances of the same server. An automatic generator of cooperation protocol may be helpful with the evaluation of MU(i) which has just examined. The protocol has to inform about the response time of all instances and each instance has to know the port of all of the other instances.

The protocol may use the mechanism of ports and messages with a master/slave relationship. A better solution would be for each instance to update and read a shared distributed file which contains those response times and those ports.

4.4.7- The concept of ports group

The ports group (one port accessible by all instances of the same group) [ARMA] would seem a good implementation for the distribution of messages to the instances because the first instance which asks the message could be the instance with the least activity. Thus, it surely offers a better response time than the other instances. In fact, there is no loss of time in the information communication.

4.5- The Virtual Dispatching of Ports strategy

It is a particular and very attractive case of the two previous strategies but only if the connections are oriented service connections. The client makes a connection to a service rather than to a server, it receives the port of an instance from the binder, it calls its service directly and when it receives the reply of the service, it makes a disconnection to the instance. The dispatching of ports and the dispatching of connections is the same when you always have only one service per connection. What we have here is a merged strategy which uses only the advantages of the two above mentioned strategies leaving aside their inconveniences. For example, there are no protocols between instances, no redirection of messages and there is no evaluation of the additional time spent in the port of an instance. Instead of the instances the binder makes the dispatching of ports according to the number of connections. But the binder has to evaluate the response time of all instances requested for a connection and wherever they are. The binder has to use a better solution for evaluating the response time of the instance than the sending of messages which contain a wrong service name. It can do it because here the disconnection request is an additional information in this context.

The method for the evaluation of R(i)

When the binder makes a connection it takes also the time TC(i,C(i)) of the instance(i). After that it increases by one the number of requested connections C(i). When it makes the disconnection it takes the time TD(i,D(i)). After that it increases by one the number of requested disconnection D(i). Let's note that it never decreases the C(i) and D(i) values.

When an instance is in "present" state after having been in "not present" state, these values also equal zero. The binder calculates the difference DELTA(i,C(i)) between TD(i,D(i)) and TC(i,C(i)) when C(i)=D(i). The binder calculates the response time R(i,C(i)) of all connections of all instances wherever they are. If the instance(i) is not on the same node as the client (or as the binder) the response time R(i,C(i))equals the time difference DELTA(i,C(i)) minus the time brought about by the message exchanges through the network TN(C(i)). Otherwise the response time R(i,C(i))equals DELTA(i,C(i)). Also we have be aware of the different network time according to the evaluation of R(i,C(i)). Indeed, if A and B are inferior to C(i) then it is possible that TN(i,A) doesn't equal TN(i,B).

This means :

The difference of time : DELTA(i,C(i)) = TD(D(i)) - TC(C(i)) such D(i)=C(i)

If the client and the instance i are on the same node(i)
Then R(i,C(i)) = DELTA(i,C(i))
Else R(i,C(i)) = DELTA(i,C(i)) - TN(C(i))

For example, the connection of a service A of the instance(i) on the same node as the binder and the disconnection of the instance(i) after the performing of service A may give the following measures : TC(i,3) and TD(i,4). Indeed the binder has received a new connection request to the instance(i) which provides a new TC(i,4) and it also received a disconnection request which provides TD(i,3). This indicates a faster execution of a service B of the instance(i). This example demonstrates that the response time of an instance R(i) may not be only based on R(i,3) or R(i,4). In this example, the binder could approximate an average response time by :

$$\frac{(TD(i,4)-TC(i,3))+(TD(i,3)-TC(i,4))}{2} <==> \frac{(TD(i,3)-TC(i,3))+(TD(i,4)-TC(i,4))}{2}$$

Thus, the response time formulas are :

If the instance(i) is on the same node than the client
Then the response time of the instance i = R(i) = $\frac{SUM(\text{for each } j \text{ from } 1 \text{ to } C(i) : R(i,j))}{C(i)}$
Else the response time of the instance i = R(i) = $\frac{SUM(\text{for each } j \text{ from } 1 \text{ to } C(i) : R(i,j)-TN(j))}{C(i)}$

A common response time of an instance for all instances of the binder

For a given instance(i) several binder instances may calculate different response times. Each binder instance keeps its calculated response time and works with it. Thus, an instance(i) may have a great response time for a binder instance but a nil response time for another binder instance which has never received a connection request for this instance(i). A common response time would seem the solution of this problem ...

The common response time of an instance is the sum of the response times calculated by each binder instance divided by the number of binder instances. Each time a new common response time is calculated, the previous common response time has to be added to it. When the server is in "present" state the common response time has the value zero.

This means :

The response time calculated of an instance(i) by a binder instance(b) : $R_b(i)$
NBBINDER is the number of binder instances
The common response time :

$$CR(i) = \frac{SUM(\text{for each } b \text{ from } 0 \text{ to } NBBINDER : R_b(i)) + CR(i)}{NBBINDER}$$

Once a binder instance has calculated the common response time, all of the individual response times of each binder instance have to be nil again because the next calculation of the common response time has to be correct. The binder has to use its protocol or its common shared file to make the connection. This protocol informs about the state, the number of connections, and in addition about the individual response times and about the common response time for all instances.

The choice of the instance for the connection

The binder chooses the instance(k) which has the minimum common response time if there is no instance(i) in "present" or "ready" state on the same node.

This means :

There is no instance(i) in "present" or "ready" state on the same node as the binder

If RC(k)=MIN{RC(j) : for each j when the instance j is in "present" or "ready" state} and k not = i
Then the connection is made with the instance(k)

The binder chooses the instance(i) on the same node if this instance is in "present" or "ready" state and if it provides a common response time CR(i) which is not greater than the minimum common response time provided by another present or ready instance(k) plus the additional time for the messages exchange through the network.

This means :

TN is the additional network time
RC(i) is the response time of the instance(i) in "present" or "ready" state on the same node as the binder
RC(k)=MIN{RC(j) : for each j when the instance j is in "present" or "ready" state}

If i not = k and RC(i) > RC(k) + LN
Then the connection is made with the instance(k)
Else the connection is made with the instance(i)

5- The saturation of a port

When the number of messages in the port M(i) reaches the limit of the port (the port size) and perhaps also the ports of the other instances according to the dispatching strategies, then the binder instance or the instance(i) of the server may ask the loader to load a new instance to allow a better response time and a bigger capacity for treating messages. But the auto-replication indicator has to be specified with "on overflow of port" as value. The number of messages in the port is known by the instance. Depending on the used connection strategy it is the binder or the instance(i) of the server which asks to load a new instance. In the dispatching of ports strategy it is the instance which is the responsible for the loading request because it has to receive the port of the new loaded instance for the correct management of the cooperation protocol. In the virtual dispatching of port strategy the maximum number of messages (or of connections) in the port may be the port size plus one. In the dispatching of connections strategy it is the binder which is the responsible of the loading request. But it does not know the number of messages in the port of the instance(i) thus it has to evaluate it with the average number of messages in the port. Indeed, for this strategy there is a replication of an instance when the arrival rate of messages LAMBDA(i) to the instance(i) is greater than the service rate MU(i) of the instance. The port saturation may be defined with a maximum number of connections to the instance(i).

The formula of the maximal number of connections of the instance(i) :

Maximal number of connection of the instance(i) = MAX(i)

such MAX (i) LAMBDA < MU(i)*
*and (MAX(i) + 1) * LAMBDA => MU(i)*

6- The inactivity of a port

When the number of messages in the port M(i) reaches the value zero and perhaps also the the number of messages in the ports of the other instances according to the dispatching strategies then the binder instance or the instance(i) of the server may ask to unload an existing instance to the loader for allowing a better weight on a node. The number of messages in the port is known by the instance. Depending on the used connection strategy it is the binder or the instance(i) of the server which asks to unload an existing instance. In the dispatching of ports strategy it is the instance itself which is responsible for the unloading request. In the virtual dispatching of ports or in the dispatching of connections strategies the empty port of an instance is known by the binder because it has no connection with it. Thus, the binder is responsible for the unloading request. But the unloading request may fail because the instance to be suppressed is the only instance of the server in the physical configuration and is required by another instance of another server in the physical configuration. Indeed, the physical topology has to remain coherent.

7- The distribution and integration of the binder

There is a local call mechanism to the binder. This call mechanism is the mechanism of bourses and commissions which lets the binder know that the localization of the caller is on the same node. Thus the binder is an autonomous and local sub-system because it is used by all of the clients which are on the same node. In addition the replication of the binder may imply a better speed performance at the time of the (dis)connection request and increases the capacity of the binder for treating the (dis)connection services. But all instances of the binder have to secure coherent information about the server instances. Instead of adding a new local sub-system (the binder) the linker is extended with the new functions for (un)binding between clients and servers. This linker has to establish the (dis)connections between the clients and the sub-systems whatever their kind may be. The linker has some informations concerning all of the sub-systems to manage the (dis)connections.

8- The requested information

In summary this item gives the whole information necessary to the binder for its management with the VDP strategy.

For each server (present in the logical topology)

- The name
- The list of service names
- The auto-load indicator
- The replication indicator (on overflow of ports, only one, on all nodes, N times)
- The port size
- The critical number of messages or of the connections (VDP strategy)
- The state

For each instance of each server "present" in the logical configuration

- The node number where it is located
- The state
- The number of connections
- The Port
- The response time calculated by the binder $R_b(i)$
- The common response time RC(i)

For each connection request of each "ready" or "present" instance

- The number of connection request C(i)
- The corresponding time TC(C(i))

For each disconnection request of each "ready" or "present" instance

- The number of disconnection request D(i)
- The corresponding time TD(D(i))

For each node

- The number of the node
- The current state of the node (Crashed, Available)

For the network

- The network time TN(C(i))

Chapter 2 : Loading management

In the traditional operating system there already is a module called the creator which manages the (un)loads of the local sub-systems [SIM2]. The distributed operating system needs an extension of the creator or of another similar module. This extension or this additional module is called the loader. The loader is a set of services which establishes the (un)loads of the instances of the servers.

1- The entities requesting the (un)load [1]

1.1- An instance of a server

Each time its message number reaches the port limit, an instance may ask the loader for a new one. It does so with the help of the dispatching of ports strategy. But the value of the replication attribute of the server declaration would have to be "overflow of port". If an instance has an empty port then it may also ask the loader for an instance unload.

1.2- The binder

With the dispatching of connections or the virtual dispatching of ports strategies the binder may ask to the loader for a new instance of a server when the connection number of any instance of the same server reaches the port size plus one or reaches the maximum connection number which may be calculated from the formulas. The binder also requires the load of the first server instance when a client (a user task) wants to make a connection with this server in "not present" state. The value of the auto-load attribute has to be "YES".

1.3- The configuration manager

It is a special program which knows the physical configuration and which asks the loader to (un)load instances on a given node for efficiency reasons or for anticipation reasons. This program is only accessible by the master operators or by the system administrators. Its main goal is to establish a stable and efficient physical topology. This program is an option module of the sub-systems management.

---

[1] Those entities may also called the callers.

The main difference between the loader and the configuration manager is that the loader makes the (un)loading of an instance according to needs (overflow of port or auto-loading) or because of a lack of activity of the instance in the physical configuration (underflow of port), whereas the configuration manager uses other criteria such as the future evolution of the configuration. The configuration manager uses the loader.

1.4- The anchor

The anchor is a special service of the kernel that asks for the loading of the startup configuration.

2- The problem of the functional dependencies

2.1- The functional migration of the creator

When the creator has to load a local sub-system, it is possible that it requires the presence of other (local or global) sub-systems which may also be in "not present" state. Thus, the creator has to know the physical configuration of all sub-systems to be able to maintain coherency. The knowledge of the creator has to be extended with the information about the global sub-systems. Here we extend the creator with the functions of the (un)loading of the global sub-systems. This creator or loader makes the loading of instances of sub-systems and the unloading of those instances. The loader is a local and autonomous sub-system. This procedure may also be called the functional integration of the loader.

2.2- In the loading phase

2.2.1- The loading of the first instance of a sub-system

The loader has to load the first instance of a (local or global) sub-system (i) but it is possible that this sub-system requires the presence of other sub-systems which may also be in "not present" state. The loader may refuse this operation and return an error code. This is not a transparant procedure because the caller will have to ask the load of a certain logical sub-configuration before asking the load of the sub-system (i). A goal among others is to hide the structure of sub-systems from the clients (this also amplifies the oriented service connection rather than the oriented sub-system connection). Thus, the loader may not refuse the operation for this reason. The sub-configuration is called the loading sub-configuration.

The root of this loading sub-configuration is the initial requested sub-system (i) for the loading. This loading sub-configuration is constructed with the "use" relationship and the sub-systems which are in "not present" state. There is always a sub-system (j) of the loading sub-configuration which doesn't use another sub-system of the same sub-configuration. First of all the loader has to load this sub-system (j) which is then in "present" state and so it doesn't belong to the loading sub-configuration anymore. The loader repeats its work until there are no more sub-systems in the loading sub-configuration.

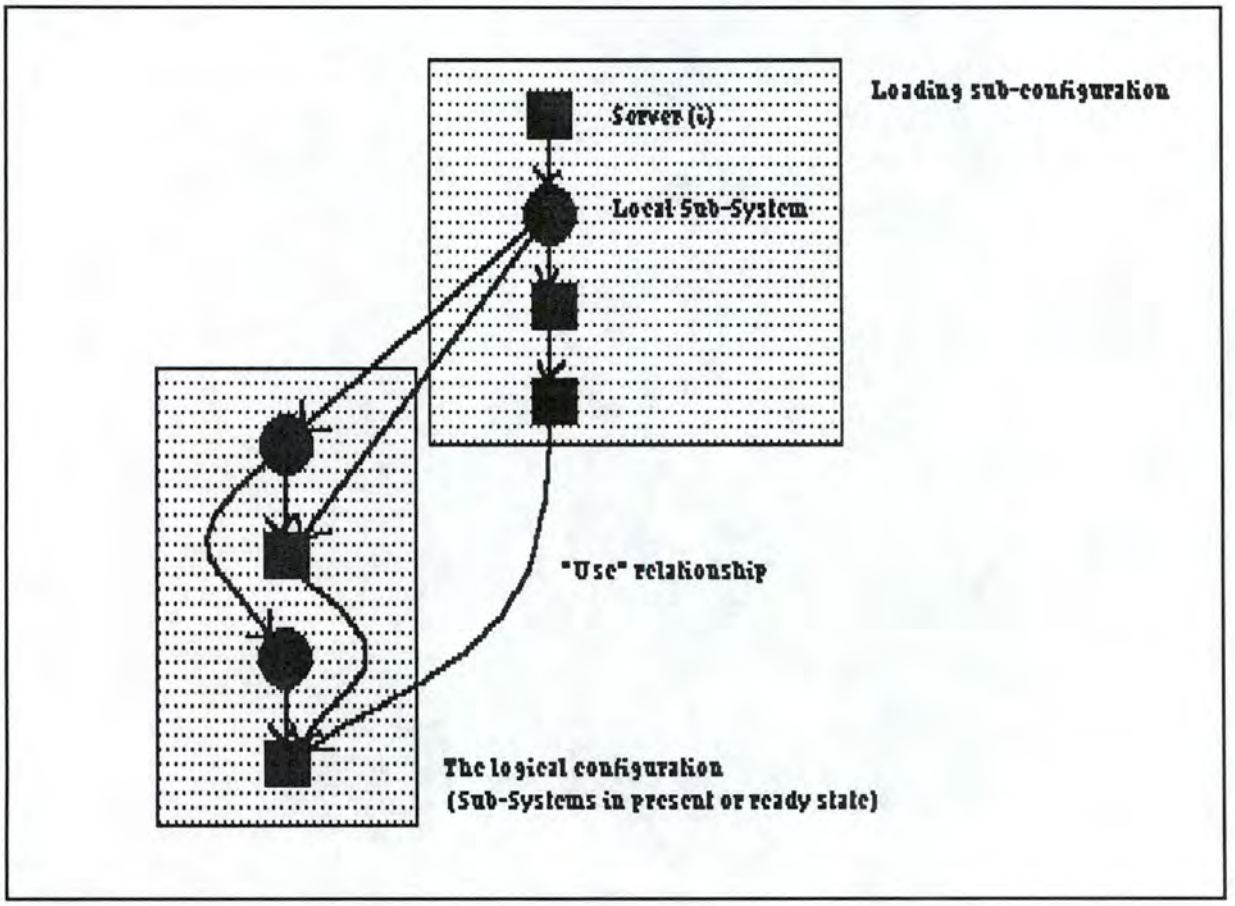

Figure 10 *The loading sub-configuration*

2.2.2- The loading of other instances of the same sub-system

The loading of an additional instance of the same sub-system doesn't create additional functional problems. But the loader has to guarantee the coherence of the physical configuration (See the chapter about the topologies of sub-systems). As we mentioned earlier, the redundancy of instances increases the capacity of the server and decreases the response time of the instances.

2.2.3- The explicit load of an instance on a given node

The explicit load would like to perform a loading of a server instance on a given node. But this does not correspond with the principle of the single image because the explicit loading is not a transparant mechanism. Thus, we can't left this possibility to the callers. Nevertheless, the loader has to provide this service to the configuration manager, the system administrators or the master operators.

2.2.4- The non recurrent loader

Because the loader is an autonomous sub-system, the algorithm of the loading phase could be recursive. Each time a sub-sytem (i) has to be loaded and if it requires the presence of a "not present" sub-system(j), the loader could call itself to perform the loading of the sub-system (j). But this doesn't correspond with the principle of a hiearchy "use" of sub-systems.

2.2.5- The initialization of an instance

The loader has to initialize the sub-system instance by calling the initial service of this instance. The name of this initial service is specified in the sub-system declaration. After that the loader changes the state of the sub-system instance in "present" state. Thus, this initial service is made without a connection establishment.

2.3- In the unloading phase

2.3.1- The unloading of the last instance of a sub-system

The loader receives an unloading request of the last instance of a sub-system in the physical configuration. If this sub-system is required by others or if this instance is in "not present" or "ready" state then the loader refuses the operation and returns an error code. Else, the loader may accept the operation. But if the instance is required again after a short period then the unloading is useless. The loader accepts the unloading of an instance if after a long period of time the instance state has not been changed in "ready" state. The unloading service has a great response time. In the dispatching of ports strategy there are two solutions for unloading the last instance : either it is the binder which requests the unloading or it is the last instance which requests the unloading from itself. In the second case the instance has to make the disconnection to the unloading service in its ending service. More over, the loader will receive an error code for its reply because the last instance will no longer exist.

2.3.2- Termination

The loader changes the state of the sub-system instance into "not present" state. After that the loader has to terminate the instance by calling the termination service of this instance. The name of this terminal service is specified in the server declaration. Thus, this ending service is made without a connection establishment.

2.3.3- The unloading sub-configuration

We could imagine a construction of an unloading sub-configuration based on the "used by" relationship and by creating new states for the instances such as the Suspended state which prevents the new connection with an instance. It would always have an instance which would not be used by another. But it may be used by a client which is a user task ... That may bring about a lot of loading and unloading of the same instances in a short period of time and therefore for efficiency reasons we cannot allow it. Therefore, this function is made by the configuration manager.

2.3.4- The unloading of another instance of the same sub-system

To unload another instance of the same sub-system which has to be in "present" state the loader makes the operation without waiting a long time.

2.4- Possible extentions : the configuration manager

In this paragraph a few possible functions are given as examples of extension. The configuration manager may collaborate with the linker and the loader if a new state is taken into account : the suspended state which prevents the new connections to an instance.

2.4.1- The purge

As one goal among others for the loader is to increase the performance per node and also to decrease the number of tasks per node then the configuration manager eliminates the instances which are not used by another and which are in "present" state.

2.4.2- The clear unloading of instances of a sub-system

The configuration manager may reduce the number of instances of the same sub-system even if they are connected. That may be designed by adding the Suspended state to the instances. For example, it may do so according to the number of connections per instance which is too low.

2.4.3- The clear unloading of instances of all sub-system

It is the same as the previous function but for all sub-systems.

2.4.4- The optimization of the physical configuration

The configuration manager optimizes the configuration either according to the node weight (suppression of replicated instances) or according to the network weight (replication of instances).

2.4.5- The stable configuration

This function is designed to prevent the (un)loading of instances. The configuration manager has to anticipate the future (un)loading. An expert system may do it if it has knowledge about the transitions of the configuration. A configuration is stable if there is little (un)loading. The goal also is to prevent the auto-loading when a connection appears.

3- The problem of the coherency of the physical configuration

Indeed, an instance of a sub-system(s) to be loaded can use instances of local sub-system. Some of them may be in "not present" state on the node(i). Then the loader has to load first those local sub-system instances on the same node. In addition, it is possible for those local sub-system instances do use other local sub-systems. This defines a set of local sub-system instances which is called the local sub-configuration of the sub-system(s). First the loader has to load on the node(i) a local sub-system of the local sub-configuration which is in "not present" state and which does not use another local sub-system which is in "not present" state. It repeats this until all of the instances of the local sub-configuration are in "present" or "ready" state on the node(i). Before unloading an instance of a local sub-system on a given node(i) this instance may not belong to any local sub-configuration of instances on the node(i).

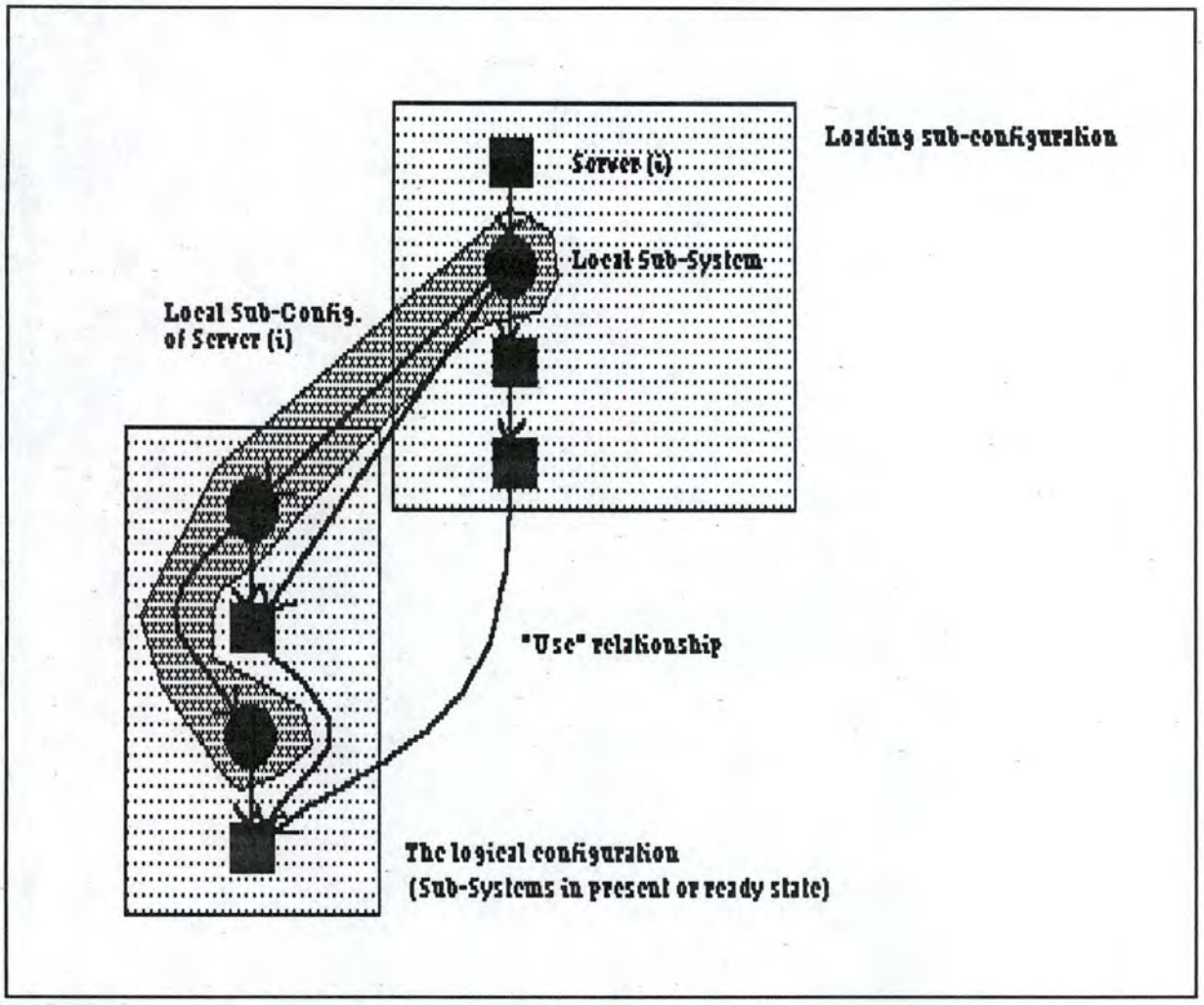

Figure 11 *The local sub-configuration of a sub-system*

4- The connection to the loader services

Many connection modes may be imagined but they are used only by the linker, the administrators, the operators and the configuration manager.

5- The dynamical components

Each loader service may be a thread and it is performed in the context of the loader. The mechanism for calling the loader may be hidden by using macros or commands ($load,$unload).

6- The interfaces of the loader

There are already two existing interfaces provided by the loader : the local sub-system loading and the local sub-system unloading. There are two new basic services provided by the loader : the instance loading and the instance unloading. This means that the caller has to know the kind of sub-system to be (un)loaded.

ServerLoad(server-name,port,ret-code)

Input parameters :

server-name : the server name.

Output parameters :

port : the port of an instance of the server if ret-code = "ok"

ret-code : the return code of the operation (ok or fail).

InstanceUnLoad(port,ret-code)

Input parameters :

port : the port of the instance to unload.

Output parameters :

ret-code : the return code of the operation (ok or fail).

6.1- The extended interfaces

The loader has to provide mainly two other services to the configuration manager, operators and administrators. They are : the loading of an instance on a given node and the clear deletion of an instance. These services are faster and are able to manage the suspended state with a collaboration between the configuration manager and the loader. This thesis does not explain how the loader manages this state and those services.

FastInstanceLoad(server-name,node,port,ret-code)

Input parameters :

server-name : the server name.

node : the node where the instance has to be

Output parameters :

port : the port of an instance of the server if ret-code = "ok"

ret-code : the return code of the operation (ok or fail).

FastInstanceUnload(port,ret-code)

Input parameters :

port : the port of the instance to unload.

Output parameters :

ret-code : the return code of the operation (ok or fail).

7- The problem of the localization for loading an instance

The main problem to be solved is the best installation of a sub-system instance in the physical configuration. The loader has to construct a physical topology allowing for better response time in the future connections between clients and sub-systems.

7.1- Network considerations

Some physical configurations may bring about more traffic through the network. The problem to be solved is the minimization of the "use" relationships between the sub-systems instances of different nodes. A lot of constraints may appear (i.e. the loading sub-configuration, the local sub-configurations, the localization of the caller, the replication indicator, ...). The problem and those constraints may be formulated by an equation system. The SIMPLEX algorithm [DANT] may solve this equation system with a minimization of the network exchanges.

The formulation of the problem and its constraints with the network viewpoint :

The variables :

r,s,t ,b: integer ranged from 1 to NBSS such NBSS is the number of different sub-systems which belong to the logical configuration and to the loading sub-configuration

i,j,k,a(b) : integer ranged from 1 to NBNODE such NBNODE is the node number

GAMA(r,i), sign(r), egal(i,j) : boolean

NS(r) : integer

C(a(1),a(2),...,a(NBSS)) : integer

Z : integer

The facts :

The "use" relationship between sub-systems

R(s,t) : boolean
R(s,t) = 1 if it exist a "use" relationship between the sub-system(s) and the sub-system(t)
= 0 else

The average number of calls to a sub-system(t) from a sub-system(s) for realizing a service of the sub-system(s)

NC(s,t) : integer
NC(s,t) = 0 if R(s,t) = 0
=> 1 else

The physical configuration before the loading request

ALPHA(r,i) : boolean
ALPHA(r,i) = 1 if there is an instance of sub-system(r) in "present" or "ready" state on the node(i)
= 0 else

The local sub-systems

L(r) : boolean
L(r) = 1 if the sub-system(r) is a local sub-system
= 0 else

The replication of a sub-system

Linf(r), Lsup(r) : integers ranged from 1 to NBNODE
Linf(r) = n if the sub-system(r) is replicated at least n times
Lsup(r) = m if the sub-system(r) is replicated at most m times

The information coming from the loading request

s0 : an instance of the sub-system(s0) has to be loaded or replicated
l0 : the node of the caller (and also of the loader)

The rules

The physical configuration after the loading request includes all instances of the physical configuration before the loading request

$GAMA(r,i) => ALPHA(r,i)$ *for each r, for each i*

If the requested sub-system is a local sub-system then it has to be found on the same node of the caller (if this local sub-system exits already on the node(l0) then the request fails)

$GAMA(s0,l0) => L(s0)$

If the requested sub-system is a server and if there is at least one instance of this server in the physical configuration then it is a replication request which implies only one more instance (the request has to be checked previously by considering the number of nodes, the replication indicator and its existing number of instances)

$SUM(\text{for each } i : ALPHA(s0,i)) * sign(s0) = 0$
$SUM(\text{for each } i : ALPHA(s0,i)) + sign(s0) <> 0$

$SUM(\text{for each } i : GAMA(s0,i))*(1-sign(s0)) =$
$SUM(\text{for each } i : ALPHA(s0,i))*(1-sign(s0))+(1-sign(s0))$

The coherency of the physical configuration

1- $SUM(\text{for each } i : GAMA(r,i)) => 1$ *for each r*
2- $GAMA(s,i) => GAMA(t,i)*R(s,t)*L(t)$ *for each s, for each t, for each i*

The replication of instances

$SUM(\text{for each } i : GAMA(r,i)) => Linf(r)$ *for each r*
$SUM(\text{for each } i : GAMA(r,i)) <= Lsup(r)$ *for each r*

The number of calls to the sub-system(t) by other sub-systems for one call to the sub-system(s0)

$NS(s0) = 1$
$NS(r) => 0$ *for each r*
$NS(t) = SUM(\text{for each } s : NC(s,t)*NS(s))$ *for each t*

The number of network exchanges between all instances brought about by one call to the sub-system(s0)

$$C(a(1),a(2),\ldots,a(NBSS)) = SUM(\text{for each } s : SUM(\text{for each } t : NS(s)*NC(s,t)*egal(a(s),a(t)))) * PROD(\text{for each } s : GAMA(s,a(s))) \text{ for each } a(1), \text{ for each } a(2), \ldots, \text{ for each } a(NBSS)$$

such $NC(s,s) = 0$ for each s

such $(1-egal(i,j))*(i-j)=0$ and $(1-egal(i,j))+(i-j)<>0$

Example given : C(1,1,2,2) = 5 indicates there are instance of sub-system 1 on node 1, an instance of sub-system 2 on node 1, an instance of sub-system 3 and of sub-system 4 on node 2, and there are 5 network exchanges brought about by their "use" relationships and the average number of calls between them

The minimization (economical function)

$$Z = SUM(\text{for each } a(1) : SUM(\text{for each } a(2) : (\ldots (SUM(\text{for each } a(NBSERVER) : C(a(1),a(2),\ldots,a(NBSERVER))) \ldots)))$$

$$MIN(Z)$$

This system of equations implies as little replication as possible because the replication of a single instance brings about a new GAMA(...) >0 and thus new possible C(...) > 0. This system gives all possible network exchanges between the instances. Another solution could be the replication of each instance on each node. But a lot of network exchanges are still possible with the dispatching strategies ... In addition, the weight of each node is bigger and the response time of the loading service increases strongly.

7.2- Task scheduling considerations

The chapter about the binder has taken the following hypothesis into account : the number of tasks is nearly the same on each nodes and also the CPU occupied rate has to be nearly the same on each nodes. In this way each task in the distributed system has the same scheduling environment. The global performance of the single image machine would have to be at a maximum. The task balancing and the CPU occupied rate balancing are insured by the loader and the job manager. The loader puts the instance on a certain node according to the number of tasks per node and the CPU occupied rate per node.

The properties of a sub-system could be taken into account [1]. The knowledge of those parameters is furnished by certain services of the kernel.

The formulation of the problem and its constraints with the task scheduling viewpoint :

The additional variables :

W(i) : integer
Y : integer

The additional facts :

The number of tasks of the node before the loading request

V(i) : integer

The additional rules

The number of tasks after the number request

W(i) = V(i) + SUM(for each r : (GAMA(r,i)-ALPHA(r,i)))

The new minimization

The goal is to minimize the number of tasks per node and to minimize the difference in the number of tasks between the nodes.

Y = SUM(for each i : W(i)) + SUM(for each j : SUM(for each k : W(i)-W(k)))

Min Y

Please note that the formulation does not take into account of the autonomous sub-system, the cpu occupied rate and so on ...

[1] This criterion may have a lot of influences on the CPU occupied rate and also on some scheduling priorities (e.g. interactive kind, i/o oriented, ...). It is mainly in a loosely coupled architecture this criterian has a great roll because the loader makes an association between the properties of the node (workstation, arithmetical pocessor, ...) and the properties of the sub-system (oriented processing, oriented i/o, interactive, ...).

7.3- Multicriterians analyse

The balancing of the node weight brings about more network exchanges. When Z decreases then Y increases. When Y decreases then Z increases. But if the goal of Y is to minimize only the number of tasks of node then Z and Y may have a positive or nil correlation. The balancing of tasks may be made either during the loading of tasks if those tasks do not use other or during the loading of a new instance (replication).

7.4- Time requested by the calculation

The time required by the resolution of the equation system seems quite long. It is the reason for giving stability to the physical configuration.

7.5- Reoptimization and anticipation

Normally the given strategies for the loading and the binding offer the best response time for all of the services at any given time. In the traditional operating system there aren't any unloads because a startup and a shutdown occurs every day. But in the distributed system those concepts tend to disappear therefore loads and unloads are more frequent. In the future the physical topology might not be the best because there are a lot of replications and a lot of deletions of instances. The configuration manager may evaluate regularly the localization of the instances of the physical configuration and make some operations to tend towards the ideal physical topology for a given logical topology. That is the concept of optimization of the physical configuration. But if the configuration is stable it is not necessary to reoptimize. The stability of the physical configuration may also be reached by the configuration manager. To avoid the unloading and loading of the same instance within a short period of time, the loader may anticipate by using an expert system. The expert designer has to deliver knowledge about certain physical topologies and their assumed transitions (e.g. of a knowledge : in this given physical configuration it is interesting to load this instance of this sub-system on this node). The knowledge may be based on the loading and the unloading frequency of the instances. The utilization of an expert system is not the most efficient but it also has a limited importance.

8- The distribution and integration of the loader

8.1- The functional analysis for the distribution

Perhaps it is interesting to split the loader into 2 sub-systems : a high level loader sub-system and a low level loader sub-system because there are two different concepts : the single image machine and the node. Thus the designer has to define the reciprocal actions between them. There are two kinds of reciprocal actions : the "use" relationship for the utilization of services and the coherency of the common information about the sub-systems and instances.

The high level loader

It has to (un)load a sub-system on the single image machine. It makes some validations (the (un)load phase) and some calculations with the SIMPLEX algorithm (the localization of an instance). It calls the low level loader to make the (un)loading of an instance in or from a task on a given node and to gather some information about the node. It is not necessary to replicate the high level loader on each node because there is no need for a high speed performance and the loader is not often requested. The time cost of (un)loading has no influence on the call time and on the (dis)connection time between sub-systems. Excepted for the connection to a sub-system for which there isn't yet a single loaded instance (the auto-loading phase). Then the time loss for this first connection kind may be called an investment time cost.

The low level loader

It manages the (un)loading of an instance of a sub-system in a task or in a holder task on its node without other considerations and it provides some node inforrmation to the high level loader. The low level loader directly uses the Nucleus services. The low level loader has to be replicated on each node because the (un)loading implies the utilization of a particular kernel interface. This interface is the (dis)allocation of task on the node where the requester (the low level loader) is situated. The principle of the single image machine leads us to assume there are requesters on each node.

8.2- The organic analysis

The high level loader provides the interfaces we have seen before but it has to know the localization of the caller.

Either it is accessed with the local call mechanism or it is accessed by a global call mechanism with the establishment of a protocol with the caller. This solution brings about a lot of problems : the performance decreases and the implementation complexity increases strongly (the interfaces have to be changed by adding the localization of the caller to the services arguments). Thus, it is not interesting to split the loader into two sub-systems. The loader has to establish the (un)loading on a node of the sub-systems whatever their kind may be. For managing the (un)loading this loader has some informations concerning all of the sub-system kinds.

9- The requested information

In summary this item gives the whole information necessary to the loader for its management activity.

For each sub-system ("present" in the logical configuration)

- The name
- The list of the service names
- The replication indicator (on overflow of ports, only one, on all nodes, N times)
- The port size
- The state
- The scope (local or global)
- The property (interactive, batch, ...)
- The name of the library wich contains the server code
- The code size
- The call mecanism kind (FITC, BALR, ...)
- The temporal dependency (BeForeSystemReady, ...)
- The list of the functional used sub-systems
- The averaged number of calling of the functional used sub-systems
- The name of the initial service
- The name of the terminal service

For each instance of each sub-system "present" in the logical configuration

- The node number where it is located
- The state

For the server :

- The Port

For the local sub-system :

- The corresponding holder task identificator
- The offset in the holder task
- The list of the entries and their address

For each holder task

- The holder task identificator
- The node number
- The free size space
- The number of included instance

For each node

- The number
- The state

PART 4

Conclusion

The idea

For the same cost a closely multiprocessor system seems be as efficient as a tightly multiprocessor system. But with certain properties (e.g. the reliability, the performance, the transparancy, ...) the distributed operating system has to manage X nodes to give the impression of a single image machine to the user ....

Services and Sub-systems

From a conceptual or functional point of view there are only three concepts : the client concept, the service concept and the sub-system concept. The functional sub-system topology indicates the relationships existing between them. The Client/Server model is the communication model which is always available whatever the distributed operating system.

From a physical point of view the generic aspect, the context of performing, the call mechanism the synchronization aspects and the possible level of parallelism allow the definition of several kinds of services and sub-systems. The sub-system architecture gives an overview of those elements.

From the distribution point of view there is a concept of instance instead of sub-system. The physical sub-system topology indicates the relationships between instances and their localizations on the nodes.

From an application programmer point of view the Remote Procedure is the concept which replaces the global service. The remote procedure call hides the concept of service, the structure of sub-systems, the port and message mechanism, ... Some new languages (i.e. Concurrent C++ and Parallel Fortran of IBM) allows to use the thread parallelism.

From the concurrence point of view the monitors, the common variables, the shared and distributed files and the inter-task communication with a protocol are the support of the service concurrency.

The fault-tolerance of sub-systems and the security of message transfer between a client and a sub-system may be guaranted by the kernel.

The sub-systems management

The management of sub-systems is insured by fives modules : the generator, the linker, the creator, the configuration manager and the anchor, and it requires two files : the catalog file and the configuration file. The scope of my thesis was to establish strategies for the binding to a sub-system and the loading of a sub-system in such a way that their services provide a response time as rapidly as possible. My thesis will be used by engineers of Siemens-Nixdorf as a start point for researches about dynamic sub-systems management in the future BS2000 distributed operating system.

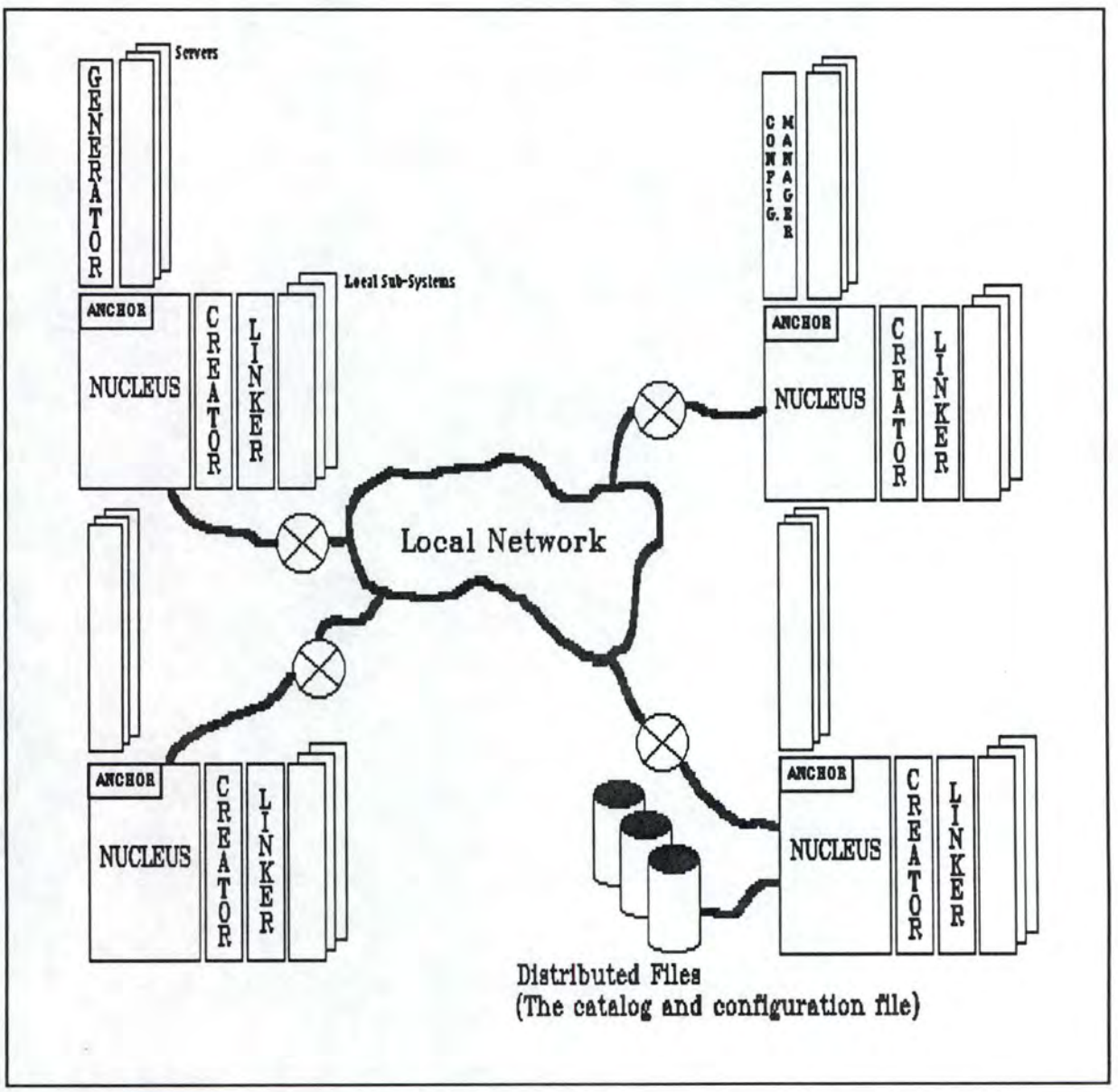

Figure 12 *The Sub-Systems in a Cluster*

The vertical limits of this thesis

This thesis could be extended by giving a deeper analysis of some items such as the configuration manager and its functions (the development of an expert system and of the knowledge base about the transitions of the physical configuration), the startup phase and the anchor, the generator and the declaration of sub-systems, the estimation of LAMBDA, the statistical validation of the evaluation of the response time (e.g. bias in the estimation, correlation between taken actions, ...), the simulation of the binding and the loading given strategies, the automatic generation of client and server stubs, the binding strategies if there are connections oriented service and oriented server, the multicriteria formulation for the loading management (relation between the weight of node and the network traffic), ...

The horizontal limits of this thesis

This thesis could also be extended by analysing some new areas such as the "loop" relationship between sub-systems, the port group mechanism, the environment provided by a loosely coupled multiprocessor system, the environment provided by a decentralized operating system, ...

List of references

Multiprocessor systems

[COMT] COMTRE CORPORATION, *"Multiprocessors and Parallel Processing"*, John Wiley & Sons Interscience Publication

[DUCE] Duce D.A., Jones G.P., *"Distributed Computing"*, Academic Press, 1984

[HWAN] Hwang K. (University of Southern California), Briggs F. A. (Rice University), "Computing Architecture and Parallel Processing" , McGraw-Hill Book Company

[KRON] Kronenberg N. P., Levy H. M., Strecker W. D., *"VAXclusters : A Closely-Coupled Distributed System"* , Digital Equipment Corporation

[TOOM] Toomey L. J., Plachy E. C., Scarborough R. G., Sahulka R. J., Shaw J. F., Shannon A. W., *"IBM parallel FORTRAN"* , IBM system journal vol 27/no 4, 1988

Distributed systems and Distributed Operating Systems

[ARMA] Armand F., Gien M., Herrmann F., Rozier M., *"Chorus : Revolution 89 or Distributing UNIX brings it back to its original virtues"*, Chorus systèmes, août 89

[BABA] Babaoglu O., *"Fault-Tolerant Computing based on MACH"*, ACM Operating System Review, january 1990

[CORN] CORNAFION, *"Informatiques répartis : concepts et techniques"*, Dunod Informatique, 1981

[COUL] Coulouris G. F. (Queen Mary College. University of London), Dollimore J. (Q. M. C. University of London), *"Distributed Systems : Concepts and Design"* , Addison-Wesley Publishing Company, 1988

[DRAV] Draves R. (School of Computer Science Carnegie Mellon University), *Slides of the conference given by R. Draves about MACH*, München 1/10/90

[OSF1] Open Software Foundation Cambridge, *"OSF Distributed Computing Environment Rationale"*, 14 mai 1990

[OSF2] Open Software Foundation, *"Distributed Computing Environment : Overview"*

[PUBL] Public/draft, *"Chorus Kernel v3.2 Specification"*, jan. 90

[RAS1] Rashid R. F., *"Threads of a new System"*, Unix review, aug. 86

[RAS2] Rashid R. F. (Computer Science Department of Carnegie-Mellon University), *"From RIG to Accent to Mach : the Evolution of a Network Operating System"*, may 86

[RUSS] Russel C. H., Waterman P. J., *"Variations on Unix for Parallel Processing Computers"*, Communications of ACM vol. 30/number 12, dec. 87

[SAMS] Samsom R. D., Julin D. P., Rashid R. F., *"Extending a Capability Based System into a Network Environment"*, april 86

[SEIF] Seifert M., *"What is a Distributed Operating System ?"*, IBM european networking center

[TEVA] Tevanion A. J., Smith B., *"Mach : the Model for future Unix"*, BYTE, nov. 89

[WAIM] Waimer L. R., Thompson M. R. (Carnegie-Mellon University), *"A MACH Tutorial"*, aug. 87

Service Modeling

[NEH1] Nehmer J. (Kaiserslautern universität), *"AG SystemSoftware"*, FB informatik von universität Kaiserslautern

[NEH2] Nehmer J.,Mattern F., (Kaiserslautern universität), *"Service Modeling in Distributed Operating Systems"*, 2 nd workshop on future trends of Distributed Computing Systems in Cairo, 30/09/90

Remote Procedure Call and Transport protocols

[BERS] Bershad B. N., Anderson T. E., Lazowska E. D., Levy H. M., (University of Washington), *"Lightweight remote procedure call"*, ACM Transactions on Computer Systems Vol. 8/no 1, feb. 90

[ECMA] ECMA, *"Basic Remote Procedure Call protocol using OSI Remote Operations"*, 1987

[SCHA] Schaeck S., *"A contribution to the analysis ot the transport protocols"*, promoteur du mémoire : van Bastelaer P., Facultés Universitaires Notre-Dame de la Paix à Namur, 1989/1990

[SCHR] Schroeder M. D., Burrows M., (Digital Equipment Corporation), *"Performance of Firefly RPC"*, ACM Transactions on Computer Systems Vol. 8/no1, feb. 90

[TANE] Tanenbaum A. S., van Renesse R., (Department of Mathematics and Computer Science, Vrij universiteit van Amsterdam), *"A critique of the Remote Procedure Call Paradigm"*

[TAY] Tay B. H., Ananda A. L., (Departement of Information Systems and Computer Science, National University of Singapore), *"A Survey of Remote Procedure Calls"*, ACM Operating Review vol 24/n 3, July 90

Siemens-Nixdorf publications

[SIM1] Siemens-Nixdorf, *"Nucleus BS2000 : technical description"*

[SIM2] Siemens-Nixdorf, *"External Interfaces Specification of the Management of Local Sub-Systems"*, EIS vol 104/BS2000 nucleus 3/chapter 52/section 2, distribution restricted by Siemens-Nixdorf

[SIM3] Siemens-Nixdorf, , *"External Interfaces Specification of the Fast Inter-Tasks Communication"*, EIS vol 12/BS2000 nucleus 1/chapter 52/section 35.6, distribution restricted by Siemens-Nixdorf.

Simulation and Performance

[DASS] Dass P. K., Baghi K. K., Bhaumik B. B., *"Validated Analytical Technique for Multiple Microprocessor Architectures"*, Computer Performance vol 5/no 3, September 1984

[PAWL] Pawlikowski K., *"Steady-State Simulation of Queueing Processes : a Survey of Problems and Solutions"*, ACM Computing Surveys vol 22/no 2, June 1990

[SAUE] Sauer C. H. (IBM research center), Chandy K. M. (University of Texas at Austin), *"Computer Systems Performance Modeling"*, Prentice-Hall

Systems of equations

[DANT] Dantzig G. D., *"Linear Programming and Extensions"*, Princeton-University Press, 1966