A Systematic Scheme to build Distributed Algorithms

Gérard FLORIN [1] , Roberto GÓMEZ[2] , Ivan LAVALLÉE[2]

Key-words: *distributed algorithms, wave, recursivity, remote procedure call, shortest path.*

Cedric [1]

CNAM

292 rue St Martin 75130 Paris Cedex 03

e-mail : gerard@cnam.cnam.fr

Tél. 40.27.20.64.

♣

Action PARADIS [2]

INRIA

Domaine de Voluceau, Rocquencourt BP 105

78153 LE CHESNAY Cedex, FRANCE

e-mail : lavallee@nuri.inria.fr

e-mail : rgomez@nuri.inria.fr

Tél. 39.63.54.80, 39.63.52.38

Abstract

Designers of distributed algorithms must deal with a variety of issues including sequential algorithms design, communication protocols, fault tolerance. The distributed design must include also a proof step of the whole algorithm features. This paper gives a new schema for the systematic design of distributed algorithms. Our distributed design method is mainly based upon parallel recursive schemas but recursivity is used in a distributed environment. Therefore we use two existing and widely available tools: the remote procedure call, and the PAR instruction (parallel execution of threads). In this approach we propose to perform the design step simultaneously with the proof step. We show a shortest path algorithm as an exemple of this method.

Introduction

Designers of distributed algorithms must bring matter to several issues. The questions at issue include the classical problems of sequential algorithm design (for each sequential process running on a given processor). It includes mainly complex synchronization features to be specified and implemented (e.g., designers must specify communication protocols managing the cooperation between the sequential processes).

Many solutions have been proposed to solve the previous problem. Lamport, Shostak and Pease's solution of the byzantine general problem [LSG82] is, as far as we know, the first example of a recursive programming of a distributed algorithm. Another solution is associated with the wave notion introduced in ([Cha90], [Fre89]). It consists in a graph traversal technique based on a flux and reflux of messages. [FlLa91] and [FGL93] propose a schema that can be used for the construction of algorithms based on distributed recursivity and on the remote procedure call. Therefore this building schema is also related to distributed object design.

In sequential programming the recursive design of algorithms is elegant but not very efficient, therefore most of the designers do not use recursive solutions. On the contrary recursive design in distributed algorithms is both elegant and efficient. In fact, a recursive descending allows to visit a set of sites according to a visiting strategy. On the other hand the recursive backtracking allows to collect results from the visited sites as in the wave schema.

Also, the systematic use of the remote procedure call, (RPC) provides a high level communication tool. It avoids to deal with the difficulties dues to the management of message passing.

The main interest of this approach is to offer a framework for the design of distributed algorithms. On the other hand a serious problem in distributed programming is to prove the correctness of an implementation. As it is often difficult to prove a distributed algorithm, we propose to code the proof of a distributed algorithm. This paper also shows how to generate a distributed program from the proof of an algorithm.

The paper is organized as follows. The first section recalls the recursive distributed idea. The third and fourth section explain the different steps of the formal proof. The last sections are dedicated to an example, namely the shortest path algorithm.

1 Recursive waves

A recursive wave is first a procedure. This procedure, during its execution, calls n (or none) concurrent remote executions of the same procedure. Therefore executing a recursive wave leads to a tree such that:

- the root represents the first execution of the procedure. It is the initiator of the computation;
- the root and the nodes represent blocked procedures. They are waiting for the termination of the parallel recursive calls;
- the leaves represent the active execution of the procedure;
- the tree is oriented by the relation between the calling procedure and the recipient procedure.

Sometimes a recursive wave uses a set of auxiliary procedures. These procedures encapsulate a data structure, as for example in an ADA package (but also as in the modular, or as in the object oriented approach). The data structure includes, for example, the current processor identifier, a set of neighbour's identifier, etc.

In this paper we assume that all communications and synchronizations between the successive remote executions are defined by the remote procedure call schema. There are no complementary data or messages exchanged between the remote executions on different processors.

Sometimes, there can be a need for communications between the different executions of the same procedure on a given processor. The procedures use shared variables to do this kind of communication.

2 Specification of the general schema.

In this section we introduce the features used to develop the proof and the construction of distributed recursive algorithms.

2.1 Programming schema

We consider concurrent executions of n procedures. Therefore a programming control structure for the parallel execution of n threads on the same processor must be available. We assume that the following instruction **par** , similar to OCCAM, can be used.

```
par i  in  <domain> do
    < INSTRUCTION BLOCK >;
enddo ;
```

For each parameter value in a discrete set, (<domain>), a thread associated with the instruction block is started on the processor executing the **par** instruction. This instruction is terminated when all the activated threads have terminated and the next instruction is executed.

We must define the location of remote procedure executions. So we consider the following syntax for a RPC execution of a given procedure, (named <pcd_name>) on a remote processor (named <prcs_id>).

```
<pcd_name> ( <parameter_list>) on  <prcs_id>;
```

In the figure 1 we present the general description of the recursive wave. The instruction blocks represent any kind of sequence. We define the schema in an ADA-like specification language. As we often use the notion of processor groups we have to manage sets. Therefore we consider that a package implementing a type set (setof) is available with its associated operations (union, intersection, ...).

2.2 Mathematical schema

The previous definition of the recursive wave induces that the recursive procedure, called R, can be defined as a set of four many to many functions:

$$R = \{ L , S , G , H \}$$

Function L

This function defines the set of local data to each site. In each site i, there can be several simultaneous executions of the recursive procedure. To each execution of the procedure is associated a data set. In a recursive environment the same data can take different values according to the execution level. Let DL_i^j be the local data set of the execution level j on the site i. This set DL_i^j is composed of two data subsets:

1. THE SITE DATA: the data local to i and global to the executions of the procedure; named DS_i

```
type processor_identifier is ...; - Type processor name
i: processor_identifier;

procedure recursive_wave ( {parameters}) is
< DECLARATION BLOCK >
begin
    < INSTRUCTION BLOCK A >
    if { condition } then  - Stop/exit of the recursive wave
        < INSTRUCTION BLOCK B>
        par  i in  <DOMAIN> do  - Execution of concurrent threads
            <INSTRUCTION BLOCK C >
            recursive_wave ({ parameters })  on  i; - RPC execution
            <INSTRUCTION BLOCK D  >
        enddo ;
        <INSTRUCTION BLOCK E  >
    endif ;
    <INSTRUCTION BLOCK F >
end  recursive_wave;
```

Figure 1: *General description of the recursive wave.*

2. THE EXECUTION DATA: the data proper to each execution: local to i and used only at the execution level j; named DE_i^j,

Therefore we can say:

$$DL_i^j = DS_i \bigcup \sum_{j=1}^{n} DE_i^j$$

where n represents the number of times that the same procedure has been executed at the site i.

During a procedure execution the values of the local data can be modified by the local code. L is the function that represents any of these modifications. It depends on two parameters. e first one represents the values of the local variables modified at the execution level k, DL_i^k. The second one represents the input parameters values of the procedure, called VP. Thus we define the L function like:

$$DL_i^j = L(DL_i^k, VP)$$

Function S

The par instruction generates a set of threads. The function S creates the set of indices of these threads, (i.e., the discrete set named <domain> in the general schema). The set is computed using the local values, DL_i, of the present active procedure; and the input parameters values, VP. Then we define the function S as:

$$\{<\text{DOMAIN}>\} = S(DL_i, VP)$$

Function G

The function gives the set of the RPC parameters, sent to the site k from the site i. This set is computed with the local data of i, the input parameter values, and the receiver site identification, k. So G is a function of three parameters:

$$\{\text{PARAMETER VALUES}\} = G(DL_i, VP, k)$$

Function H

It computes the return value (before the end of the call). This result value is to be sent to the calling procedure. This function represents the last step in the recursive wave. It is computed with the local values, the input parameters values, and the set of values collected from the son's sites, H_{sons}. Therefore H is defined as follow.

$$< \text{RETURN VALUE} > = H(DL_i, VP, H_{sons})$$

H_{sons} is a set, built from the results of the sons of site i.

If i is a leaf then H_{sons} is an empty set.

2.3 Formal description.

We give a formal description of the recursive wave. It is a combination of the previous functions and the general description. This formal description is presented in figure 2. The line numbers are used to better understand the general description of the algorithm's design, presented in the next section.

```
type processor_identifier is ...; - Type processor name
DS;  - Site local data set
i : processor_identifier;
procedure recursive_wave ( VP: parameters value, Hi : Result ) is
DE;  - Execution local data set
begin
    DL = A (VP, DL)  - Instruction Block A
    if { condition } then  - Stop of the recursive wave
        DL = B(VP, DL)  - Instruction Block B
        par  i in S(DL,VP) do  - S: Set of indices threads
            DL = C(VP, DL)  - Instruction Block C
            recursive_wave  (G(DL,VP,i) , Hi) on  i; - G: Parameters function
            DL = D(VP, DL)  - Instruction Block D
        enddo ;
        DL = E(VP, DL)  - Instruction Block E
    endif ;
    DL = F(VP, DL) - Instruction Block F
    R = H ( DL, VP, Hi) - H: Backtrack function
end  recursive_wave ;
```

Figure 2: *Formal description of the Recursive Wave.*

3 General Description of Algorithm's Design

One of our goals is the generation of the code from the proof of the algorithm. In this way we divide the description of an algorithm's design in five ordered steps. There is no way to help the designer in the initial part of his work. The designer is responsible of the main guidelines definition of its solution.

3.1 First step: Definition of the local values set

The first step is to define the local value set. This set supports the input, auxiliary, and output values of the algorithm. Examples of these values are the name of a site, the result of the distributed computation, or some intermediate values involved of the solution). This definition is concerned with the site data, DS_i; the execution data, DE_i^j; and the input parameters

These arguments lead to the construction of the declaration part, of the recursive wave code. Following the code presented in the figure 2 we can see that this step is concerned with the lines 2,3,4 and 5 of this code.

3.2 Second step: The visit strategy

The designer of a distributed algorithm, is concerned first with the choice of the visit strategy to the different sites. For example the visit can be performed according to a predefined spanning tree, [FlLa91]. This visit must be such that each path, without circuits, from a root site to any reachable site represents a path of the recursive tree. So the goal in this step is to show that the distributed algorithm realizes the designed visit strategy.

From the previous ideas we can see that it is necessary to:

- define a run way of the algorithm coherent with the goal of the calculus;
- show that the pattern of the recursive calls and the stop test induces the expected visit strategy;
- show that the termination condition is correct, i.e., the visit strategy doesn't generate an infinite tree.

We can see that these arguments are concerned with the S function, and the lines 8,9, and 10, of the code in figure 2.

3.3 Third step: The descending.

The recursive calls induces a network traversal that supports a distributed computation. This distributed computation is done in two steps; the first is called the *descending*, and the second *backtracking*.

The descending traversal of the network generates a tree. We explained how to calculate the identifier sites, to which the RPCs are to be sent. Also we need to know the parameters values of these RPCs.

Within each procedure execution on a vertex, we perform operations with local data and input parameters values of the procedure. After performing these operations a RPC is sent, with new parameters values, according to the traversal and the results obtained on local data. The last vertices reached by a RPC are the leaves of the recursive calls tree.

The proof of the *descending* step is concerned with the following arguments:

- the presentation of an assertion, and its demonstration for all the different levels of the tree;
- the criterion of correctness of the calculated parameters;

- the stopping criterion.

Thus, we obtain the proof of some properties in all visited vertices, and then particularly for the leaves of the tree. We can see that these properties are concerned with the lines 12, 13 and 14 of the formal description of the wave.

3.4 Forth step: The backtracking

The backtracking of the recursive wave is done from leaves to the root. It goes through the father, and the father of the father, and so on. This way induces a result collect of the recursive calls, and a local computation with the local data and the collected results.

The backtracking can begin once that the current process is a leaf of the recursive calls tree [1]. The proof of the properties of this backtracking is obtained according to the following arguments (induction proof):

- from the leaves it must be proved that a property is true on all leaves which have the same father;
- prove the correctness of the data collection. It goes from the vertex of depth p, to one of depth $p-1$, in the recursive call tree.

This step is related to the function H, (see lines 16,17,18 and 19 at figure 1).

3.5 Fifth step: Coherence of the local values

Every time a procedure is executed in a site, the local values of that site are modified. We must take care of these modifications, because they can change the result of the algorithm.

In the mathematical schema we defined DL_i^j as the set of local data of the execution j on the site i. Each time a procedure is executed on a site, the local data are modified, but the sequence of the different modifications is unknown. Then the different modifications must be equivalent to a serial execution (serial criterion). The selection of the best solution will depend on the problem nature (from simple mutual exclusion until ACID transactions can be defined).

4 Example: The shortest path problem

In this section we apply all the showed ideas in the previous section to the shortest path problem. We proceed with development of the proof, instead of finding a recursive solution to this problem, following the different steps introduced in the section 3.

[1] See section 2

4.1 Informal considerations

The problem consists in the computation of the shortest paths from a vertex r called root, (which is the initiator of the calculus), to all vertices of a valued graph. To build the shortest paths tree, eventually we want to obtain all single paths of the graph from the root to all others.

On the other hand we use two optimizations to avoid scanning unnecessary paths. Our first optimization avoids building elementary circuits, sending RPCs to the caller. The second optimization uses the Bellmann's principle: it computes the path length and then selects the shortest.

In contrast with many authors, we suppose that all costs are positives [2] and all edges are bidirectional. That is to say the valuation of (a, b) is not necessarily the same as the one of (b, a). This case is the most realistic, but perhaps more difficult to handle.

Thus at the start of the calculus, each vertex x knows its identifier, the identifier of its neighbours (i.e., $\Gamma(x)$), and the cost of all the arcs, $C(x, y)$. After the computation, each vertex knows its father in the shortest paths tree, the weight of the path from r to itself, and its sons in the shortest paths tree [3].

In view of the formal proof of the scanning, we consider that all the possible paths are potentially scanned, and particularly the shortest one. The elimination of longer paths doesn't avoid the construction of the shortest path. Consider that a path from the initiator to another vertex is longer, (i.e., with larger cost) than another path from the initiator to the same vertex. So it is not necessary to continue the construction of this path since it can't become the shortest path.

4.2 Notations in view of the formal algorithm proof.

We use the following notation in the proof:

$GR = (X, \Gamma, C)$	the weighted and directed graph.
$\Gamma(x)$	the set of neighbours of the site x.
r	the root.
$PC(x, y)$	the cost (weight) of path (x, y).
$PC_{min}(x, y)$	the minimal path cost from x to y.
$C(x, y)$	the cost (weight) of arc (x, y)
$Ch^{(j)} = \{x_o = r, x_{i_1}, x_{i_2}, ...x_j\}$	the path from the root to x_j.

The path weight value also can be expressed as follow:

[2] To consider negatives values is possible.

[3] It is possible to modify this algorithm to obtain more information at each vertex, and specially the root could know the entire shortest paths tree.

$PC(Ch^{(j)}) = PC(x_0, x_j)$ the path cost from the root to x_j.

5 First step of the proof: Definition of the local value set.

The first step is the definition of the local value set, like it was stated in section 3.1. So we divide the proof in differents steps.

5.1 Definition of the local data to each site, DS_i

In the shortest paths problem the local data set of the site i, DS_i, is composed of two subsets. The first one is a dynamic one named DSd_i. The second one is a static one named DSs_i.

We said that the data in DSd_i is dynamical because their values are modified every time a new site is involved. Then we define:

$DSd_i = \{mpc_i \ , \ son_i(y) \ , \ father_i \}$

The data in the static set, DSs_i, will not change at any moment of the execution. This is because the cost of the edges is the same during all the running time.

$DSs_i = \{y \, \epsilon \Gamma(i) \ , C(i,y)\}$

Thus the local data set at site i is:

$DS_i = DSd_i \bigcup DSs_i$

$\Gamma(i)$ and $C(i,y)$ have been defined, as for the rest:

mpc_i	current minimum cost of the path from the root, r, to the site i;
son_i	the sons of the site i;
$father_i$	the father of the site i.

5.2 Definition of the local data to each execution, DE_i^j

In 2.2.2 we define DE_i^j as the data proper to each execution. The local data in every site i, that change at every execution j are the sons of every site i. So we define:

$DE_i^j = \{ \ local\_sons_i \ \}$

where $local\_sons_i$ represents the sons of the site i at the execution j. They are all the neighbours of the site i, except its father [4]. So $local\_sons_i$ is computed as:

$DE_i^j = \Gamma(i) - callers\_list(i)$

where $callers\_list(i)$ are the callers arriving to the actual site i, (it includes the father of the site).

[4] In section 9 we discuss a coherence problem that can be presented.

5.3 Definition of the parameters, VP

We can see that the values needed in every site, at any execution, are the path cost and the callers list of a site i. So we define VP, the set of input values parameters, like:

$VP = \{\ pcost\ , callers\_list\ \}$

where *pcost* is the cost of the path and *caller_list* represents the caller's list.

5.4 Definition of the L function

In subsection 3.2 we defined the local data set like:

$$DL_i^j = DS_i \ \bigcup \sum_{j=1}^{n} DE_i^j$$

so the local data set is composed of the neighbour's identifiers of the site (*neighbours*) and their path cost (*cost*); the minimum path cost of the root to the site (*mpc*); the father and the sons of the site (*father, sons*); and the list of callers arriving to the site (*callers*) .

6 Second step: Formal proof of the visit strategy

In the shortest path problem the visit strategy proof, consist to know which nodes will be the receivers of the RPC calls. This involves the next considerations:

- compute of the set of nodes receiving the RPC, in view of building all the paths from the root to all the sites;
- avoid the construction of the circuits , and show that this action does not avoid to find the shortest path;
- set up the stopping condition for the recursive algorithm.

6.1 Recursive building of the paths of length l

The specification of the path building is given by:

$$\begin{array}{lll} l=0 & Ch^{(0)} = & \{x_0 = r\} \\ l=1 & Ch^{(1)} = & \{x_0, x_{i_1} \mid \\ & & \quad x_0 \in Ch^{(0)}, \ \text{and} \\ & & \quad \forall x_{i_1} \in \Gamma(x_0)\} \\ & \cdots & \end{array}$$

$$
\begin{aligned}
l = n \quad Ch^{(n)} = \quad & \{x_0, x_{i_1}, x_{i_2}, ..., x_{i_{n-1}}, x_{i_n} \mid \\
& x_0, x_{i_1}, ..., x_{i_{n-1}} \in Ch^{(n-1)}, \text{ and } \\
& \forall x_{i_n} \in \Gamma(x_{i_{n-1}})\}
\end{aligned}
$$

Property:

We use the previous schema to build all the paths of length n [5].

`Proof:`

If all paths of length $n-1$ are built, then all paths of length n are potentially built. This is because they are obtained by extension of all paths of length $n-1$ and by all the neighbours of its last edge.

Lemma:

According to the property, there is no path of length n omitted.

`Proof:`

Suppose that it is false. Then there is a path of length n that was not obtained as an extension of a path of length $n-1$. It is absurd because all path of length n contains one path of length $n-1$ and, by the way all path of length $n-1$ was previously built.

6.2 Avoidance of circuits construction

A characteristic of this algorithm is that it avoids the construction of elementary circuits. We will show how to do this, and the consequences of this peculiarity.

Property :

Avoiding the construction of circuits preserves the minimum cost paths[6].

`Proof :`

Consider the path described in the figure 3. We can see that this path contains a circuit, denoted by x_{i_k}, x_{i_l} and x_{i_m}. Let PC^{circt} be the cost of the path from x_{i_1} to x, including the circuit. This cost is given by:

$$
\begin{aligned}
PC^{\text{circt}} = PC(x_{i_1}, x_{i_2}) \; + \; \cdots \; + \; PC(x_{i_j}, x_{i_k}) \; + \; PC(x_{i_k}, x_{i_l}) \; + \; \cdots \\
\cdots \; + \; PC(x_{i_m}, x_{i_k}) \; + \; PC(x_{i_k}, x_{i_n}) \; + \; \cdots \; + \; PC(x_{i_p}, x)
\end{aligned}
$$

Let $PC^{\text{no-cirt}}$ be the cost of the path from x_{i_1} to x, without the circuit; it is computed as follows:

$$
PC^{\text{no-cirt}} = PC(x_{i_1}, x_{i_2}) \; + \; \cdots \; + \; PC(x_{i_j}, x_{i_k}) \; + \; PC(x_{i_k}, x_{i_n}) \; + \; \cdots
$$

[5] Remember that the number of edges defines the length of a path.

[6] It is the major property which authorize an extension of our algorithm in case of negatives' costs

$$\cdots + PC(x_{i_p}, x)$$

Then we can say that: $PC^{\text{circt}} > PC^{\text{no-cirt}}$

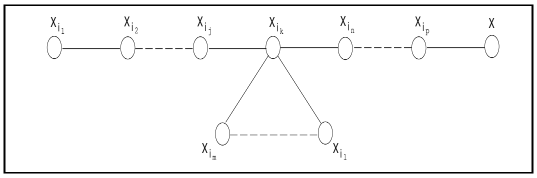

Figure 3: *Example of a path containing circuits.*

The circuits' construction can be avoided by sending no RPCs to the previous caller, [7] (i.e. the father in the execution tree).

Thus at step n we have :

$$\begin{aligned} l = n \quad Ch^{(n)} = \quad & \{x_o x_1 ... x_n \mid \\ & x_o ... x_{n-1} \,\epsilon\, Ch^{(n-1)}, \text{ and} \\ & x_n \in \Gamma(x_{n-1}), \text{ and} \\ & x_n \neq x_{n-2}\} \end{aligned}$$

6.3 Stopping condition

Our goal is to find the shortest path cost from the root to all sites. So we can take this as an argument to establish a condition that stops the RPC sending.

We start with the next property:

Property :

Let pc_1 the path cost of a path called $path1$, and pc_2 the path cost of another path called $path2$. If pc_1 is greater than pc_2 it is not necessary to continue with the $path1$

`Proof:`

The path obtained $path1$ cannot be better than the continuation of $path2$, then there is not necessary to continue with the first one.

CONSEQUENCE 1 :

It is not useful to continue the building of a path that will reach a site, in which the cost will be greater than a previously computed one.

CONSEQUENCE 2 :

If the path cost needed with the current RPC is not better that a previously computed one, then the wave is stopped and the RPC activation is terminated.

[7] This avoidance is a critical point of the algorithm, and it is related to the stopping condition.

Considering this, we can establish the next rule:

$$\text{If} \begin{cases} PC(Ch^{(k)}) \; < \; PC(Ch^{(k-1)}) & \Rightarrow \quad \text{send the recursive call.} \\ PC(Ch^{(k)}) \; \geq \; PC(Ch^{(k-1)}) & \Rightarrow \quad \text{stop the recursive call.} \end{cases}$$

6.3.1 Proof of the algorithm termination

The only case where the algorithm never ends arrives when the recursive calls continue indefinitely. This means that the stop condition of the recursive call $PC(Ch^{(k)}) \geq PC(Ch^{(k-1)})$ will never be true.

On the other hand the number of edges is finite and the building paths are elementary, (i.e., without circuits) then the number of building paths is finite and at the end of each path the previous condition become necessary true and then the algorithm stops.

6.4 Definition of the S function

S is the function that creates the set of indices. This set contains the identifiers of the sites that will receive the RPC. From the previous sections, we define S as:

$$S(DL_i^{j-1}, VP) = \begin{cases} \{\emptyset\} & \text{if } mpc_i^j >= mpc_i^{j-1} \\ \{\Gamma(i) - callers\_list\} & \text{if } mpc_i^j \text{ otherwise} \end{cases}$$

Where mpc_i^j is the current minimum path cost from the root to i, calculated at j [8].

7 Third step: The recursive descending scheme proof

To show that the proposed paths construction from the root to the different sites, includes the shortest path, we must show how to calculate the cost of a path, and the minimum path cost, and prove that after the descending all the leaves will have a possible shortest path.

7.1 Path costs computation.

For the root $x_0 = r$ we set

$$PC(Ch^{(0)}) = PC(x_0, x_0) = 0$$

the path cost of the root to the root is zero, from this we can compute the path cost from the root to the other sites:

$$PC(Ch^{(1)}) = PC(x_0, x_{i_1}) = PC(Ch^{(0)}) + C(x_0, x_{i_1}) \; ; \qquad x_{i_1} \in \Gamma(x_0)$$
$$PC(Ch^{(2)}) = PC(\{x_0, x_{i_1}, x_{i_2}\}) = PC(Ch^{(1)}) + C(x_{i_1}, x_{i_2}) \; ; \quad x_{i_2} \in \Gamma(x_{i_1})$$

[8] We could define S differently, changing the message size but incrementing the number of messages.

generalizing we have :

$$PC(Ch^{(n)}) = PC(\{x_0, x_{i_1}, ..., x_n\}) = PC(Ch^{(n-1)}) + C(x_{n-1}, x_n) \tag{1}$$

Using (1) we will compute the minimal path cost in the following subsection.

7.2 Definition and calculus of the minimum cost.

We define E_i^p, as the p^{th} value of the minimum cost waited from the site i.

Initially, at $p = 0$, the cost will be infinite to all the sites except the site 0, (remember that we want the minimum cost from the site 0, i.e. the root, to any site).

$$E_i^0 = \infty \;\; i \neq 0$$

The site i receives a first cost value, $PC(Ch^{(j)})$, from a neighbour j. Then from the equation 1 of the last subsection, we can estimate a first path cost value from the site i to the root:

$$PC^{(1)}(Ch^{(i)}) = PC(Ch^{(j)}) + C(i, j)$$

where $PC^{(1)}(Ch^{(i)})$ is the first path cost value.

Then the first minimum cost expected in the site i, is:

$$E_i^1 = min[PC^{(1)}(Ch^{(i)}) \; , \; E_i^o]$$

A second cost value arrives, $PC(Ch^{(k)})$, from another neighbour, (or from the father in the recursive tree). Let k be the identifier site of that neighbour, and $PC(Ch^{(k)})$ its path cost. Then we can compute a second path cost value, from the root to the site i.

$$PC^{(2)}(Ch^{(i)}) = PC(Ch^{(k)}) + C(i, k)$$

the second minimum cost waited at i will be:

$$E_2^1 = min[PC^{(2)}(Ch^{(i)}) \; , \; E_i^1]$$

Let $PC^{(m)}(Ch^{(i)})$ be the m^{th} path cost value, computed from all path costs that arrive to the site i, then we can define E_i^p as:

$$E_i^p = min[PC^{(m)}(Ch^{(i)}) \; , \; E_i^{(p-1)}]$$

Property :

The function E_i^p is decreasing and monotone in p by definition having 0 as lower bound.

`Proof :`

All path costs are positives, then function E_i^p selects only between positive values, and at each time a lowest value than the previous one. As there are not negative values the possible lowest value is zero.

Property :

The function E_i^p converges to $PC_{min}(r,i)$.

`Proof:`

It is true thanks to the fixed point theorem. The minimum path cost is computed, then this value corresponds to the lower bound.

Lemma :

With positive costs, if a path contains a circuit, it is not minimal.

`Proof :`

Obvious because the path obtained with avoidance of the circuit is of lower cost, (Bellman's principle).

Lemma :

We obtain the shortest path from the site r to all sites x in a finite number of scans.

`Proof :`

We avoid any path, with more than n edges, because it contains a circuit and it can't become a shortest path. On the other hand the number of vertices is finite.

CONSEQUENCE:

We obtain the shortest path after the last visit of the recursive wave.

7.3 Definition of the G function

Since G is the parameter value function and every site needs to know the path cost and the callers set, we define:

$$G(DL_i^j, VP, k) = \{pcost, callers\_list_i + i\} \quad \text{if} \quad \{\Gamma(i) - callers\_list_i\} \neq \{\emptyset\}$$

where i is the identifier site.

8 Fourth step: Proof of the recursive backtrack

The recursive backtrack collects the global information concerning the shortest path tree. The recursive calls compute a semi global information in each vertex. These are the shortest path cost from the root to the current site, and its father in this shortest path. The recursive backtrack allows to collect the information created after the corresponding recursive call by the other visited sites to the current site. Thus a site knows its sons in the shortest path tree. It could be possible to collect the global information from each vertex to the root, but we prefer to collect site local information because the global information collected from the root are not necessarily interesting and it can take an important amount of informationin each message. Every site i knows its father in the shortest path tree, and the minimum cost of the path from r to it. The missing information is the set of its sons in the shortest path tree. In the next subsection we show how to obtain this information.

8.1 Computation of the identifiers of the sons' site.

At the end of the "descending" each site knows its father in the shortest path tree, $father_i^*$; that can be different from its father in the execution tree, $father_i$.

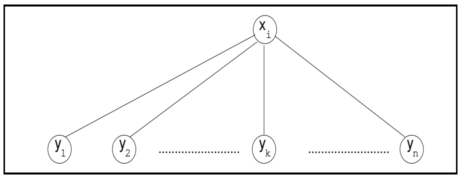

Figure 4: *A site with its sons.*

If x_i represents a site and $(y_{i_1}\ ,\ y_{i_2}\ \cdots\ ,\ y_{i_k}\ ,\ \cdots\ ,\ y_{i_n})$ its sons in the execution tree, (see figure 4), then:

$$father_{y_{i_1}} = father_{y_{i_2}} = \cdots = father_{y_{i_k}} \cdots = father_{y_{i_n}} = x_i$$

and we can establish the following rule:

RULE:

$$\text{if } father^*_{y_k} = x_i \text{ then } sons^*_{x_i} = \text{union}(y_k, sons^*_{x_i})$$

where $sons^*_{x_i}$ are the sons of the site x_i in the shortest path tree, and *union* is a set function that adds a new element to a set.

This means that if the father's identifier in the shortest path tree of the site y_k is equal to the identifier x_i, y_k is its son in the shortest path tree.

The opposite case, $(father^*_{y_k} \neq x_i)$ is true when to the site x_i arrives a better cost from another node. This makes that x_i changes its father identifier, (in the shortest path) to the one that corresponds to the arrived improvement; this father will be different to the first one.

8.2 Definition of the H function

The H function computes and sends the result of the wave backtracking to all vertices of the execution tree. At the begining of the section we mentioned the information required by each vertex, then the H function is given by:

$$H(DL_i^j, VP, H\text{sons}) = \{id\_site, father_i^*, sons_i^*, pcost\}$$

We have to remember that $father_i^*$ is the father of the site i in the shortest path tree, and not necessarily the father in the execution tree.

9 Fifth step: The local value coherence

The local value coherence, in the shortest path problem, must be treated in the backtracking. During the recursive backtrack, a vertex must know that are its sons. However it's difficult to know if a neighbour is the son of a vertex, because it can become son of another vertex. Then the collect of the information about the sons of a vertex is difficult.

To avoid modification conflicts and transactions problems we build a total order on the transactions on a process. In our case there are two ways to assume this :

- a systematic solution for this problem is to give a sequence number, (a watch stamp) at each sent message or RPC. It is a systematic way but it is not necessary here ;
- it is possible to use the cost paths values, to induct order between modifications. In the following we will use this solution.

10 Algorithms' Specification

The code of the recursive wave for the shortest path problem is described in figure 5, we use ADA-like specification language.

The data structure used in the algorithm is based on two records. They are defined as follow:

type InRec **is record**
    callers : **integer**;
    pcost : **string**;
end record

where *callers* represents all the callers arriving to the site, and *pcost* is path cost value between the caller and the site.

type OutRec **is record**
    mpc : **integer**;
    father : **string**;

sons : **string**;

end record

where *mpc* is the current minimum cost of the path from the root, r to the actual site; *father* and *sons* are the corresponding father and sons of the actual site in the shortest path tree.

The two records are used as parameters in the recursive wave procedure. The fist one is an input parameter and the second an "output" parameter, (it can be considered as a value parameter).

```
ego : id_processus; - l'identificateur du site courant
voisins : setof id_processus;

procedure court_chemin(in Vague: InRec, out Res: OutRec) is
Res: OutRec;
alien : array [1..MAX_VOISINS] of OutRec;
i,site : id_processus;

begin
    if Meilleur_Coût(Vague.coût) then
        concat( Vague.appelants, ego);
        Res.coût_minimal := Vague.coût;
        Res.père := appelant;
        par i in (voisins - Vague.appelants) do
            Vague.coût := Vague.coût + Coût(ego, i)
            court_chemin(Vague, Res) on i;
            alien(i) := Res;
        enddo;

        while site in (voisins - appelants) do
            if alien(i).père = ego then
                Res.fils := concat(Res.fils, site)
            endif
        enddo
    endif;
end court_chemin;
```

Figure 5: *Code of the recursive solution to the shortest path problem.*

11 Conclusions

In this paper we introduce a basic method for the design of distributed algorithms based on the distributed recursive wave idea.

The algorithms obtained with this method are simples to be read and understood. The design of these algorithms is closed to a distributed object representation. We have showed a general scheme for the construction of recursive distributed algortihms while we make the algorithm's proof.

Clearly every problem will be traited in a different way. We will find problems in wich somes steps will take more time and developpment than others.

All this work has been applied in one way or other. Many algorithms have been implemented and tested down the recursive distributed idea.

The failure semantic of the RPC is a problem that was not mentioned here. We believe to use the exception idea, as in ADA, to treat the problem of faults. At the same way the management of some problems' dues to the semantic of RPC will be studied.

References

[BiNe83] Andrew D. Birrell and Bruce Jay Nelson - *Implementing Remote Procedure Calls.* ACM Trans. on Computing Systems, Vol. 1, No. 3 August 1983, Pages 222-328.

[Cha90] M. Y. Chan and F. Y. L. Chin *Distributed election in complete networks* Dist. Computing, pages 19-22 No. 1, Vol. 3, 1988

[FlLa91] Gérard Florin, Ivan Lavallée - *La récursivité, mode de programmation distribuée.* Rapport de Recherche No. 1536, INRIA-Rocquencourt Octobre 1991.

[FGL93] Gérard Florin, Roberto Gómez, Ivan Lavallée - *Recursive Distributed Programming Schemes*
Proceedings ISADS'93, pages 122-128 Mars 30-31 & April 1 1993, Kawasaki Japon.

[Fre89] Greg N. Frederickson - *A single source shortest path algorithm for a planar distributed network* STACS-LNCS, pags 143-150 Springer Verlag, Vol. 182, 1985

[LSG82] Lamport L., Shostak R., Pease M. - *The Byzantine Generals Problem* ACM Toplas, Vol. 4, No. 3 July 1982, Pages 382-401.