Definition and implementation of a flexible communication primitive for distributed programming

Achour Mostefaoui, Michel Raynal

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Definition and Implementation of a Flexible Communication Primitive for Distributed Programming

Achour Mostefaoui and Michel Raynal

N° 2086

Novembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

1993

ISSN 0249-6399

Definition and Implementation of a Flexible Communication Primitive for Distributed Programming

Achour Mostefaoui* and Michel Raynal*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet Adp

Rapport de recherche n ° 2086 — Novembre 1993 — 12 pages

Abstract: Distributed programming has to face problems due to asynchronism of underlying communication networks. If for some applications the only use of FIFO channels eliminates the undesired effects due to asynchronism, this is generally not sufficient. Total or causal order of deliveries of messages have been proposed to overcome such problems but in some cases these orders impose a too strong property that can reduce the potential parallelism of the application.

This paper proposes a flexible broadcast primitive that attaches a type (ordinary or causal) to each message; these types impose constraints on messages deliveries. In that way the programmer is able to exploit the potential parallelism of his application in order to get an efficient program.

Key-words: broadcast primitive, causal order, delivery constraint, distributed memory, efficiency, parallel machine.

(Résumé : tsvp)

This work was partially supported by the ESPRIT project BROADCAST (number 6360) of the Commission of European Communities.

This paper will appear in the proceedings, published by North-Holland, of the IFIP WG 10.3 Int. Conf. on Applications of Parallel and Distributed Computing, April 1994.

*e-mail: {mostefaoui, raynal}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

Une primitive de communication flexible pour les applications réparties

Résumé : Cet article propose une primitive de communication répartie, qui attache un type à tout message diffusé. Ce type impose des contraintes sur la livraison du message auquel il est associé. Ainsi un message *causal* ne peut être délivré que lorsque tous les messages diffusés dans son passé l'on été, alors que la livraison d'un message *ordinaire* n'est a priori pas contrainte. Cette flexibilité permet au programmeur de mieux exploiter le parallélisme présent dans son application. La primitive proposée peut être vue comme une extension de l'ordre causal et des *flush-channels*.

Mots-clé : Contrainte de livraison, efficacité, machine parallèle, mémoire distribuée, ordre causal, primitive de diffusion

1 Introduction

In a distributed context order notions are more and more used to understand, model and design the behavior of distributed software. At the system layer designers of operating systems generally agree on the construction of a total order service that allows all nodes to perceive relevant events in the same total order. This total order aims at mastering uncertainties induced by asynchronism and failures: thanks to this unique perception of interesting system events (e.g. failure notification, join of a new node) all the nodes are supplied with the same view of the evolution of the system and consequently are able to take consistent decisions (distributed state machines [9] and virtual synchrony [2] are two well-known approaches that use such a total order to solve problems generated by unreliability of distributed supports). Nodes provided with such an operating system facility are then able to offer the users an abstract distributed machine that can correctly execute their distributed programs. In some cases the hardware support (e.g. distributed memory parallel machine) is reliable enough to be considered as such an abstract machine.

This paper considers order notions at the application layer. It is devoted to the definition and implementation of a communication primitive that allows the programmer to exploit the parallelism his application displays. The proposed communication primitive is the broadcasting of a message to all processes, with constraints on the delivery of messages. If no constraint is put on deliveries, messages can be processed as soon as they arrive at their destination processes, allowing as much parallelism as possible (as in that case processes can only be delayed by the communication network). More generally deliveries of messages can be constrained (some messages must be delivered before others) in order application processes behave consistently. Among all possible constraints causal order [2,8] is particularly useful as it associates a "sending context" with each message (this context, called its past, is composed of all the messages that causally precede [1] it); this context determines when this message can be delivered. This paper proposes two types of deliveries for messages: *ordinary* and *causal*; each type puts some constraints on the deliveries of messages they are associated with. These two types provide the application programmer with some flexibility that can be used to increase the parallelism of his distributed program.

[1] The causal precedence relation will be formally defined in Section 2. It is essentially a partial order relation on events similar to Lamport's one [4].

The paper is divided into 3 main Sections. Section 2 formally describes the constraints and their interest. Section 3 presents a protocol that implements this broadcast primitive and gives its correctness proof. Section 4 discusses extensions of this communication primitive into two directions. The first extension concerns the structural side. Ordinary and causal messages can be multicast within groups that can overlap. In other words broadcast domains are not obliged to include all processes, they can be defined according to the application structure. The second extension concerns the addition of other types of deliveries, either to restrict parallelism (*serial* messages) or to increase efficiency. We consider in this paper the less constrained are messages deliveries, the most parallel is the program. We can see that the order notion introduced (by *ordinary* and *causal* types) concern only one application program at a time [2]; that is a fundamental difference with the total order provided by an operating system which is on system events i.e. application-independent.

2 The Communication Primitive

2.1 Model of distributed computation

We consider distributed programs composed of n processes $P_1, \ldots, P_n$ that communicate by exchanging messages through unidirectional logical channels. The set of these channels constitutes the communication graph. There is neither common shared memory nor a global clock. These programs are executed by a distributed system in which processors execute processes at their own speed, physical channels are FIFO or not, and message transfer delays are finite but unpredictable. In other words the underlying system in asynchronous.

Execution of a process produces a sequence of events. Only communication events are relevant for our purpose. The broadcast of a message m by process P_i constitutes an event denoted $send_i(m)$. Every process P_j will receive this message (as channels are reliable) and the delivery of m to P_j constitutes an event denoted $del_j(m)$; only after this event, the context of m is interpreted by P_j.

[2] The algorithm implementing the proposed communication primitive can be kept in a library and linked at compile-time to the distributed program that uses it; this program can then be loaded onto the distributed machine and executed. Such an approach emphasizes the borderline between system and application layers.

2.2 Types of messages

Concerning communication, a type, *ordinary* or *causal*, is associated with each message in order to constraint its delivery. In the notation $(m, x), m$ denotes a message and x its type. These constraints prevent *del* events to occur in arbitrary order. Correct deliveries are described by rule iv) of the next Section.

2.3 Partial order associated with distributed program executions

Consider all the events *send* and *del* produced by processes of a distributed program. If there was no constraints on message deliveries (a message is delivered as soon as it arrives) all events would be structured by the Lamport's partial order relation [4]. Here, types of messages put additional constraints on the possible partial order on events. Events are structured by the following partial order relation, called *causality*, and noted $\rightarrow$.

i) for any two events e_1 and e_2 produced by the same process, with e_1 produced before e_2: $e_1 \rightarrow e_2$

ii) for any message m broadcast by P_i, we have for any P_j:

- $del_j(m)$ is an event of P_j
- $send_i(m) \rightarrow del_j(m)$

iii) if $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$.

The partial order defined by i), ii) and iii) characterizes any distributed execution based on broadcast in any asynchronous system. The next rule defines the constraint on deliveries.

iv) if $send_i(m, x) \rightarrow send_j(m', y)$ and x or y is *causal*

then $\forall k$: $del_k(m) \rightarrow del_k(m')$.

Notation: $(m, x) \rightarrow (m', y)$ or $m \rightarrow m'$ will be used as a shorthand for $send_i(m, x) \rightarrow send_j(m', y)$.

2.4 Discussion

If all messages are causal then the distributed execution obeys causal order (as defined in the distributed systems *ISIS* [2] or *Psynch* [7]). If all messages are ordinary, no constraint exists on deliveries: messages can be delivered as soon as they arrive. In the general case a causal message plays, at the communication graph level, a role analogous to the one a *marker* plays on an individual channel [3]. In both cases the causal message or the marker can be delivered only after all messages m' belonging to its "past" [3] have been delivered and before messages from its future are delivered.

Causal and ordinary types of delivery provide the programmer of a distributed program with some flexibility he can use to exploit parallelism of his application. Causal messages can be seen as control messages that "resynchronize" processes in a consistent way: when a causal message is delivered, all messages of its past have been delivered. According to particular applications only a subset of the processes can be allowed to send causal messages (for example only masters in a master-slave decomposition). Due to space limitation uses of such constraints on deliveries cannot be described in this paper.

3 The Protocol

3.1 Principle: the delivery condition

At run-time three types of events are associated with each message: sending, arrival and delivery. Due to constraints imposed by its control type (ordinary or causal) a message that has arrived can be delayed until its delivery condition becomes true. The heart of the protocol consists in stating the *delivery condition* associated with each message. To do that, each process manages, and each message carries, data structures that encode the state of the application, known by its sender, as far as messages are concerned.

The controller CTL_i associated with each P_i manages three arrays $past_i[1..n]$, $barrier_i[1..n]$ and $del_i[1..n]$ with the following meanings:

- $past_i[j]$ is, to CTL_i's knowledge, the number of messages sent by P_j (these messages are in P_i's causal past).

[3] The past of a message denotes all the messages sent causally before it. The past of a causal message m, defined by $past(m) = \{m' \mid m' \rightarrow m\}$, is not bound to a particular channel, whereas for a marker m it is composed of only those messages sent before m on the same channel.

- $past_i[i]$ is used to generate sequence numbers of messages sent by P_i.
- $barrier_i[j] = \alpha$ means that all future messages sent by P_i could be delivered only after all messages sent by P_j with sequence number less than or equal to α are delivered.
- $del_i[j]$ is a set containing all the sequence numbers of the messages sent by P_j and delivered to P_i.

Vectors $past_i$ are managed exactly as vector-clocks [5]; so we inherit properties of vector-clocks when considering $past_m$ as timestamp of message m ($past_m$ denotes the value of the array $past_i$ of the sender of m, at sending time).

Each message m also piggybacks the value of the array $barrier_i$ of its sender process P_i. Let $barrier_m$ be this value; it expresses the constraint on the delivery of m: for m to be delivered to some P_i, for any k at least all messages sent by P_k with sequence number less than $barrier_m[k]$ must have been delivered. So delivery of m is delayed until:

$$\forall k \in 1..n : \{1, 2, \ldots, barrier_m[k]\} \subseteq del_i[k]$$

After it has delivered m to P_i, CTL_i updates its context, namely $past_m[j]$ is added to $del_i[j]$ (where P_j is the sender of m), $past_i$ is updated as vector-clocks, and $barrier_i$ is updated according to the type of m (ordinary or causal).

3.2 The protocol

More formally the protocol executed by CTL_i can be described by the two following rules. The instruction *broadcast* is assumed to send a copy of the message to each process, including its sender. Integers are initialized to 0, and sets to $\emptyset$. Except for the *delay until* instruction, both statements are executed indivisibly.

Notations

Considering vectors ranging from 1 to n, we use the following notations (a, b are vectors of integers):

- 1_k is the zero vector with 1 in entry k
- $a := a + 1_k \Leftrightarrow a[k] := a[k] + 1$
- $a \leq b \Leftrightarrow \forall k \in 1..n : a[k] \leq b[k]$

- $a < b \Leftrightarrow a \leq b$ and $\exists k : a[k] < b[k]$
- $a := max(a, b) \Leftrightarrow \forall k : a[k] := max(a[k], b[k])$
- $a := b \Leftrightarrow \forall k : a[k] := b[k]$
- $set(x) = \{1, 2, \ldots, x\}$ if $x \geq 1$, else $\emptyset$

Sending rule: executed by CTL_i **when** P_i **sends** (m, x)

If $x = causal$ **then** $barrier_i := past_i$ **fi**;	1
$past_i := past_i + 1_i$;	2
broadcast $(m, x, past_i, barrier_i)$;	3
If $x = causal$ **then** $barrier_i := past_i$ **fi**;	4

Delivery rule: executed by CTL_i **when** $(m, x, past_m, barrier_m)$ **arrives from** P_j

delay until $(\forall k \in 1..n : set(barrier_m[k]) \subseteq del_i[k])$;	5
delivery of m **to** P_i; % event $del_i(m)$ %	6
$del_i[j] := del_i[j] \cup \{past_m[j]\}$; % P_j sender of m%	7
$past_i := max(past_i, past_m)$;	8
case $x = causal$ **then** $barrier_i := max(barrier_i, past_m)$	9
$x = ordinary$ **then** $barrier_i := max(barrier_i, barrier_m)$	10
endcase;	

Remark

In an actual implementation each set $del_i[j]$ can have a compact representation by using an additional variable $consec_i[j]$ with the following meaning: $consec_i[j] = \alpha \Leftrightarrow$ all messages from P_j with sequence number less than or equal to α have been delivered to P_i. As in communication protocols, the size of the $del_i[j]$ can be bounded by constraining sending of messages to receipt of additional *ack* messages ($consec_i[j]$ is then the left size of the window).

3.3 Some properties

Lemma 1

$m_1 \rightarrow m_2 \Leftrightarrow past_{m_1} < past_{m_2}$.

The proof follows directly from [5] as arrays *past* are vector-clocks used to timestamp messages.

Lemma 2

2.1 For every process P_i: $barrier_i \leq past_i$

2.2 For every message m (let P_i be its sender):

- $barrier_m < past_m$
- $barrier_m[i] \leq past_m[i] - 1$

Proof

Property 2.1 is initially true. Suppose it is true before executing each rule, we show it is true after.

1. Sending rule:

   - if x is *causal*: invariance follows from lines 1-3.
   - if x is *ordinary*: invariance follows as $barrier_i$ is not increased while $past_i$ is.

   Moreover it follows from this, that for the message m that is sent: $barrier_m < past_m$ and $barrier_m[i] \leq past_m[i] - 1$.

2. Delivery rule:

   As $barrier_m < past_m$ it follows from lines 8-10 that property 2.1 remains true. □

Lemma3

$m_1 \rightarrow m_2 \Rightarrow barrier_{m_1} \leq barrier_{m_2}$

Proof

If m_1 and m_2 have the same sender P_k, the lemma follows directly from the fact $barrier_k$ is not decreasing. If they have distinct senders label the $\rightarrow$ relation in the following way: $m_1 \stackrel{1}{\rightarrow} m_2$ if the sender of m_2 has previously been delivered m_1; in the other case we note: $m_1 \stackrel{k>1}{\rightarrow} m_2$,

with $m_1 \stackrel{k>1}{\rightarrow} m_2 \Leftrightarrow m_1 \stackrel{k-1}{\rightarrow} m'$ and $m' \stackrel{1}{\rightarrow} m_2$.

We have $m_1 \rightarrow m_2 \Leftrightarrow m_1 \stackrel{k\geq 1}{\Rightarrow} m_2$. The proof of the lemma is by induction on k.

1. $k = 1$

   At the delivery of m_1 to P_z (the sender of m_2) we have (lines 9, 10 and lemma 2: $barrier_{m_1} < past_{m_1}$): $barrier_{m_1} \leq barrier_z$.

   As $barrier_z$ is not decreasing the result follows: at the sending of m_2 by P_z: $barrier_{m_1} \leq barrier_{m_2}$.

2. $k > 1$

   Then by induction hypothesis we have: $barrier_{m_1} \leq barrier_{m'}$

   and by the base case: $barrier_{m'} \leq barrier_{m_2}$. □

Lemma 4

$((m_1, x) \rightarrow (m_2, y)$ and x or $y = causal) \Rightarrow past_{m_1}[k] \leq barrier_{m_2}[k]$ where P_k is the sender of m_1.

Proof

If m_1 and m_2 have the same sender the result follows from lines 1 and 4. In the other case let $m_1 \rightarrow m_a \rightarrow m_b \rightarrow \cdots \rightarrow m_d \rightarrow m_2$ be a causal chain of the $\rightarrow$ relation.

Case 1 $x = causal$

Consider P_z the process that has been delivered m_1 before sending m_a. We have:

- $past_{m_1} \leq barrier_z$ (as m_1 is causal and line 9)
- $barrier_z \leq barrier_{m_a}$ (lines 1-3 of P_z)

so $past_{m_1}[k] \leq barrier_{m_a}[k]$. Then by lemma 3:

$past_{m_1}[k] \leq barrier_{m_a}[k] \leq barrier_{m_b}[k] \leq \cdots \leq barrier_{m_2}[k]$.

Case 2 $x = ordinary$ and $y = causal$

Consider P_z the process that has been delivered m_d before sending m_2.

- $past_{m_1}[k] \leq past_{m_d}[k]$ (lemma 1)
- $past_{m_d}[k] \leq past_z[k] = \alpha$ (at the delivery of m_d to P_z, line 8)
- $barrier_{m_2}[k] \geq \alpha$ (as m_2 is *causal* and $past_z[k]$ cannot decrease, line 1)

so $past_{m_1}[k] \leq barrier_{m_2}[k]$. □

3.4 Safety

Safety consists in showing that deliveries respect rule *iv*) defined in Section 3.3, namely:

$$((m_1, x) \rightarrow (m_2, y) \text{ and } x \text{ or } y \text{ is } causal) \Rightarrow \forall i : del_i(m_1) \rightarrow del_i(m_2)$$

Proof

Suppose m_2 has been delivered to some P_i. Let P_k be the sender of m_1. Just before m_2 is delivered we have (line 5):

$$set(barrier_{m_2}[k]) \subseteq del_i[k] \tag{1}$$

as $past_{m_1}[k] \leq barrier_{m_2}[k]$ (lemma 4) it follows from (1) that:

$$\{past_m[k]\} \in del_i[k]$$

As the sequence number of a message from P_k is added to $del_i[k]$ only at its delivery (line 7) it follows that m_1 has necessarily been delivered before m_2. □

3.5 Liveness

Liveness consists in showing that each message will eventually be delivered.

Proof

First as channels are reliable all messages will arrive at their destination by finite time. Now consider the following delivery condition, at any P_i, for a message sent by P_j:

$$set(past_m[j] - 1) \subseteq del_i[j] \text{ and } \forall k \neq j : set(past_m[k]) \subseteq del_i[k]$$

If this condition is true, so is the actual delivery condition (lemma 2.2). We show that this condition will eventually be true for each message m (and consequently that proves liveness).

Let m_s be one of the smallest (according to the acyclic relation $\rightarrow$) of the messages that have been sent and not yet delivered to P_i (there can exist more than one such message as $\rightarrow$ is a partial order). If m_s, sent by some P_j, cannot be delivered we have:

$\exists k \neq j : set(past_{m_s}[k]) \not\subseteq del_i[k]$ or $set(past_{m_s}[j] - 1) \not\subseteq del_i[j]$

let m_{s_1} be a message sent by some P_k such that:

$$\left(\begin{array}{ll} & (k \neq j) \wedge (past_{m_{s_1}}[k] \in set(past_{m_s}[k])) \\ \textbf{or} & (k = j) \wedge (past_{m_{s_1}}[k] \in set(past_{m_s}[k] - 1)) \end{array} \right) \quad (2)$$

$$\textbf{and} \qquad past_{m_{s_1}}[k] \notin del_i[k] \quad (3)$$

we have (2) $\Rightarrow m_{s_1} \rightarrow m_s$ (lemma 1).
(3) $\Rightarrow m_{s_1}$ has not been delivered

This contradicts the hypothesis, namely m_s was one of the smallest of the messages not delivered. Consequently each message will eventually be delivered even by using this stronger delivery condition. □

4 Discussion

The previous communication primitive can be extended into two directions: groups communication and additional types of messages.

4.1 Groups communication

We have considered a broadcast primitive addressing all the processes of the application. It is possible to allow definition of groups of processes and to consider multicast as basic primitive; in that case a message sent within a group is delivered to, and only to, all processes of the group; the send operation takes the destination group as a parameter. Causality has then to be maintained through groups. Techniques developed in [6] and [8] can be used to extend the previous protocol to cope with this extension.

4.2 Additional types for messages deliveries

It is possible to add a type *serial* characterized by the following property: serial messages are causal messages whose deliveries are totally ordered. This allows to guarantee the same delivery order, at their common destination processes, for each pair of causal messages even if they are not related by relation $\rightarrow$. Of course such a possibility adds constraints on deliveries of messages and so restricts parallelism.

At the other extreme it is possible to allow more parallelism by introducing two new types *backward* and *forward*: the first one constraints the delivery of a message with respect to its past only, the second one to its

future (In fact the constraint imposed by the *causal* type can be seen as the addition of these two types). The reader interested by more details about these types can consult [1] in which these types are defined for individual channels.

5 Conclusion

This paper presented a flexible communication primitive for distributed programming. An implementation has been described and proved. Extensions have been suggested. Such a primitive allows programmers to tune the parallelism of their programs to the potential parallelism of their applications. Actually it makes feasible on the one hand the selection (thanks to ordinary messages) of the "good" asynchronism generated by the underlying supports (increasing consequently parallelism), and on the other hand the elimination (by using causal messages) of the "bad" asynchronism that creates "noise and inconsistency".

This primitive is being implemented on a distributed memory parallel machine (hypercube with 32 processors).

References

[1] M. Ahuja,
Flush Primitives for Asynchronous Distributed Systems.
Inf. Proc. Letters, Vol. 34,2,(1990), pp. 5-12.

[2] K. Birman, A. Schiper and P. Stephenson,
Lightweight and Atomic Group Multicast.
ACM TOCS, Vol. 9,3,(1991), pp. 273-314.

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

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

[5] F. Mattern,
Virtual Time and Global States of Distributed Systems.

Proc. of Int. Workshop on Parallel and Dist. Systems, North-Holland, 1988, pp. 215-226.

[6] A. Mostefaoui, M. Raynal,
Causal Multicast in Overlapping Groups: Towards a Low Cost Approach.
Proc. 4th Int. Conf. on Future Trends of Dist. Comp. Systems, Lisboa, (Sept. 1993).

[7] L.L. Peterson, N.C. Bucholz and R. Schlichting,
Preserving and Using Context Information in Interprocess Communication.
ACM TOCS, Vol. 7,3, (1989), pp. 213-246.

[8] M. Raynal, A. Schiper and S. Toueg,
The Causal Order Abstraction and a Simple Way to Implement it.
Inf. Proc. Letters, Vol. 39, (1991), pp. 343-350.

[9] F.B. Schneider,
Implementing Fault-tolerant Services Using the State Machine Approach: a Tutorial.
ACM Computing Surveys, vol. 22,4(Dec. 1990), pp. 299-320.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
ISSN 0249-6399