The rudiments of object distribution in a distributed system

Claude Jard, Michel Raynal

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

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° 668

THE RUDIMENTS OF OBJECT DISTRIBUTION IN A DISTRIBUTED SYSTEM

Claude JARD
Michel RAYNAL

Mai 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Publication Interne n°358 - Avril 87 - 16 pages

The Rudiments of Object Distribution in a Distributed System *

Eléments pour distribuer un objet dans un système réparti

Claude Jard, Michel Raynal
IRISA, Campus de Beaulieu
F-35042 RENNES Cedex, France
El. mail (from UUCP-FNET): seismo!mcvax!inria!irisa!jard

Abstract

Distributed systems are built from objects and controllers. Controllers manage the access to the objects in order to keep invariant relations between different parts of the environment. We present two different approaches (extreme each other) to the distribution of objects and their associated controllers. In the first one, the object is replicated on each site of the system. In the second approach, the object is partitioned over the sites (in this latter case, a particular object: a queue of sites serves as illustration). Methods and tools in respect of the distributed control are discussed using an analytical and teaching view. We focus our attention on basic concepts (i.e. general mechanisms and their underlying assumptions), providing an unified view of the problem of object distribution in a distributed system.

Résumé

Nous présentons deux approches différentes (extrêmes l'une de l'autre) de la distribution d'objets dans les systèmes répartis. Dans la première, l'objet est répliqué sur chacun des sites. Dans la seconde approche, l'objet est partitionné sur l'ensemble des sites et un objet particulier, une file de sites sert d'illustration. Des méthodes et outils relatifs au contrôle distribué sont discutés de façon didactique, en donnant une vision unifiée du problème de la distribution d'objets dans un système réparti.

*Invited paper, 2nd Int. Symposium on Computer and Information Sciences, Istambul (october 1987). This work has been partly supported by the French national project C^3

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

1 Introduction

Two different kinds of systems are usually considered, according to their purpose. The first class consists of systems which process input data in order to produce a definite result. Termination of their computations and availability of a result characterize such systems called "transformational systems". Systems of the second class aim at reacting to their environment; this is no case to produce a result (i.e. to compute a function) but to maintain relations between different parts of their environment. They are called "reactive systems" [HP 85].

Operating systems, which support invariants, fall into the second category. A reactive system includes several controllers. Each controller must keep invariant a given relation. They are usually implemented by an automaton (a finite state machine) which reacts to the interactions produced by the environment, possibly initiating output interactions to this environment.

This may be exemplified by a mutual exclusion controller, which manages the access to a critical section. It guarantees the following invariant: at each time, the number of requests (initiated by processes of its environment) enabled to access the section is less or equal to one. To achieve that, the controller must observe the request and release input interactions and produce a list of authorizations according to the rules defining its behavior.

From an internal point of view, systems are also composed with objects: objects implement the system functions. In this paper, we are concerned with the distribution of objects and of their access control (i.e. the state machine controller).

Two different approaches (extreme each other) can be considered for the distribution problem. The first kind of solution is to replicate the object on each site of the system. Synchronization is then needed to avoid incoherent evolutions of the distributed object. There exist general methods as explained in section 2. The other extreme approach is to partition the object over the sites. Since the structure of the object must be encoded into the controllers, general methods are difficult to get. Deriving controllers from an object to be partitioned is still an open problem. A particular object, a queue of sites serves as illustration for this latter approach in section 3. Obviously, this allows for a simpler (i.e. of lesser complexity) algorithm. In both sections, we take great care to precise the assumptions about the communication environment.

Methods and tools in respect of the distributed control are discussed

using an analytical and teaching view. We focus our attention on basic concepts (i.e. general mechanisms and their underlying assumptions), providing an unified and original view of the problem of object distribution in a distributed system.

2 Replicate to distribute

2.1 Distributed objects

What we call a distributed system is a finite set of n sites, for which exchange of messages is the only mean to communicate. Communication is supported by two-way channels. We consider each site at the execution place of one process (processes and sites will be referred as synonymous in the following).

An object is defined by the set of its access operators and by the rules which order their occurrences during any execution [LZ 75, Gutt 77]. As already said, these rules can be implemented by an automaton. The access operators are called by processes.

An object instance may be implemented on one or several sites. This section deals with the case where the instance is replicated on each site. What problems put such a duplication? What solutions are available?

2.2 Principles

2.2.1 Object coherence

We consider the control activity itself distributed. There is no privileged process which might control the distributed activity: such a structure does not take advantage in general of the system distribution. We choose to duplicate the object and its associated controller on each site of the system.

The main theoretical (and practical) limit about distributed systems under distributed control is the non-observability of the instantaneous state of the system: no global clock with enough accuracy is available, the only mean to acquire knowledge is to exchange messages [CL 85]. In other words, if a process P_j does not know the occurrence of an event α when it produces the event β, there is no mean to know which event is the first: α and β are said "concurrent". The ordering of concurrent events is not significant: α occurs then β, or β then α or both simultaneously. This problem implies that some requirements in which a particular order on concurrent events is specified cannot be distributed. For example, the mutual exclusion problem where the authorizations of access to a critical section must always be in

the same order than the coresponding requests, is impossible to distribute: as already said, the order of the requests is not always observable.

Consequently, the objective must be limited to provide a "coherent" evolution of the distributed object. Coherence is achieved when all the copies of the object change in a same way [TGGL 82]. This can be done by the following rules [SL 85]:

- all the controllers start with the same initial state,
- they observe the same set of interactions initiated by processes (access operator calls),
- these interactions are observed in the same order.

It is clear that if all the state machines are identical, starting in the same initial state, and consuming the same sequence of inputs, they will present the same behavior. Thus all the copies of the object will have the same evolution.

2.2.2 Broadcasting

In order to observe the same operator calls, each process has to broadcast (using messages) its own calls to all the other sites. Such broadcast services can be easily implemented on a common bus, a complete graph, a ring ...; for arbitrary networks, there exist simple algorithms [Seg 83, Ray 87] (they are all based on the construction of a spanning tree for the distributed system modelled by a graph G ={processes,channels}).

2.2.3 Total ordering

In order to observe the operator calls in the same order, we have to introduce a device allowing to define a total order on the set of calls of the whole system. This may be achieved in two steps. The first one is to implement an abstract virtual clock to order non-concurrent events. The second is to order the concurrent events.

We require the following properties about the virtual clock:

- time progresses on all the sites and with enough accuracy,
- this progression is coherent, meaning that if a process P_j initiates a call to y (event β) after having received the knowledge of a call to x (event α) from P_i, then the date of β is more recent than the date of α.

The Lamport's logical clocks [Lam 78] are a correct implementation of the virtual clock defined above. Each site P_i is endowed with a natural variable h_i, initially set to 0 and increasing continuously. When P_i initiates an access call, it stamps and broadcast the message with the actual value of h_i. Between two successive calls, the controller associated to P_i increases h_i executing the statement $h_i := h_i + 1$. Upon the receipt of a message stamped with k, the controller adjusts its variable h_i executing the statement $h_i := max(h_i, k) + 1$. The first update defines a total order between the local calls. The second guarantees the coherence implied by communication (i.e. causality between send and receive actions).

To order concurrent calls (see section 2.2.1), the sites must use a common arbitrary rule of ordering. To define such a rule, we consider that all the sites have different and total ordered identities. The unique name of the site is added to any message initiated by the site. A call is then characterized by the time-stamp (h, i) where h and i are respectively its logical date and the name of the initiator. The Lamport's rule is the following:

$$\text{Let } \alpha \text{ and } \beta \text{ be two events stamped by } (h, i) \text{ and } (k, j)$$
$$(\alpha \prec \beta) \Leftrightarrow (h < k) \text{ or } (h = k \text{ and } i < j)$$

Until now, the implicit assumptions on the behavior of the channels are that every message is delivered within a finite delay after emission, and is never lost or altered.

2.2.4 Progressing

At this point, each controller observes the same calls to operators and knows the same total ordering of these calls based on their time-stamps. The problem now, is to consume the messages according to this ordering. How the controller C_i can conclude it has received the oldest message?

Let β be the oldest message received by C_i stamped with (k, j). Let us consider that messages cannot pass each other on a channel (i.e. channels are FIFO queues). In that case, C_i can conclude that all the messages it will receive later from P_j will have their time-stamps greater than k. It is not the case for non-FIFO channels (for which we might conclude knowing the maximum transit delay of messages)[Lam 84].

Moreover, if C_i knows (at less) one call from each of the sites, it is allowed to execute the oldest message β since those it will receive later from any site will be subsequent. The call relative to β is then said stable [SL 85]: no older call can be delivered to C_i. The converse is not true. If C_i does not

know one call from each of the sites, the call β is not stable, and C_i may receive an other call older than β from a site from which it does not have any call to execute.

In the case where all the calls of P_j have been consumed, the controller C_i is not able to conclude and may block. To avoid such a blocking, C_i must execute a progression protocol which insures that no older message is on an incoming channel.

The protocol is the following. Before consuming a call, C_i sends to C_j (from which it has no pending calls) a control message named $prog(h_i, i)$; let us remember that h_i is greater than all the time-stamps received by C_i (see section 2.2.3). When receiving such a message, C_j updates its clock h_j and answers to the site i by an acknowledgment $ack(h_j, j)$. Channels being FIFO, all the messages (if any) sent from P_j before the receipt of *prog* will be received before *ack*. Hence, C_i will be able to conclude in finite time and consume the oldest call, at worst upon the receipt of such *ack* messages.

2.3 Algorithm of a controller C_i

The work of a controller is made of three parts:

- take into account calls of operators x, y, ..., initiated by the site P_i itself;
- treat messages sent by the others controllers C_j; these messages are of the following types:
  - $operation(x, (h, j))$ which indicates a call to the operator x, initiated by P_j at the logical date h;
  - $prog(h, j)$ and $ack(h, j)$ which realize the progression protocol discussed in section 2.2.4.
- modify the object according to the calls to operators requested by the sites.

This last part acts as an interpreter consuming the sequence of modifications; the sequence is the same on each controller (if necessary, it can be made visible to the processes P_i).

Each controller C_i is endowed with two variables: the local clock h_i and an array of fifo queues $queue_i[1..n]$. The variable $queue_i[j]$ records the sequence of calls to operators initiated by P_j and perceived by C_i after broadcasting. Each call is recorded with its time-stamp (a similar algorithm has been proposed in [HV 79] to manage duplicated files).

Figure 1: Text of the algorithm

upon a call to x from P_i
$h_i := h_i + 1$;
insert $((x,(h_i,i)), queue_i[i])$;
broadcast $operation(x,(h_i,i))$.

upon the receipt of $operation(y,(h,j))$
$h_i := max(h_i,h) + 1$;
insert $((y,(h,j)), queue_i[j])$.

upon the receipt of $prog(h,j)$
$h_i := max(h_i,h) + 1$;
send $(ack(h_i,i))$ to P_j.

upon the receipt of $ack(h,j)$
$h_i := max(h_i,h) + 1$;
insert $((ack,(h,j)), queue_i[j])$.

interpreter part
while $\exists j /\ queue_i[j] \neq \emptyset$
do
1. let $\mathcal{F} = \{sites j / queue_i[j] = \emptyset\}$ $\forall j \in \mathcal{F}$. send $prog(h_i,i)$ to P_j ;
2. wait $(\forall j \in 1..n : queue_i[j] \neq \emptyset)$;
3. let γ the oldest call from $queue_i[j], j \in 1..n$. (γ is stable).
execute γ ; suppress γ

od
Remark: *ack* is used as a ghost operator for which the action is *skip*.

3 Divide to distribute

3.1 Partitioned objects

As claimed in the introduction, duplication over sites is not the only way to distribute an object. Partitioning constitutes a very interesting approach: each site owns a different part of the object. In real situations, when some kind of redundancy is required, distribution of objects relies on the both techniques.

Unfortunately, there is no general method providing for the partitioning and associated managing of objects in a distributed system. So we use some kind of heuristics [FF 84]. We propose to consider the properties of the object to be distributed as such an heuristic. Every object being characterized by a specific set of properties, we limit our presentation of the partitioning technique to a case study: a queue of sites. This is motivated by the fact that queues are widely used in systems and applications and possess well defined properties.

3.2 Partitioning a queue

3.2.1 A queue of sites

A queue of sites can contain between 0 and n sites, according to a fifo discipline. From a site P_i, the queue can be accessed by three operators:

- *enqueue* which allows P_i to enter at the rear of the queue,
- *amIfirst*? which allows P_i to know if it is at the front of the queue,
- *dequeue* which removes P_i from the queue if it has reached ultimately the front position.

Such a queue can be used to solve many problems in which sites have to wait. The mutual exclusion problem is an example: only the site at the front of the queue being authorized to enter the critical section.

As explained in section 2.2.1, a strictly fifo discipline cannot be implemented in a distributed way. We consider here only the order on the arrivals of requests at the rear of the queue (remember that the rear location changes according to the requests).

3.2.2 Principle of the solution

The queue can be easily implemented by a list: a local pointer indicating the next site in the queue if any. Such an implementation is characterized by two noteworthy elements: the front and the rear of the queue.

To be at the front of the queue may be materialized by the possession of a particular unique message (unique messages are usually called tokens). When a site leaves the front of the queue, it has only to pass the token to the next site (addressed by its pointer).

The last item of the queue is also unique, but unlike the front, each site must know it. Moreover, at any time a site can enter the queue and consequently must update the rear of the queue (which is a non-local information). Such a control property can be caught by a dynamically evolving spanning tree. The root of the tree is associated to the rear of the queue. Any site entering the queue can notify the root that it is no longer the root if there exists an ascending routing towards the root. A new spanning tree is then defined according to the new root.

These two devices, token and spanning tree, have been introduced to catch in a distributed context some basic properties of the partitioned object. They have been extracted from a mutual exclusion algorithm based on such mechanisms and proposed in [NT 87].

3.2.3 Management of the spanning tree

A spanning tree is initially defined. When a site P_i requests to enter the queue, it sends a control message $enter(i)$ to its father defined by the ascending routing towards the root, and then becomes the new root.

Upon receiving such a message, a site P_j, which is not the root, first forwards it and then updates its new father in the (new) spanning tree. Two solutions are possible: the new father may be the sender of the message or the new root P_i. As we consider complete graphs (there exists a two-way channel between any pair of sites), we choose to present the second solution. This one produces in the average case, trees of a lesser hight. If the receiver P_j is the root, it notes it is no longer the root (by considering P_i as its father) and updates its list pointer to P_i. [TN 87] claims that $O(log(n))$ messages are necessary in the average case to enter a site into the queue, and proves the correctness of the protocol in case of conflicting entering requests.

A last technical problem is to be solved to obtain the solution: the token and the root existing at every time, how to notice the emptyness of the

Figure 2: Local context of the algorithm

var
$father_i : \{1, .., n, nil\}$ init nil for P_k, k for the others;
– This variable implements the local part of the routing spanning tree –
$next_i : \{1, .., n, nil\}$ init nil ;
– Implements the local part of the queue (list pointer) –
$token\_here_i$: $boolean$ init $true$ for P_k, $false$ for the others;
– Defines the front of the queue –
$in\_the\_queue_i$: $boolean$ init $false$;
– Takes the value $true$ when P_i is in the queue –

Remark:
$(\exists j/(token\_here_j \wedge (father_j = nil) \wedge \neg in\_the\_queue_j)) \Leftrightarrow$ the queue is empty

queue? In that case, a site alone in the queue (owning the token and being the root) must set a local flag $in\_the\_queue_i$ to false before leaving.

3.3 The algorithm

A reliable two-way channel (no loss, no alteration, no overtaking of messages) is assumed between any pair of sites. Travelling times are finite.

Every site P_i, $i \in 1..n$, is endowed with some local context described in figure 2. The text of the algorithm for a process P_i is given in figure 3.

4 Conclusion

Some principles of object distribution in a distributed system have been discussed in this paper. Two lessons can be drawn from this study.

Firstly, a general method exists to distribute an object using replication. The method relies on three abstractions:

- broadcasting (of relevant events to controllers, each one managing a copy of the object),
- total ordering (on these events),

Figure 3: Text of the algorithm

```
upon a call to enqueue
  in_the_queue_i := true ;
  if father_i ≠ nil then
    send enter(i) to father_i ;
    father_i := nil
  fi ;

upon a call to amIfirst?
  result (in_the_queue_i ∧ token_here_i) ;

upon a call to dequeue
  in_the_queue_i := false ;
  if next_i ≠ nil then
    send token to next_i ;
    token_here_i := false ;
    next_i := nil
  fi ;

upon receiving token
  token_here_i := true ;

upon receiving enter(j)
  if father_i = nil then
    if in_the_queue_i
      then next_i := j
      else send token to P_j ;
           token_here_i := false
    fi
    else send enter(j) to father_i
  fi ;
  father_i := j ;
```

- progress protocol (to avoid blocking behaviors).

These abstractions can be defined as properties of the underlying system on which the object is to be distributed. Such a design of distributed applications in terms of useful control abstractions has been advocated by Schneider [Sch 86] who uses it to improve fault-tolerance. The main features of the three abstractions we have presented are essential parts of many distributed algorithms. So for example, they are mixed together in the Lamport's mutual exclusion algorithm [Lam 78]; a broadcasting abstraction is needed to provide mutual exclusion in an arbitrary network [HPR 86], and [HJPR 87] provides another use of such an abstraction as an element to solve the stable properties detection problem in a distributed system. Although the idea to structure systems in terms of abstractions is not new, it is a promising challenge to identify the useful ones for distributed applications. Consequently, two research areas should be explored: their implementations and their compositions; this latter point is a crucial one to design structured distributed applications and master them.

The second lesson is that (at the present time and limited to our knowledge) there is no general method to partition an object. In that case one has to rely the partitioning on some heuristics [FF 84]. For a queue of sites, it has been shown how the properties of a queue can lead to an interesting solution. Two distributed structures have been exhibited according to these properties: a token and a spanning tree respectively associated to its front and rear elements.

Hence mastering of distributed systems requires the definition of some paradigms which are useful control abstractions or mechanisms catching the essential features of distributed data structures.

5 Bibliography

[CL 85] Chandy K.M., Lamport L.
Distributed Snapshots: Determining Global States in Distributed Systems, ACM, TOCS, Vol.3,1, (February 1985), pp. 63-75.

[FF 84] Filman R.E., Friedman D.P.
Coordinated Computing: Tools and Techniques for Distributed Software, Mac Graw Hill, (1984), 370p.

[Gut 77] Guttag J.
Abstract Data Types and the Development of Data Structures, Comm.

ACM, Vol. 28,6, (June 1977), pp. 396-404.

[HP 85] Harel D.,Pnueli A.
On the Development of Reactive Systems, NATO ASI Series 13, Logics and Models of Concurrent Systems, (Springer Verlag), (1985), pp.477-498.

[HJPR 87] Helary J.M., Jard C., Plouzeau N., Raynal M.
Detection of Stable Properties in Distributed Applications, Proc. 6th SIGACT-SIGOPS Symposium on PODC, (August 1977), Vancouver Canada.

[HPR 86] Helary J.M., Plouzeau N., Raynal M.
A Distributed Mutual Exclusion Algorithm on an Arbitrary Network, Reseach Report INRIA 281, (1986), To appear in The Computer Journal (1988).

[HV 79] Herman D., Verjus J.P.
An Algorithm for Maintaining the Consistency of Multiple Copies, Proc. 1rt Conf. on Distributed Computing ,Huntsville, (October 1979).

[Lam 78] Lamport L.
Time, Clocks and the Ordering of Events in a Distributed System, Comm. ACM, Vol. 21,7, (July 1978), pp.558-565.

[Lam 84] Lamport L.
Using Time instead of Time-out for Fault Tolerant Distributed Systems, ACM Toplas, Vol. 6,2, (April 1984), pp. 254-280.

[LZ 75] Liskov B., Zilles S.
Specification Techniques for Data Abstractions, IEEE Trans. on Soft. Eng., Vol.1,1, (March 1975), pp. 7-18.

[NT 87] Naimi M., Tréhel M.
A Distributed Algorithm for Mutual Exclusion Based on Data Structures and Fault Tolerance, Proc. Phoenix Conf. on Computer and Communications, (1987).

[Ray 87] Raynal M.
Systemes Répartis et Réseaux, Eyrolles, (1987), 200p.

[Sch 86] Schneider F.B.
Abstractions for Fault Tolerance in Distributed Systems, IFIP, (1986), pp. 725-733.

[Seg 83] Segall A.
Distributed Network Protocols, IEEE Trans. on Information Theory, Vol. 29,1, (January 1983), pp. 23-35.

[SL 85] Schneider F.B.,Lamport L.
Paradigms for Distributed Programs, in Distributed Systems, LNCS 190, (Springer Verlag), (1985), pp. 431-480.

[TGGL 83] Traiger I.L., Gray J., Galtieri C.A., Lindsay G.B.
Transactions and Consistency in Distributed Database Systems, ACM TODS, Vol. 7,3, (September 1982), pp. 323-342.

[TN 87] Tréhel M., Naimi M.
Un Algorithme Distribué d'Exclusion Mutuelle en Log(n), TSI, Vol. 6,2, (1987), pp. 141-150.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique