An overview of the gothic distributed operating system

Jean-Pierre Banâtre, Michel Banâtre, Florimond Ployette

▶ To cite this version:

Jean-Pierre Banâtre, Michel Banâtre, Florimond Ployette. An overview of the gothic distributed operating system. [Research Report] RR-0504, Inria. 1986. inria-00076050

HAL Id: inria-00076050
https://inria.hal.science/inria-00076050

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA
CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 504

AN OVERVIEW OF THE GOTHIC DISTRIBUTED OPERATING SYSTEM

Jean-Pierre BANATRE
Michel BANATRE
Florimond PLOYETTE

Mars 1986

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

Publication Interne n° 284
24 pages - Janvier 1986

An overview of the GOTHIC Distributed Operating System *.

Jean-Pierre BANATRE
IRISA/INSA and INRIA-Rennes,

Michel BANATRE, Florimond PLOYETTE
IRISA/INRIA-Rennes,

Campus de Beaulieu, 35042, RENNES-cédex (FRANCE)

Abstract:

This paper introduces a distributed operating system currently under development at IRISA/INRIA-Rennes. This system is based on the new notion of multi-function allowing the description of parallel tasks which may be nested in a very general fashion. Multi-functions should possess the property of atomicity. The implementation of this property led us to design a new object oriented and highly performant stable storage facility. Multi-functions and the overview of different levels of the architecture constitute the major topics developed in this paper.

Key Words: Distributed Systems, Multi-functions, Stable Storage, Atomicity, Nested Activities.

Résumé:

Cet article introduit un système opératoire distribué en cours de développement à l'IRISA/INRIA-Rennes. Ce système est basé sur la notion de multi-fonction qui permet la description de tâches parallèles imbriquées. Les multi-fonctions possèdent la propriété d'atomicité. La mise en œuvre de cette propriété nous a conduit à la conception et la réalisation d'une mémoire stable rapide adaptée à la gestion d'objets structurés. Les multi-fonctions et la présentation des différents niveaux de l'architecture du système constituent les principaux points traités dans cet article.

Mots clés: Systèmes distribués, multi-fonctions, mémoire stable, atomicité, activités imbriquées.

*The GOTHIC project is partially supported by the BULL Company since 01-01-1986.

1-Introduction.

Near future, computers have to be fast, flexible and highly reliable. One way of achieving these goals is to build computers from a set of basic modules assembled together to realize "machines" different in size and power.

Such "computer installations" will soon be available as domestic users will be equipped with workstations composed of a powerful CPU and a quite large amount of memory. Provided that the interconnexion network be fast enough, one can imagine that every user will possess his/her workstation and an access link to part of computing power of the other users. The role of an ideal operating system for such an installation would be to create an integrated environment in order to allow any user to take benefit of the largest possible amount of memory and computing power, in a transparent way.

Two recent systems apply the same type of approach. The Apollo DOMAIN which provides network transparent data access [LEAC-83] and the SUN network file system which allows transparent access to files located on remote machines [LYON-84].

Of course, as users are sharing memory and computing resources, where each resource belongs primarily to its proper owner (a workstation belongs to somebody) and secondly to the computer installation made out of all workstations linked through the network, one can imagine that several constrainsts have to be met: fault-tolerance, protection (personal information should not be accessed by anybody)...

The GOTHIC system presently developed at IRISA aims in providing such a general distributed system. In section 2, we describe the background of this research. Some basic structuring facilities of GOTHIC are then described, in particular we introduce the concept of multi-function as the primary means of describing operational behaviour and communications (section 3[1]). GOTHIC structure is presented in section 4. In particular, we insist on stable memory management[2] which is one of GOTHIC original features. Section 5 concludes by describing open problems and plans and giving the present status of the project.

1: Section 3 corresponds to the paper [BANA-86c] which will be published in the proceeding of the 6th Conference on Distributed Computing Systems, May 1986.

2: Stable storage board has been patented under n° 85 18437, dec. 1985.

PAPIER RÉCUPÉRÉ ET RECYCLÉ

2-Background.

GOTHIC is assumed to be a general purpose distributed operating system designed by generalizing concepts previously implemented in the application-oriented distributed system, ENCHERE [BANA-84, BANA-86a]. We think it worthwhile to recall some salient contributions of ENCHERE.

2.1. ENCHERE software structure.

The essential structuring facility of ENCHERE is the activity. An activity is a set of "cooperating processes" which interactions define the dynamic structure of the application. Activities can be nested at any level, thus an application is represented by a tree af activities.

Two nested activities (mother and daughter) can communicate only via parameter passing (when the mother "creates" the daughter) and via result transmission (when the daughter terminates).

An activity may also possess the property of atomicity:

-Indivisibility: its intermediate states are hidden to other activities,

-Recoverability: any object currently being modified can be restored to its initial state in case of failure.

The implementation of these properties relies upon two facilities: stable storage and commit protocols. As the main originality stands in stable storage, let us describe it briefly.

2.2. ENCHERE stable storage.

To support failure atomicity and permanence for small objects, the ENCHERE project has developed a stable storage device in RAM. The stable RAM has 8 non-volatile memory banks mapped into the address space of the application processor. Hardware faults in these banks are masked by sequentially writing stable objects into two banks. Because the non-volatile RAM is intended to be used by multiple processes, there are mechanisms that protect against software faults, such as uncontrolled memory accesses [BANA-83].

2.3. From ENCHERE to GOTHIC.

The design and implementation of nested atomic activities is the major achievement of the ENCHERE project. Two aspects have to be emphasized:

-this is one of the few actual distributed implementations of nested activities.

-an original stable storage facility was designed and built. Its use was determinant in the implementation of atomic activities.

However, due to the specificity of the application, some simplifications in the implementation of nested activities and of atomicity were considered as reasonable... The concept of activity, although very useful, was not fully investigated and further studies were considered as necessary. These observations led us to consider the generalization of ENCHERE concepts in order to produce a more general distributed operating system: this is the origin of GOTHIC.

3. GOTHIC structuring facilities.

GOTHIC handles objects through the use of appropriate operations on concrete data structures. Before presenting GOTHIC objects, let us describe the notion of multi-function.

3.1. Multi-functions.

The design of distributed systems is encouraging research in the fundamental area of program structures for programming such systems. Some proposals are beginning to be recognized as important contributions. The ARGUS approach provides the guardian concept and atomicity of operation [LISK-84]. Several authors have also studied the Remote Procedure Call (RPC) concept as a mechanism ensuring well structured and efficient communications [BIRR-83, SHRI-82]. More recently, a proposal has been made to consider "process group" as structuring facility for distributed computations [CHER-85].

The present section describes a program structure currently investigated in the GOTHIC project. This structure generalizes the concept of procedure as it allows for concurrent execution of several sequential programs which may possibly be nested in a very general fashion.Multi-functions provide the basic structure allowing a proper description of nested activities as presented in section 2.

3.1.1. The concept of multi-function.

A well-known structuring concept for classical operating systems and even for distributed operating systems is the procedure or function. The procedure is an abstraction of the notion of block with strict rules for communication with the environment (parameter and result passing mechanisms). Furthermore, procedures offer the possibility of nested computation through "recursive" calls.

Our purpose was to discover a somewhat similar concept allowing for:

-simultaneous processing of different components,

-parameter/result communication mechanisms,

-general nesting facilities.

a)-Block and parallel clauses.

A block may be represented as (D;F) where D stands for declarations and F a sequence of instructions.

Given two blocks, B_1: $(D_1; F_{11}; F_{12})$ and B_2: $(D_2; F_2)$,

B_2 "nested within" B1 may be represented as:

(1) $(D_1; F_{11}; (D_2; F_2); F_{12})$, with the following properties:

D_1 and F_{11} are the first executed, then block B_1 is interrupted. D_2 and F_2 are executed and B_1 is resumed thus allowing the execution of F_{12}; Visibility rules are such that F_2 "sees" D_2 and D_1, F_{11} and F_{12} "see" D_1.

A parallel clause may be described as:

$((D_1; F_1)//(D_2; F_2)// \ldots //(D_n; F_n))$

Components $(D_i; F_i)$ may be run in parallel. Several languages allow the following structure:

(2) $(D_0; F_0; ((D_1; F_1) // \ldots // (D_n; F_n)); F_{n+1})$

where D_0 and F_0 are first executed, then $(D_i; F_i)$'s are executed and finally F_{n+1} is executed. Visibility rules are such that F_{n+1} "sees" D_0.

Procedural nesting (1) is qualified of 1-1 nesting (one caller, one callee), and parallel clause nesting is qualified of 1-p nesting (one caller, p callees). This last form of nesting is the one generally found in distributed systems where such concepts as nested actions or nested activities are implemented [LISK-84, MUEL-83, MOSS-81]. Let us now describe a more general form of nesting as introduced in GOTHIC.

b)-A general form of nesting.

The nesting of a parallel clause within another parallel clause may be visualized as follows:

cobegin

$(A_{11}/$		$/A_{12})$
$(A_{21}/$	(B_1)	$/A_{22})$
.	(B_2)	.
.	.	.
.	(B_p)	.
$(A_{n1}/$		$/A_{n2})$

coend

Where the parallel clause $((B_1)// \ldots //(B_p))$ is nested within the parallel clause $((A_{11}/A_{12})//\ldots//(A_{n1}/A_{n2}))$. The execution of this structure can be described as follows:

A_{i1}'s sequences of instructions are initiated, and when they have <u>all</u> reached their "/", B_j's are executed. Upon termination of <u>all</u> B_j's A_{i2}'s are resumed with the property that A_{i2} may access the context defined in A_{i1}.

This form of nesting is the most general (n-callers, p-callees) and one can realize that 1-1 and 1-p nesting are particular cases of this n-p nesting [BANA-80]. After this informal description of nested parallel clause, let us introduce their logical properties.

c)-Logical properties of nested parallel clauses.

These logical properties are expressed with a formalism introduced in [LAMP-84] and known as Generalized Hoare Logic (GHL).

In order to describe control information, GHL uses the following predicates:

at(π)≡"control resides at entry point of program fragment π".
in(π)≡"control resides inside π".
after(π)≡"control resides at a point following immediately π".

Let us first describe the logical properties of nested blocks. Consider the fragment program π defined as:

π: (π1 : α;
  π2:β;
  π3:γ
)

If α, β and γ are characterized by their pre and post conditions as follows: $\{P\}\alpha\{Q\}$, $\{R\}\beta\{s\}$ and $\{T\}\gamma\{U\}$. For simplicity sake, α, β and γ are considered as atomic (or indivisible), so ($at(\alpha)\equiv in(\alpha)$, $at(\beta)\equiv in(\beta)$, $at(\gamma)\equiv in(\gamma)$). Block nesting control properties may be described as follows in terms of predicate at and after:

(I1) $at(\pi_1)\Rightarrow P$ (I_2) after $(\pi_1)\Rightarrow Q$
(I3) $at(\pi_2)\Rightarrow R$ (I_4) after $(\pi_2)\Rightarrow S$
(I5) $at(\pi_3)\Rightarrow T$ (I_6) after $(\pi_3)\Rightarrow U$

The invariant which characterizes the behavior of π is $I\equiv(\Lambda^6_{i=1}I_i)$, where I is the pre-condition of π, (i.e., $at(\pi)\Rightarrow I$). Furthermore, the following relationships are true:

(1) $at(\pi) \Rightarrow at(\pi_1)$
(2) $after(\pi_1) \Rightarrow at(\pi_2)$
(3) $after(\pi_2) \Rightarrow at(\pi_3)$
(4) $after(\pi) \Rightarrow after(\pi_3)$

They are derived from the properties of the sequentiality (;) operator.

We can generalize these properties in order to deal with nested parallel clauses:

π: **cobegin**
 (π_{11}:A_{11}/ /π_{12}:A_{12})

$(\pi_{21}:A_{21}/$ $\quad$ $\pi'_1:(B_1)$ $\quad$ $/\pi_{22}:A_{22})$

... $\quad$ $\pi'_2:(B_2)$ $\quad$ ...

...

$\pi'_k:(B_k)$

$(\pi_{n1}:A_{n1}/$ $\quad$ $/\pi_{n2}:A_{n2})$

coend

As above, A_{ij},$(i\in[1,n], j\in[1,2])$ and B_j, $j\in[1,k]$, are considered as atomic.

If A_{ij}'s are characterized in Hoare's notation, by $\{P_{ij}\}A_{ij}\{Q_{ij}\}$ and B_j's by $\{R_j\}B_j\{S_j\}$, then we have the following relations in terms of predicates at and after:

(I1) $\Lambda^n_{i=1}at(\pi_{i1})\Rightarrow\Lambda^n_{i=1}P_{i1}$ $\quad$ (I2) $\Lambda^n_{i=1}after(\pi_{i1})\Rightarrow\Lambda ni=1Q_{i1}$

(I3) $\Lambda^k_{i=1}at(\pi'_i)\Rightarrow\Lambda^k_{i=1}R_i$ $\quad$ (I3) $\Lambda^k_{i=1}after(\pi'_i)\Rightarrow\Lambda^k_{i=1}S_i$

(I4) $\Lambda^n_{i=1}(\pi_{i2})\Rightarrow\Lambda^n_{i=1}P_{i2}$ $\quad$ (I6) $\Lambda^n_{i=1}after(\pi_{i2})\Rightarrow\Lambda^n_{i=1}Q_{i2}$

The invariant which characterizes the behaviour of π is $\mathbf{I}\equiv(\Lambda^6_{i=1}\mathbf{I}_i)$, where **I** is the precondition of π. The following hold:

(1') $at(\pi)\Rightarrow\Lambda^n_{j=1}at(\pi_{j1})$

(2') $\Lambda^n_{i=1}after(\pi_{i1})\Rightarrow\Lambda^k_{j=1}at(\pi'_j)$

(3') $\Lambda^k_{j=1}after(B_j)\Rightarrow\Lambda^n_{i=1}at(\pi_{i2})$

(4') $after(\pi)\equiv\Lambda^n_{i=1}after(\pi_{i2})$

These formulae describe the synchronization rules governing the nesting of parallel clauses. Used in conjunction with GHL, they would allow to prove properties of programs involving nested parallel clauses.

3.1.2. General form of multi-functions

In the same way as procedure are abstraction of blocks we can define a computational model, the **multi-function**, which may be seen as the abstraction of the parallel clause. It is possible to call a multi-function from a procedure but also from another multi-function, thus providing a general form of nesting.

The description of multi-functions can be seen as a generalization of PASCAL functions.

For example, here is the definition of a multi-function called "mf":

muti-function mf;

```
(x, y, z:integerl): (u, v, w:integer);

var
 <declarations>

cobegin
    (x,y)u: begin    ...   return u end //        (1)
    (z)v: begin      ...   return v end //        (2)
    (y,z)w: begin    ...   return w end           (3)
coend;
```

This multi-function is made out of three components. Component (1) deals with input parameters (x,y) and delivers the output u, component (2) deals with input parameter z and delivers v and finally (3) deals with input parameters (y,z) and delivers w. Let us describe the usual multi-fonction call (or 1-p call).

a) 1-p multi-function call.

The statement (l,m,n) := mf(a,b,c) describes a statement where input parameters are (a,b,c) and the final result of the call will be assigned to variables l, m, n. The execution of this multi-function call may be depicted as follows:

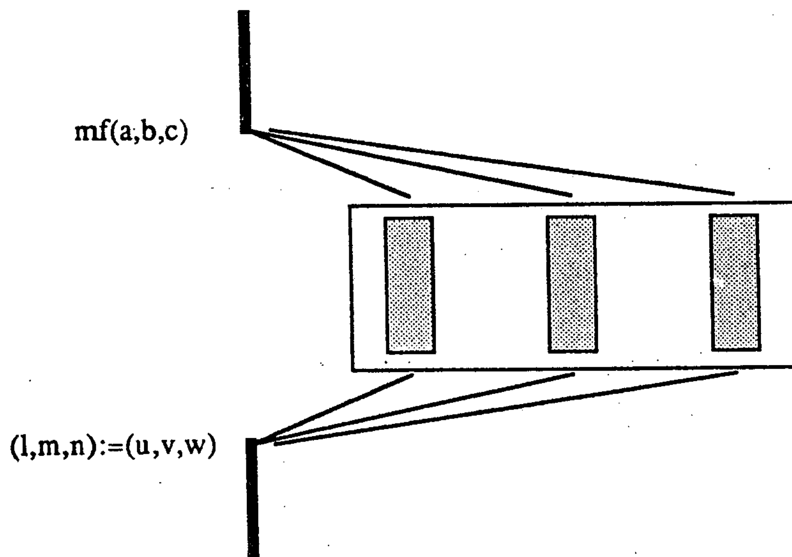

Figure 1: 1-3 call

Where the call mf(a,b,c) results in:

-distribution of input parameters to components of the instance of the multi-function mf created according to the call,

-parallel execution of the components,

-synchronization for result construction and transmission,

-resumption of the caller.

This execution scheme can be seen as a generalization of the usual procedure call (1-1 to 1-3).

b) Coordinated multi-function call (or n-p call).

Let us now introduce the most general multi-function call, the coordinated call (or n-p call).

Assuming the multi-function mf previously defined, consider the following program skeleton:

```
    cobegin
(1)     (integer a, k, l; ... ; (k,l):=mf (x<-a).(u,v); ...  ) //
(2)     (integer b, c, m; ... ; m:=mf(y<-b,z<-c).v; ...   ) //
(3)     (integer n;  ... ; n:=mf().w;  ... )
    coend
```

The execution of the call to mf can pictured as follows:

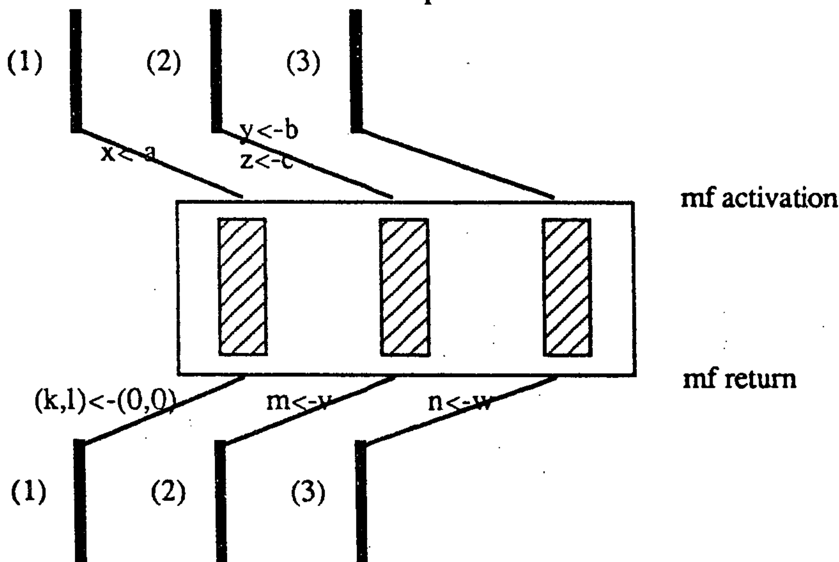

Figure 2: 3-3 call

where this coordinated call results in:

-coordination of components (1), (2) and (3) of the calling parallel clause for input parameter transmission. Notice that component (3) gets coordinated without providing any input parameter,

-distribution of the input parameters to the components,

-parallel execution of the components,
-synchronization of the components for result construction and transmission,
-distribution of the result to the components of the calling parallel clause. For example, notation n:=mf().w means that this component is involved in the coordinated call without providing any input parameter, but, after the processing of the call receives the w part of the result which is assigned to n.

3.1.3. Applications of multi-functions.

In this part we describe some potential uses of multi-functions.

a)-Generalized rendez-vous.

The use of nested multi-functions provide a possible mean to implement communications between components of a multi-function. as illustrated by the following example:

```
begin

    multi-function rdv;
        (input: integer): (output: integer);
        cobegin
            (input): output:
                begin
                output:= input
                end
        coend; #rdv#

    multi-function com;
        cobegin
(1)         begin
            ...
            rdv(input<-9);
            ...
            end //

(2)         begin
            y:integer;
            ...
            y:=rdv.output;
            ...
            end //
        coend #com#;

    ...
    #com invocation#
    com
```

...

end.

Component (1) of the multi-function com sends the value 9 to the component (2) by calling the muti-function rdv. This nested call implements a generalized form of Rendez-vous [HOAR-78].

One can remark that this use of nested multi-functions may involves too many coordinated calls and, because of the over-synchronization of the execution of components, may become inefficient. Moreover, the logical structure of the multi-function tends to be destroyed. This reason led us to propose that components of multi-functions could communicate through a common shared context, which is described in the header of the multi-function declaration. We do not describe this feature in more details here.

b)-Cooperation between groups of processes.

As an example, imagine the situation where a common agreement must be reached by two groups of people, G1 and G2. The following communication scheme may be taken:

Groups G1 has a meeting and makes a proposal which is sent to group G2 for approval or modification, eventually G1 takes the final decision. Notice that this is a group to group communication and that a member of a group cannot be distinguished.

If we model the behaviour of a group by the execution of a multi-function where a component represents a member of the group, control flow governing the above decision taking algorithm may be pictured as follows (fig. 3).

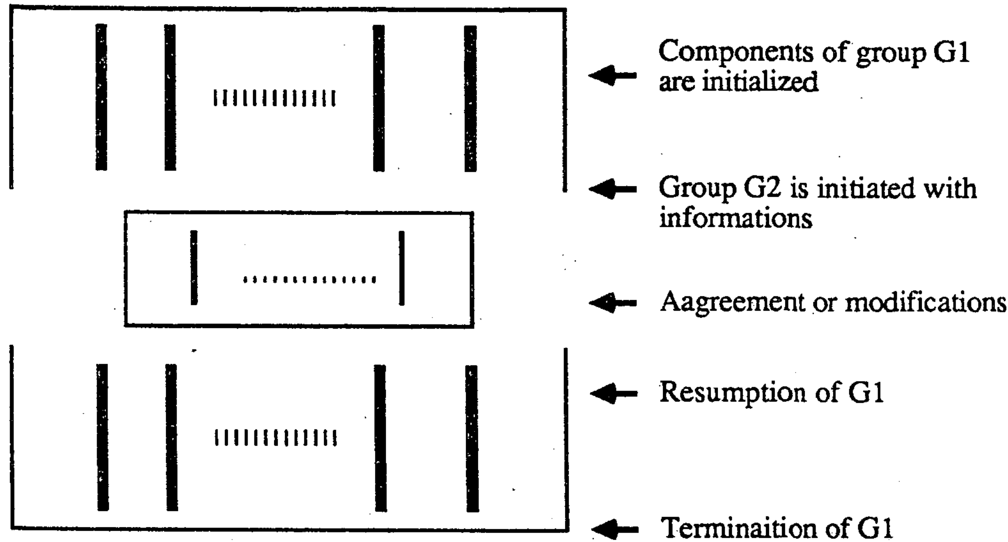

Figure 3: Cooperation between G1 and G2.

Of course, G2 is nested within G1. One can imagine that multi-functions provide a highly structured tool for describing this type of control. In this view, multi-functions may be seen as the abstraction of V-kernel process group [CHER-85].

c)-Global virtual memory.

Let us consider the problem of implementing a global virtual memory over a network as described in [LEAC-83]. In the most general case, an object is made out of a number of pages which may be located on different nodes.

Imagine that we want to allocate a number of pages (say pages) for a new object O. This will be achieve in three steps:

(i) A multi-function ("allocate-pages") is called. A component of allocate-pages runs on each node S_i of the network and determines for each S_i the number (ap_Si) of pages potentially available on node S_i.

(ii) The different numbers ap_Si are determined by a coordinated call to another multi-function "available-pages". Available-pages receives as input the number of pages request for the object, the number (fp_Si) of free pages of each Si and determines for each site the number of pages which can be allocated to the object.

(iii) The execution of the multi-function allocate-pages is resumed and actual allocation takes place on each node by updating local data structures (page lists).

Figure 4 visualizes control and information transfers occuring during this distributed page allocation algorithm:

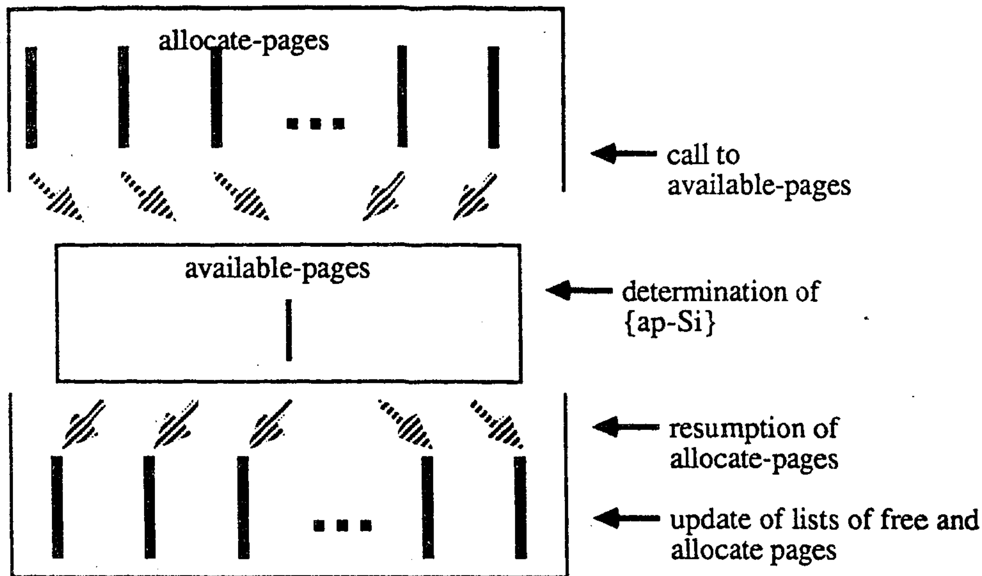

Figure 4: pages-allocation

Now we give a detail program of the global virtual memory example.

We assume the availability of the type **list** with three operations:

add:**list**⊕**list**→**list**, which concatenates two lists.

remove:¯N⊕**list**→**list**, which removes the n first elements of a list and creates a new list with these elements.

card:**list**→ N, which delivers the cardinality of a list.

Let us define the two following types:

type site_memory: **record**(allocated-pages, free-pages:**list**)

type obj-repr: **record**(S1_pages, S2_pages: **list**)

Actually, we restrict our problem where only two sites are available. Dealing with the problem in its full generality would lead us to manage list of sites or multisets of sites, thus making the solution less clear. So pages allocated for the representation of an object belong either to S1 or to S2.

The following program represents a possible solution:

```
begin
...
S1, S2 : site_memory;
Obj : obj_repr; #lists of pages#
...
multi-function available_pages;
      (nb_pages, fp_S1, fp_S2 : integer): (ap_S1, ap_S2: integer);
```

```
    cobegin
    (nb_pages,fp_S1,fp_S2):(ap_S1,ap_S2):
        var np: integer;

        begin
        if fp_S1≥nb_pages
            then
            ap_S1:=nb_pages;
            ap_S2:=0
            else
            np:=nb_pages-fp_S1;
            if np>fp_S2
                then
                #error#
                ap_S1:=ap_S2:=0;
                else
                ap_S1:=fp_S1;
                ap_S2:=np
            fi
        fi;
        return (ap_S1,ap_S2)
        end

    coend #available_pages#

multi-function allocate_pages;
    (pages: integer): O: obj_repr;

    cobegin
    (pages):(O.S1_pages):
        var v_ap_S1: integer;
        begin
        v_ap_S1:=available_pages(nb_pages<-pages,
                                 fp_S1<-card(S1.free_pages)).ap_S1;

        if v_ap_S1≠0
            then
            O.S1_pages=remove(v_ap_S1,S1.free_pages);
            S1.allocated_pages:=
                append(S1.allocated_pages,O.S1_pages)
            else
            O.S1_pages:= nil;
        fi;
        return O.S1_pages
        end //

    (pages):(O.S2_pages):
        var v_ap_S2: integer;
        begin
        v_ap_S2:=available_pages(nb_pages<-pages,
                                 fp_S2<-card(S2.free_pages)).ap_S2;

        if v_ap_S2≠0
```

```
            then
            O.S2_pages=remove(v_ap_S2,S2.free_pages);
            S2.allocated_pages:=
                  append(S2.allocated_pages,O.S2_pages)
            else
            O.S2_pages:= nil;
        fi;
        return O.S2_pages
        end

    coend #allocate-pages#

...
Obj:=allocate_pages(pages<-10).O;
...

end.
```

From these examples, it is clear that multi-functions are well suited to program distributed problems. Generally, the treatment of such problems involves three steps:

i)-Invocation of a multi-function each component of which is dedicated to the processing of a particular data fragment. In the last example, the set of pages can be considered as a fragmented data distributed over the set of sites. Each search for free pages is performed by a component of the multi-function.

ii)-common agreement between components. Of course, this phase can be expressed with a nested multi-function.

iii)-termination of the multi-function execution and delivery of the result.

Other distributed problems may be expressed in a very elegant fashion using this high level concept: updating a replicated object or maintaining a distributed naming server are examples.

3.3. Objects.

In GOTHIC, an object encapsulates some data and a set of operational facilities which are available to the user of the object. Data structures used in GOTHIC may be either localized on one node of the network (centralized object), split up into several parts (fragmented object) or possibly replicated (several copies of a given object exist simultaneously on different sites). Operational facilities are provided through the concept of multi-function which allows a straighforward management of fragmented and replicated data, as one can imagine from the example given in §3.1.3-(c), where the virtual memory can be seen as an object whose representation is fragmented into pages.

4-The GOTHIC architecture.

The integrated environment seen by GOTHIC users is a collection of software sub-systems operating on the set of node machines. Each machine supports the GOTHIC kernel which provides the set of primitives to implement objects and multi-fonctions management. Relation between these logical levels are shown in figure 5.

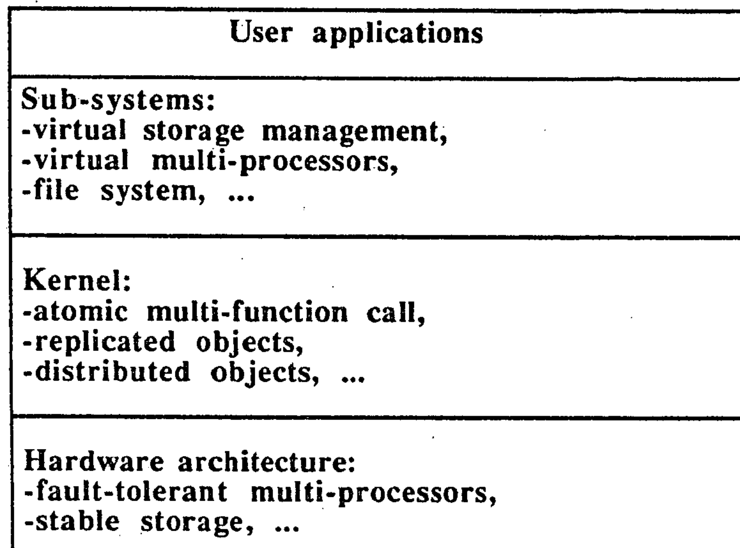

Figure 5: GOTHIC architecture

In the following we examine more closely the kernel and the hardware architecture

4.1. The kernel.

This level provides the following services:

1)-Implementation of replicated and fragmented objects.

2)-Implementation of multi-functions, which implies finding a solution to the following problems:

- -initialization of a multi-function call, i.e., establishment of a new "multi" context and input parameters transmission.
- -termination of the multi-function execution. Some distributed agreement is currently under consideration.
- -resumption of the callers.

3)-Remote multi-function call: a multi-function may run totally or partially on a set of

nodes distinct from the caller set nodes. Here too, we have to deal with a generalization of RPC protocols.

4)-Implementation of atomic multi-function invocation. This atomicity ensures that:

i)-either appropriate parameters are transmitted to every component of the multi-function or a failure is reported to the caller.

ii)-in case of failure of a component of the callee, a failure is reported to the caller and the effects of this tentative call are undone.

iii)-all the results, elaborated independantly by the components of the callee should be transmitted to the caller. This means that in case of failure during result transmission, this operation is re-attempted till it is successful.

The implementation of points (i) and (ii) makes necessary the design of protocols for distributed agreement. These aspects are currently under investigation.

However, it is important to provide an performant atomic multi-function call, this makes necessary the availability of a stable and efficient storage facility. This led us to design such a hardware mechanism which is presented in the following paragraph.

4.2.Hardware architecture.

In most respects, GOTHIC hardware architecture resembles that of existing networks of personal computers. Each user has a node machine consisting of one multi-processor and a bit-mapped graphics sub-system. Node machines are interconnected by a local network.

In this part, we describe the architecture of the SM90 [FING-82] that we have selected as node machine, then we present the stable memory board that we have built to ensure a performant execution of atomic multi-functions.

4.2.1. SM90 architecture.

The hardware-level architecture of SM90 is shown on the following figure.

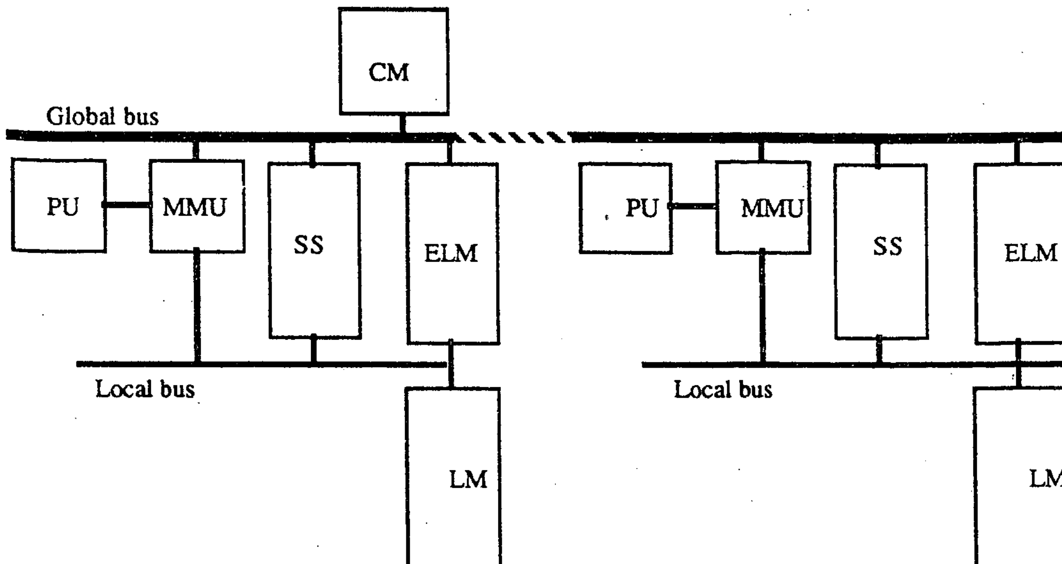

Figure 6: SM90 architecture

The SM90 architecture supports until 8 processing unit (PU), 16 exchange units (EU) and 16 megabytes of memory. The machine is organized around a global bus on which are plugged the different boards. Moreover, every PU has its own local bus.

Every memory module can be plugged either on the global bus or on a local bus. In the first case this common memory (CM) is shareable by the processors of different PU's. In the second case, the local memory (LM) can be accessed only by the processor of the PU. It is also possible to plug a memory board both on the global bus and on a local one; in this case this external local memory (ELM) can be accesses through the two busses.

Every PU is equiped with a microprocessor (Motorola 68010), with a small amount of private memory and with a memory management unit (MMU). The MMU is a set of 1024 registers each containing a segment descriptor (physical address, lengh, access right, etc.). Segment described by such a descriptor can be located in any memory module connected either to the local bus of the pro cessing unit or to the global bus.

The EUs are intelligent I/O modules, each one is made of a microprocessor, some memory and device controlers.

Every PU will be enriched by a stable storage board (SS) plugged both on the global bus and on a local one in order to build a fault -tolerant machine. Let us describe briefly its basic structure, a complete description can be find in [BANA-86b].

4.2.2. Structure of the stable storage memory.

We had three goals in mind during the design phase of the stable storage memory of the GOTHIC system:

i)-to build a large stable storage,

ii)-to provide short access time (equivalent to a ordinary RAM memory),

iii)-to provide object management implemented in hardware.

Disks seem to be desirable devices according to point (i) but their access time is not acceptable (point (ii)). We choose the approach of the building of a stable storage only from non-volatile RAM memory.

Before describing the hardware mechanisms, let's define more precisely our fault hypothesis:

i)-a memory device may decay over time. As a consequence, each object has two copies on two decay-independent banks.

ii)-In case of crashes, information stored in a memory bank might be corrupted, (bad addresses,...). Therefore, we have to ensure that all elements (words) belonging to the updated object, and only these elements, can be addressed by the processor.

Here below is a description of the hardware structure of the stable storage memory:

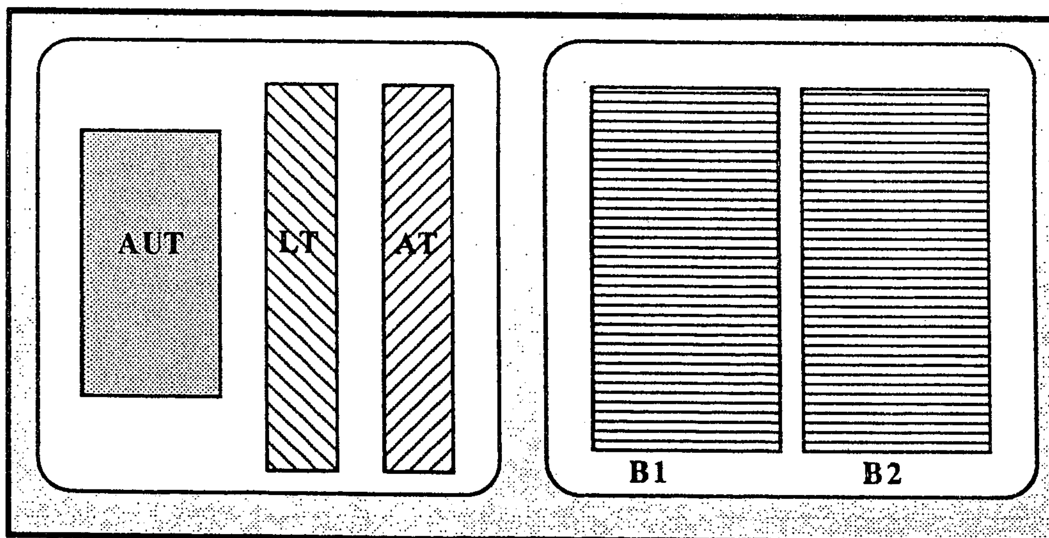

Figure 7: Stable storage structure

On this figure,

-B1 and B2 are coupled memory banks, each of which contains 512K bytes.

-AUT is the hardware automaton which is responsible for the access control on B1 and B2.

-LT is a link table which contains logical links between the different elements of each object located in (B1,B2) as explained later.

-AT is an access table which characterises the elements of the current object Obj under modification. AT[i] is set if i^{th} element of the current object is being modified.

The two tables LT and AT are only accessed by the automaton AUT, they are not visible from the processor.

When an object is created in stable storage the automaton is put in the initial state "object creation" and memory is allocated on B1 and B2. The LT table is updated such that, k and k+1 being two consecutive elements of the object located at real adresses p and q respectively, LT[q] contains a logical link to LT[p].

The object update operation follows four phases:

i)-The automaton is put in the initial state "object update".

ii)-the object is read and the two copies are compared in order to detect any eventual decay. The access table is set, so as to enforce that the only words of stable storage that can be modified by the processor are those marked in the access table.

iii)-The object is written on the first bank element by element. When the jth element of the object is to be written accessed, it is checked that the previous written element was the $j\text{-}1^{th}$ one. This mechanism implies that, by the end of the update operation, it has been controlled that the object has been fully written on the first bank.

iv)-The object is then written from the first to the second bank, element by element with the same checks as described in step (iii).

Recovery schemes are provided in case of errors during the processing of these phases.

As far as performances are concerned, a first estimation indicates that the time needed to update a stable object is three times more than the one needed to update a standard object in RAM memory.

5-Summary.

This paper describes the main characteristics of the GOTHIC distributed system. Let us emphasize the most important aspects of this system:

-replicated and fragmented data,

-multi-fonctions: A generalization of the concept of procedure has been presented. This generalization orthogonalizes procedures in two respects:

(i)-several concurrent "bodies" may execute concurrently,

(ii)-nesting is extended to become a basic synchronization (and communication) facility.

The formal properties of multi-functions have been described and programming examples have been developed.

-efficient stable storage for implementation of atomic multi-function call.

These basic facilities constitute the kernel of the GOTHIC system. The level above the kernel contains "user-oriented" sub-systems such as the GOTHIC memory management, file system or virtual multi-processors.

The present status, (january 86), of the project is the following:

-the design of the hardware and software of the system is completed.

-the development of a stable memory board is currently under way.

-the implementation of atomic multi-functions is undertaken.

-the design of the virtual storage sub-system is well under way [JEGA-85].

Other topics such as other sub-systems are issues presently debated and discussed.

References:

[BANA-80] BANATRE J.P.
Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables.
Thèse d'Etat, Université de Rennes 1, déc. 1980.

[BANA-83] BANATRE J.P., BANATRE M., PLOYETTE F.
Construction of a Distributed System Supporting Atomic Transactions.
Proc. of 3nd Symp. on RDSDS, Clearwater-Beach, Oct. 1983, pp. 95-99.

[BANA-84] BANATRE M.
Le Système ENCHERE: une Expérience dans la Conception et la Réalisation d'un Système Réparti.
Thèse d'Etat, Université de Rennes 1, Mars 1984.

[BANA-86a] BANATRE J.P., BANATRE M., LAPALME G., PLOYETTE Fl.
The Design and Building of ENCHERE, a Distributed Electronic Marketing System.
Communications of the ACM, Vol 29, n°1, January 1986. pp. 19-29.

[BANA-86b] BANATRE M., MULLER G.
Conception et réalisation d'une mémoire stable rapide pour la gestion d'objets structurés.
Rapport de Recherche INRIA (to appear).

[BANA-86c] BANATRE J.P., BANATRE M., PLOYETTE Fl.
The Concept of Multi-fonction: a General Structuring Tool for Distributed Operating System.
Proc. of 6th Distributed Computing Systems, Cambridge, Mass., May 1986.(to appear)

[BIRR-84] BIRRELL A., NELSON B.
Implementing Remote Procedure Calls.
ACM TOCS, Vol 2, n°1, Feb. 1984, pp. 39-59.

[CHER-85] CHERITON D.P., ZWAENEPOEL W.
Distributed Process Group in the V Kernel.
ACM TOCS, Vol. 3, N°2, May 1985, pp. 77-107.

[FING-82] FINGER U., MEDIGUE G.
Architectures multiprocesseurs: l'exemple de la SM90.
Minis-Micros, n°173, 1982, pp. 65-69.

[HOAR-78] HOARE C.A.R.
Communicating Sequential Processes.
Com. ACM 21,8, Aug.1978, pp.666-677.

[JEGA-85] JEGADO M.
Proposal for the Design of a Stable Object Manager.
Proc. of the second Newcastle-Rennes workshop on Distributed Computing.
Linden-Hall (G.B.), march 1985.

[LAMP-84] LAMPORT L., SCHNEIDER F.
Formal Foundation for Specification and Verification.
LNCS 190, 1984, pp. 203-270.

[LEAC-83] LEACH P.J., LEVINE P.H. DOUROS B.P., HAMILTON J.A., NELSON D.L., STUMPF B.L.
The architecture of an Integrated Local Network.
IEEE Journal on selected areas in comm.,Nov. 1983, pp.842-856.

[LISK-84] LISKOV B.
The Argus Language and System.
LNCS 190,1984, pp. 343-430.

[LYON-84] LYON B., SAGER G.
Overview of the Sun Network File System.
Sun Microsystem, Inc., Oct.1984.

[MOSS-81] MOSS J.E.B.
Nested Transactions: an Approach to Reliable Distributed Computing.
MIT/LCS/TR-260, M.I.T. LCS, Cambridge, Ma., 1981.

[MUEL-83] MUELLER E., MOORE J., POPEK G.
A Nested Transaction System for LOCUS.
Proc of 9th SOSP, Bretton Woods, N.H., Oct. 10-13.

[SHRI-82] SHRIVASTAVA S., PANZIERI F.
The Design of a Reliable Remote Procedure Call Mechanism.
IEEE Trans. on Computer, vol C-31, n°37, July 1982, pp. 692-697

PI 278 **Controlling knowledge transfers in distributed algorithms — Application to deadlock detection**
Jean-Michel Hélary, Aomar Maddi, Michel Raynal — 32 pages ; Janvier 86.

PI 279 **Une méthode de conception de programmes fonctionnels**
Raymond Durand, Martine Vergne — 16 pages ; Janvier 86.

PI 280 **Commande de systèmes redondants et évitement d'obstacles**
Bernard Espiau — 52 pages ; Janvier 86.

PI 281 **A distributed algorithm for mutual exclusion in an arbitrary network**
Jean-Michel Hélary, Noël Plouzeau, Michel Raynal — 16 pages ; Janvier 86.

PI 282 **Stabilité robuste dans la commande adaptative indirecte passive**
Philippe de Larminat — 70 pages ; Janvier 86.

PI 283 **Data synchronized pipeline architecture pipelining in multiprocessor environments**
Yvon Jégou, André Seznec — 36 pages ; Janvier 86.

PI 284 **An overview of the GOTHIC Distributed Operating System**
Jean-Pierre Banatre, Michel Banatre, Florimond Ployette — 24 pages ; Janvier 86.

Jean-Pierre BANATRE

Michel BANATRE

Florimond PLOYETTE

AN OVERVIEW OF THE GOTHIC DISTRIBUTED OPERATING SYSTEM

Publication interne
n° 284

Janvier 1986