An Implementation of Global Flush Primitives Using Counters

M. Ahuja, Michel Raynal

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

An Implementation of Global Flush Primitives Using Counters

M. Ahuja, M. Raynal

N° 2395

Novembre 1994

PROGRAMME 1

ISSN 0249-6399

An Implementation of Global Flush Primitives Using Counters

M. Ahuja*, M. Raynal**

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

Rapport de recherche n° 2395 — Novembre 1994 — 9 pages

Abstract: We present an implementation of *Global-Flush* Primitives using counters. This implementation costs comparable to the most lightweight implementation of causal ordering. Thus, at a comparable cost, the presented implementation enriches functionality compared to causal ordering; as *Global-Flush* Primitives permit making an assertion about messages sent in the past of sending m, in the future of sending m, about both, or neither, where the past and the future of an event is defined using the relation "happened before." Using *Global-Flush* Primitives, a message can be sent to any subset of processes specified as a parameter.

Key-words: Synchronization, Asynchronous Distributed Programming Systems, Efficiency, Parallel Machines, High-performance Computing and Communication.

(Résumé : tsvp)

Paru dans Parallel Processing Letters, Vol 5,1, (janvier 1995).

*Dept. of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0114, mahuja@cs.ucsd.edu. This research is supported in part by NSF under grants MIP–9111045 and MIP-9296204.

**IRISA, Campus de Beaulieu, 35042 Rennes Cédex, FRANCE raynal@irisa.fr. This work was supported in part by the ESPRIT project BROADCAST (Number 6360) of the Commission of European Communities.

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

Une mise en oeuvre des primitives *global flush* fondée sur les compteurs

Résumé : Cet article présente une mise en oeuvre de l'ensemble de primitives connues sous le nom de *global flush*. Ces primitives permettent de faire des assertions sur le passé et le futur des messages. Elles incluent comme cas particulier l'ordre causal. L'implémentation proposée est fondée sur les compteurs.

Mots-clé : efficacité, machines parallèles, synchronisation, systèmes répartis asynchrones.

0 Introduction

Asynchronous distributed systems research, in the past, has been mainly motivated by the need to improve fault tolerance. Asynchronous distributed systems, however, can also be used to improve performance or reduce distributed-program execution time. Towards this, waits by processes for receiving messages become important. Clearly, a program will have higher-performance if at execution time processes have to wait lesser for receiving messages. To aim at higher-performance, i.e a reduction in process waits for receiving messages, reference [2] proposed *Global-Flush* Primitives for a broadcast environment, and reference [4] presented *Global-Flush* Primitives for sending a message to any subset of processes specified as a parameter. *Global-Flush* Primitives is a suite of four primitives for sending message m such that each specifies one of the following: (0) there is no restriction on the order of receipt of m and messages sent in the past/future of sending m unless specified by the options used for other messages; (1) messages sent in the past of sending m must be received before m; (2) messages sent in the future of sending m must be received after m; and (3) messages sent in the past/future of sending m must be received before/after m. The past and the future of an event is defined using the relation "happened before" [13], as is also done in other relevant studies [5, 9, 14, 15, 16]. These primitives reduce waits by a receiver process (and reduce the resequencing delay) by increasing the number of possible orders in which messages can be received compared to message receipt in the order of sending. This reduction has been shown [6] to be significant when the past and the future of an event is defined using the relation "local before;" and this reduction is expected to increase, when the past and the future of an event is defined using the relation "happened before," as we do. Other researchers [8] have also pointed to the need for reduction in the resequencing delay.

In addition to the higher performance, these primitives offer another advantage that is particularly useful for system design. Using first three of these primitives, we should be able to design algorithms that, when superimposed on an underlying computation, do not interfere with the computation[0].

For a broadcast environment, reference [2] gave applications of *GFPs* for algorithms for termination detection; identifying consistent cuts [7]; recording snapshots (global snapshots without channels states); recording global snapshots; implementing a shared token; and accessing replicated data using mutual exclusion. Primi-

[0] That is, if sending or receipt of a message of the underlying computation is not inhibited when the algorithm is not superimposed, then it would not be inhibited when the algorithm is superimposed. This property is useful for many applications, e.g. algorithms for program correction tools (debugger) or monitoring of systems.

tives implemented here can be used for these applications in a general environment in which a message is sent to a subset of processes.

Clearly, the issue of efficiency of implementation determines viability of these primitives. For a broadcast environment, reference [2] addresses this issue successfully and presents an implementation using counters that has costs comparable to the most light weight implementation of causal broadcast (CBCAST) [5]. In this paper we extend this implementation to *Global-Flush* Primitives for sending a message to a subset of processes. In the process we also extend definition of *flush-vector-time* [2] from a broadcast environment to an environment in which a message can be sent to a subset of processes. This extension involves maintaining in *flush-vector-time* two counters per channel in the system as opposed to two counters per process p in the broadcast environment [2] (in a broadcast environment, since each message is sent to every other process, for p there must be an outgoing channel to every other process and the counters for each of these channels will always have the same value and so all such counters can be merged in to one for p; so the implementation of [2] can be seen as a particular case of the one presented in this paper.)

Compared to causal ordering [5] which involves maintaining vector time which is one counter per channel in the system this implementation involves maintaining *flush-vector-time*, which as we mentioned requires two counters per channel. This increase in the cost, we think, is marginal and is worth the resulting gains, i.e. an ability to implement *Global-Flush* Primitives which increase the number of orders in which message can be received compared to causal ordering. This implementation is an improvement over the implementation presented in [10] because this uses only two counters per channel, has a natural reinitialization of counters, uses a vector as opposed to a matrix, and is simpler.

1 The System Model

The asynchronous distributed system comprises a finite number N of processes (each of which is viewed as a sequence of events) that communicate with each other by asynchronous sending and receipt of messages along any of the finite number of unidirectional channels. We use p, q, r, and s to refer to processes, and use $c_{p,q}$ to refer to a channel from p to q. We use $S(\mathcal{Q},m)$ to denote event of sending m to a specified set of processes $\mathcal{Q}$; whenever $\mathcal{Q}$ is not relevant, we use $S(m)$ to denote the send event. The event $S(\mathcal{Q},m)$ or $S(m)$ is atomic. We use R_q to denote a receipt event at q; and use $R_q(m')$ to denote q receiving m'. A receipt event is atomic. For

a message m', we refer to a message sent in the past of $S(m')$ by m and a message sent in the future of $S(m')$ by m''.

A process q receives m' by executing command *receive*($m'.se$, $m'.ty$, $m'.co$), where $m'.se$, $m'.ty$, and $m'.co$ are variables to which are assigned the identifier of the sender of m', the type of m' (defined by the primitive used to send m'), and the contents of m', respectively. After issuing the command, q waits until m' is delivered to it.

Let $\mathcal{E}$ be the set of send and receipt events that happen in the system. We assume that no spurious messages are added and message transmission takes finite time. We assume that each message is sent using our primitives. We make no assumptions about the relative speed of processes.

The system has four primitives *o-send*, *f-send*, *b-send*, and *t-send*, and we refer to a message m' sent using them by o', f', b', and t', respectively. We refer to the four primitives collectively as *GFPs* and give their definitions from [4].

Definition 0 (Relation "Happened Before" [13]) *Happened Before, denoted by* $\rightarrow$*, is the smallest relation on* $\mathcal{E}$ *such that* $e \rightarrow e''$ *if (0)* e *and* e'' *happened on the same process and* e *happened locally before* e'' *with respect to the process's clock, or (1)* e *is an* $S(m)$ *and* e'' *is the receipt of* m*, or (2) there exists an* e' *in* $\mathcal{E}$ *such that* $e \rightarrow e'$ *and* $e' \rightarrow e''$. ∎

Definition 1 (Past of an event e')
$\mathcal{P}(e') = \{e \mid e \rightarrow e'\}$. ∎

Definition 2 (Future of an event e')
$\mathcal{F}(e') = \{e'' \mid e' \rightarrow e''\}$. ∎

Definition 3 (*g-overtake*) m' *is said to g-overtake (read as 'globally overtakes')* m *if* $S(\mathcal{Q},m) \in \mathcal{P}(S(\mathcal{Q}',m'))$ *and there exists* q *such that* $q \in \mathcal{Q} \cap \mathcal{Q}'$ *and* $R_q(m) \in \mathcal{F}(R_q(m'))$. ∎

Definition 4 (o') *An* o' *is sent using primitive o-send. It does not specify any restriction on* o' *g-overtaking any other message or any other message g-overtaking* o'. ∎

Definition 5 (f') *An* f' *is an* m' *sent using primitive f-send. It does not g-overtake any message, that is,* $(\forall\, m, q, \mathcal{Q}, \mathcal{Q}' : S(\mathcal{Q},m) \in \mathcal{P}(S(\mathcal{Q}',m')) \ \wedge \ q \in \mathcal{Q} \cap \mathcal{Q}' : R_q(m) \in \mathcal{P}(R_q(m')))$. ∎

Definition 6 (b') *A b' is an m' sent using primitive b-send. No message g-overtakes it, that is,* $(\forall\, m'', q', \mathcal{Q}', \mathcal{Q}'' : S(\mathcal{Q}'',m'') \in \mathcal{F}(S(\mathcal{Q}',m')) \wedge q' \in \mathcal{Q}' \cap \mathcal{Q}'' : R_{q'}(m'') \in \mathcal{F}(R_{q'}(m')))$. ∎

Definition 7 (t') *A t' is an m' sent using primitive t-send. It does not g-overtake any message and no message g-overtakes it:*
$(\forall\, m, m'', q, q', \mathcal{Q}, \mathcal{Q}', \mathcal{Q}'' :$
$S(\mathcal{Q},m) \in \mathcal{P}(S(\mathcal{Q}',m')) \ \wedge \ q \in \mathcal{Q} \cap \mathcal{Q}'$
$\wedge \ S(\mathcal{Q}'',m'') \in \mathcal{F}(S(\mathcal{Q}',m')) \ \wedge \ q' \in \mathcal{Q}' \cap \mathcal{Q}'' :$
$R_q(m) \in \mathcal{P}(R_q(m')) \ \wedge \ R_{q'}(m'') \in \mathcal{F}(R_{q'}(m')))$. ∎

Note that the function of a t cannot be achieved using fs and bs. Also, note that the order in which an o can be received is restricted due to restrictions specified by other messages.

In the sequel, we use bt to refer to an m that is either b or t, of to refer to an m that is either o or f, ob to refer to an m that is either o or b, and ft to refer to an m that is either f or t.

GFPs can be implemented using F-channels[1]. F-channels can be implemented using any one mechanism among, e.g selective flooding [3], counters [12, 17], or acknowledgments, or a combination of the three. So *GFPs* can also be implemented using any one mechanism among, e.g selective flooding, counters, or acknowledgments, or a combination of the three. An implementation using any of these mechanisms, if specifically designed for *GFPs* can be more efficient than an implementation of *GFPs* using F-channels which are implemented using that mechanism. In this paper we demonstrate this using counters.

2 The flush-vector-time

To lead us to a simple presentation of the implementation, here, we extend definition of *flush-vector-time* [2] from a broadcast environment to an environment in which a message can be sent to a subset of processes. We denote the resulting timestamp by τ, which is assigned values using *flush-vector-clock*. Flush-vector-clock of p, $p.c$ is a vector of elements one (element) per unidirectional channel. Elements of this vector are arranged in the lexicographic order among channels using pairs ¡sender, receiver¿ for channels. Element of $p.c$ or τ for channel $c_{r,s}$ will be referred to as $p.c[r,s]$ or as $\tau[r,s]$. The same τ is assigned to $S(m')$ and m' and is denoted by $S(m').\tau$ and $m'.\tau$. By design, a receive event is not assigned a τ. Flush-vector-time is of general use with respect to potential causality of all $S(m')$ in $\mathcal{E}$ and is preferable to vector

time [9, 14] since, at a comparable cost, it provides more reasoning power and richer functionality, e.g. use in implementing GFBCASTs which increase the number of possible orders in which messages can be received compared to CBCASTs [5].

We design a *Global Flush Manager* (*GFMan*) for a p, which executes at the same site as p, executes p, maintains $p.c$ as we shall discuss, assigns $S(m').\tau$ and $m'.\tau$, receives messages for p sent by any process, and delivers messages to p.[1] The transmission of an m' to *GFMan* for a receiver entails transmission of the identifier of the sender of m', the type of m', the $m'.\tau$, and the contents of m'.

Central to the design of $p.c$ is the idea of batching messages sent by a process to another process such that on sending of a bt[2] the current batch closes and a new batch opens. Using this idea, two counters are maintained by *GFMan* for every $c_{p,q}$, which together form $p.c[p,q]$. The first counter is $p.c[p,q].btp$, which is the number of $\underline{b}ts$ (or closed batches) sent in the $\underline{p}$ast along $c_{p,q}$. The second counter is $p.c[p,q].sbt$, which is the number of messages sent $\underline{s}$ince the last $\underline{bt}$ was sent (or the current batch was opened) along $c_{p,q}$. Each element $p.c[r,s]$ is a pair of two integers. We note a relationship between two integers of such a pair; $p.c[p,q].sbt$ is reinitialized to 0 when $p.c[p,q].btp$ is incremented by one. This suggests that such a pair can be implemented using one word[3]. For pairs of integers, let $\prec$ denote the lexicographic ordering. We define $\preceq$ and $\prec$ relations on two flush-vector-clock values or two timestamps τs, as follows:

Definition 8 (Relations $\preceq$ and $\prec$ on two flush-vector-clock valuesor two timestamps τs)

For any two flush-vector-clock values or two timestamps τs, say τ_1 and τ_2:
$\tau_1 \preceq \tau_2$ *iff for each* $c_{r,s}$ $\tau_1[r,s] \preceq \tau_2[r,s]$;
and $\tau_1 \prec \tau_2$ *iff* $\tau_1 \preceq \tau_2$ *and there exists* $c_{p,q}$ *such that* $\tau_1[p,q] \prec \tau_2[p,q]$. ∎

[1] Deliverability of a message depends on the communication architecture and will be discussed later; messages may be delivered in an order different from the order of their arrival at *GFMan* for the receiver, which may be different from the order of their sending.

[2] Recall first that bt denotes a message that may be either a b or a t and second that no message can *g-overtake* a bt.

[3] $p.c[p,q].sbt$ can be represented by the first w_1 (least significant) bits and $p.c[p,q].btp$ by the next w_2 bits of a word of $w_2 + w_1$ bits. Assigning such semantics reduces the cost of our implementation for all practical purposes and may not cause any overflow compared to use of two words if w_1 is carefully selected. For example, for $w_2 + w_1 = 64$ and $w_1 = 32$, as long as 1) the number of sends along a channel between two successive bts, and 2) the number of bts sent along the channel, each, is not more than 2^{32} (about 4 billions), this representation does not lead to an overflow. Further, more flexibility in avoiding overflow can be achieved by providing an ability to change w_1 by associating with each element a variable that gives its value.

Using definition 8 we write rules for updating $p.c$ for the *GFMan* for p presented next.[4] At any time, *arrived-set* has messages that have arrived at the *GFMan* for p but have not been put in to *deliverable-set*; *deliverable-set* has all messages that are currently deliverable to p.

GFMan(p)::
1: $\star$ [true $\rightarrow$ [Execute p until p issues an $S(\mathcal{Q}',m')$ or an R_p;
1.1: **if** p issued an $S(\mathcal{Q}',m')$ **then**
1.1.1: for each $q \in \mathcal{Q}'$ **do** $p.c[p,q].sbt := p.c[p,q].sbt + 1$ **od**;
1.1.2: $S(m').\tau$:=$p.c$; $m'.\tau$:=$p.c$; execute $S(\mathcal{Q}',m')$ of p;
1.1.3: **if** m' is a *bt* **then**
for each $q \in \mathcal{Q}'$ **do** $p.c[p,q].btp := p.c[p,q].btp + 1$; $p.c[p,q].sbt := 0$; **od**
1.2: **if** p issued an R_p and *deliverable-set*$\neq\emptyset$ then
1.2.1: choose any m' from *deliverable-set*;
1.2.2: τ:=$m'.\tau$; if m' is a *bt* from r then $\tau[r,p]$:=<$bt.\tau[r,p].btp$+1,0>;
1.2.3: $p.c := sup(\tau, p.c)$; /* Where *sup* denotes the elementwise maximum using $\preceq$ /*
1.2.4: deliver m' to p; execute R_p of p; delete m' from *deliverable-set* **fi**]
[] The network has a message for $p \rightarrow$ add it to *arrived-set* for p;
[] [*arrived-set* $\neq\emptyset \rightarrow$ non-deterministically select an m from *arrived-set*;
if *deliverable(m)* then add m to *deliverable-set* and delete m from *arrived-set*;
/**deliverable(m)* is defined for the given communication abstractions /*]]
The initialization is: *deliverable-set*, *arrived-set*:=$\emptyset$, for each r and s $p.c[r,s].btp, p.c[r,s].sbt := 0, 0$;

The following loop invariants follow from definitions.

I0: $p.c[p,s].sbt$ = the number of messages sent so far by p to s after sending the last *bt* to s.

I1: $p.c[p,s].btp$ = the number of *bt*s sent so far by p to s.

I2: $p.c[r,s]$ is the highest value (using $\preceq$) that $r.c[r,s]$ has taken, as per data structures maintained by p.

I3: After the iteration that lead to sending of message number j (which can be maintained as an auxiliary variable) by p to s and given that the last *bt* that p sent to s was send number i by p to s, $j = i + p.c[p,s].sbt$.

Theorem 0 *For all m, m':* $S(m) \in \mathcal{P}(S(m')) \equiv S(m).\tau \prec S(m').\tau$.
Proof: Follows from invariants and an inductive argument. ∎

[4] We use CSP [11] syntax (i.e. '$\star$' for a loop, '[]' for an alternate command, and '$\rightarrow$' for a guard.)

3 An Implementation of Global-Flush Primitives

Our task is to constrain message receipts by processes such that messages are received without violating any of the restrictions. We do so by designing a boolean function *deliverable*(m') which *GFMan* for p uses to determine deliverability of m' in the *arrived-set* of *GFMan*.

deliverable(m')::
[For an ft' /* Recall that an ft' cannot *g-overtake* any other message. */
if all ms such that $S(\mathcal{Q},m)$ is in $\mathcal{P}(S(\mathcal{Q}', ft'))$ and p belongs to $\mathcal{Q} \cap \mathcal{Q}'$ have been delivered
x /* All such bts can be identified using rule 0 and all such ofs can be identified using rule 1. */
then return *true* (i.e. ft' is *deliverable* to p)
else return *false*;
For an ob' /* Recall that an ob' cannot *g-overtakes* a bt (but may *g-overtake* an of). */
if all bts such that $S(\mathcal{Q},bt)$ is in $\mathcal{P}(S(\mathcal{Q}', ob'))$ and p belongs to $\mathcal{Q} \cap \mathcal{Q}'$ have been delivered
/* Such bts can be identified using rule 0. */
then return *true* (i.e. ob' is *deliverable* to p)
else return *false*;
]

Rule 0 *(for identification of bt messages) Given an m' and r,*
$S(\mathcal{Q},bt)$ is in $\mathcal{P}(S(\mathcal{Q}',m'))$ and $p \in \mathcal{Q} \cap \mathcal{Q}'$ for exactly $m'.\tau[r,p].btp$ (number of) bts,
each with unique $bt.\tau[r,p].btp$ such that $0 \leq bt.\tau[r,p].btp < m'.\tau[r,p].btp$,
and only for these bts. ∎

Rule 1 *(for identification of of messages) Given an m' and r,*
$S(\mathcal{Q},of)$ is in $\mathcal{P}(S(\mathcal{Q}',m'))$ and $p \in \mathcal{Q} \cap \mathcal{Q}'$ for exactly the following ofs sent by r:
(0)[5] *For each bt such that $0 \leq bt.\tau[r,p].btp < m'.\tau[r,p].btp$, $bt.\tau[r,p].sbt - 1$ (number of) ofs with unique*
$of.\tau[r,p]$ such that $of.\tau[r,p].btp = bt.\tau[r,p].btp$ and $0 < of.\tau[r,q].sbt < bt.\tau[r,q].sbt$.
(1)[6] *If $r \neq m'.se$ then $m'.\tau[r,q].sbt$ (number of) ofs with $of.\tau[r,q].btp = m'.\tau[r,q].btp$ and unique $of.\tau[r,q].sbt$*
such that $0 < of.\tau[r,q].sbt \leq m'.\tau[r,q].sbt$
else $m'.\tau[r,q].sbt - 1$ (number of) ofs with $of.\tau[r,q].btp = m'.\tau[r,q].btp$ and unique of

[5] These *of*s are sent by r before the last bt sent by r to p.
[6] These *of*s are sent by r after the last bt sent by r to p.

$.\tau[r,q].sbt$
such that $0 < of.\tau[r,q].sbt < m'.\tau[r,q].sbt$. ∎

Theorem 1 (Safety) *GFMan implements GFPs.*
Proof : For a bt' and any m'' such that $S(m'')$ is in $\mathcal{F}(S(bt'))$, $S(bt')$ is in $\mathcal{P}(S(m''))$. Hence, by function *deliverable*(m') and rule 0, no message *g-overtakes* bt'. By function *deliverable*(m') and rules 0 and 1, an ft' does not *g-overtake* any message. ∎

Theorem 2 (Liveness) *GFMan for p will deliver to p each sent message.*
Proof : Each message sent to p will be received by the *GFMan* of p as message transmission to the *GFMan* is reliable. At a given time all messages sent to p that have been received by *GFMan* of p and that have minimal timestamps according to $\prec$ will be deliverable. By induction on timestamps and since a *GFMan* does not indefinitely delay delivery of a message that is deliverable, each message will eventually become deliverable. ∎

Acknowledgments : We thank anonymous referees for their helpful comments.

References

[1] M. Ahuja. *Flush* primitives for asynchronous distributed systems. *Information Processing Letters*, 34(2):5–12, Feb. 1990.

[2] M. Ahuja. Assertions about past and future in highways: Global flush broadcast and flush-vector-time. *Information Processing Letters*, 1(48):21–28, October 1993.

[3] M. Ahuja. An implementation of *F-Channels. IEEE Transactions on Dist. and Par. Systems*, 4(6):658–667), June 1993.

[4] M. Ahuja. Assertions about past and future: Communication in a high-performance distributed system highways. In *proceedings Symposium on Parallel and Distributed Processing*, pages 241–244. IEEE, December 1993.

[5] K.P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[6] T. Camp, P. Kearns, and M. Ahuja. Fast batched data transfer with flush channels. Technical Report WM–93–1, The College of William and Mary, 1993.

[7] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[8] D. Cheriton and D. Skeen. Understanding the limitations of causality and totally ordered communications. In *proceedings of the fourteenth Symposium on Operating Systems Principles*. ACM, December, 1993.

[9] J. Fidge. Partial orders for parallel debugging. In *Proceedings of ACM SIGPLAN/SIGOPS workshop on parallel & Distributed Debugging*, pages 183–194, 1985.

[10] A. Gahlot, M. Ahuja, and T. Carlson. Global *Flush* communication primitive for sending a message to a group of processes. In *proceedings Symposium on Principles of Distributed Computing*. ACM, 1994.

[11] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[12] P. Kearns, T. Camp, and M. Ahuja. Implementation of *Flush* channels based on a verification methodology. In *proceedings Twelfth Int. Conf. on Distributed Computing Systems*, pages 336–343. IEEE, 1992.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[14] F. Mattern. Time and global states of distributed systems. In *proceedings Workshop on Parallel and Distributed Algorithms, Elsevier*, pages 215–226, 1988.

[15] L.L. Peterson, N.C. Buchholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[16] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, September 1991.

[17] K. Shafer and M. Ahuja. Distributed modeling and implementation of high performance communication architectures. In *proceedings Thirteenth Int. Conf. on Distributed Computing Systems*. IEEE, 1993.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du 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