Causality oriented shared memory for distribured systems

Michel Raynal, Masaaki Mizuno, Mitchell Neilsen

INRIA

UNITÉ 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

1 9 9 2

25*ème anniversaire*

N° 1680

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

CAUSALITY ORIENTED SHARED MEMORY FOR DISTRIBUTED SYSTEMS

Michel RAYNAL
Masaaki MIZUNO
Mitch NEILSEN

Mai 1992

RR-1680

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

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

Causality oriented shared memory for distributed systems.

Michel RAYNAL, Masaaki MIZUNO, Mitch NEILSEN
Programme 1, Projet ADP (Algo. Distribués et Applications)
IRISA
Campus de Beaulieu
35042 Rennes Cédex
raynal@irisa.fr

Publication Interne n° 656 - Avril 1992 - 8 pages

Abstract

This paper presents an efficient protocol that implements causal memory in a distributed system. This protocol is deduced from a general framework that allows to derive other implementations.

Mémoire partagée causale pour systèmes répartis.

Résumé

Cet article décrit un algorithme réparti qui implémente une mémoire causale dans le contexte des systèmes répartis. Cet algorithme est déduit d'un cadre très général duquel peuvent être dérivés d'autres protocoles.

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)

Causality Oriented Shared Memory for Distributed Systems (Extended Abstract)

Michel Raynal
IRISA
Campus de Beaulieu
35042 Rennes-Cédex, FRANCE

Masaaki Mizuno Mitchell L. Neilsen
Dept. of Computing and Info. Sciences
Kansas State University
Manhattan, Kansas 66506

1 Introduction

Implementation of shared data objects raises several problems in distributed systems. Among these, the efficiency of access is an important issue. In order to improve the efficiency of read access to an object, a traditional approach is to maintain several copies (replicas) of the object. However, this approach causes the problem of possible inconsistency between copies of the data object. Several replica control protocols have been proposed to solve this problem. Replica control protocols ensure that different copies of each object appear to the users as a single non-replicated object. This property is called *one-copy equivalence*. In other words, one-copy equivalence ensures that a value returned by a read operation is the value written by the latest write operation to the object.

For many applications, the consistency provided by one-copy equivalence is not necessary [1]. By relaxing the definition of consistency, the complexity of managing replicated objects may be significantly reduced [1, 2]. One such definition is called *causal consistency*, which is defined based on causal relations among values in objects. Memory that satisfies causal consistency is called *causal memory*.

The contributions of our work are twofold. The first is to provide a general framework of a protocol which effectively captures causal relations among values in objects. The second is to show an efficient protocol, based on the general framework, that implements causal memory.

2 Causal consistency and causal memory

This section reviews causal consistency and causal memory. Most of the definitions and notation presented in this section follow [1].

A system consists of N processors, and the processors share causal memory. The causal memory may store M different objects. Processors issue two types of operations on the memory: read and write. A read operation on object x issued by processor P_i is denoted $r_i(x)v$, where v is the value returned. A write operation on object x issued by processor P_i is denoted $w_i(x)v$, where v is the value written. If the processor which issued an operation is not important, the subscript may be omitted. Similarly, if the value read or written is not important, that may be omitted.

Let o and o' be two operations on the causal memory. Operation o is said to causally precede o' (denoted $o \rightarrow o'$) *if*

1. o and o' are successive operations issued by the same processor, and o is executed before o',

2. o and o' are write and read operations, respectively, and o' reads the value written by o, or
3. there exists operation o" such that o $\rightarrow$ o" and o" $\rightarrow$ o'.

If o and o' are not related by $\rightarrow$, they are said to be concurrent (denoted by o $\|$ o'). Let o_1, o_2, $\cdots$, o_k be a sequence of operations such that $o_i \rightarrow o_{i+1}$ for $1 \leq i \leq (k-1)$. We call such a sequence a *causal path* from o_1 to o_k.

Definition 1 (Live Values): Let o and o' be operations r(x) and w(x)v, respectively. Then, the value v is live for o if

1. o $\|$ o', or
2. o' $\rightarrow$ o and there exists no operation o" on any causal path from o' to o such that o' $\rightarrow$ o", o" $\rightarrow$ o, and o" is either w(x)v' or r(x)v' (v' is produced by some write on x other than o'). If such o" exists, we say that o" invalidates the value v.

Definition 2 (Causal Memory): An execution on causal memory is correct if the value returned by every read operation is live for the read operation.

3 Implementation

3.1 A general framework

Since the correct and efficient implementation of causal memory requires identification of causal relations among values written in the objects, we first consider a general framework to capture such causal relations. Vector clocks are often used to identify causal relations among events in message passing systems [4, 7, 8]. Thus, it is natural to use a mechanism similar to vector clocks for our purpose.

For vector clocks in a message passing system, causal relations among instances of one unique type, namely *events*, are of concern. Since all of the events produced by one process are totally ordered, they are uniquely identified by numbering them; for instance, "the fourth event produced by process 2" uniquely identify the event produced in the system. We call such numbers event identifiers. Each process i maintains one dimensional array, VT_i[Process_Range]. When process i has produced a new event e_i, the value $VT_i[j]$ at this moment denotes that "all the events up to and including the $VT_i[j]^{th}$ event produced by process j causally precede e_i." Furthermore, the value $VT_i[i]$ denotes the identifier of e_i.

Now, let's consider causal relations among values written into objects. We first assume that each object x is fully duplicated at each processor, and that each processor performs write operations on its local copies. Note that this assumption will be modified in Section 3.2. As with events in vector clocks, all the writes on an object performed by one processor may be uniquely identified by numbering them, such as "the fourth write on object x performed by processor 2."

Unlike vector clocks in message passing systems, which manage causal relations among only events, all of the writes on the same and different objects are of concern in the causal memory system; that is, causal relations must be identified among all of the different writes. Thus, as a natural extension, two dimensional arrays are used instead of one dimensional array. Let each processor i maintain two dimensional array, $CR2_i$[Object_Range, Processor_Range]. When processor i writes on object x, the value $CR2_i$ denotes that "all the writes up to and including the $CR2_i[y, j]^{th}$ write on object y performed by processor j causally precede this write on x." Value $CR2_i[x, i]$ denotes the identifier of this write.

Now, we have a tool to capture causal relations among all the values in objects. We call this method, which uses the two-dimensional arrays, the *general framework*. Based on the general framework, we presents an efficient protocol to implement causal memory.

3.2 An efficient protocol

Managing two dimensional arrays in the system is very costly. There are several ways to simplify the implementation. In this section, we will describe an efficient protocol to implement causal memory based on one such simplification.

In this implementation, we collapse the two-dimensional arrays $CR2_i$ into one-dimensional arrays CR_i[Object_Range], such that $CR_i[x] = \sum_{k=1}^{N} CR2_i[x, k]$, $1 \leq x \leq M$. This is possible by assigning unique system wide identifiers to all the writes on one object, instead of processor-level local identifiers. We call such identifiers *version numbers*. For example, a write may be identified as "version 4 of a write on object x."

For simplicity, as does the implementation by Ahamad et al. [1], we assume a primary copy approach. For each object, there is a primary copy site. A write operation to object x requires the processor to communicate with the primary copy site of x to write to the primary copy. Each processor maintains a set (maybe subset) of objects in its cache memory. A primary copy site does not use the primary copy as a cache. For example, assume that the primary copy site of object x is processor i. If processor i needs to access x, it allocates memory for x in cache, aside from the memory assigned for primary copy x.

Primary copy approach provides the following two advantages:

1. System-wide, unique version numbers may be issued by the primary copy site.
2. The primary copy can always provide a live value to any read operation; thus, when a processor needs to read a live value, it reads from the primary copy.

Data structures maintained by processor i are as follows:

- If processor i is the primary copy site of object x, it maintains memory area P[x] consisting of two parts:
  - P[x].value
  - P[x].CR[Object_Range]
    P[x].CR[y] ($x \neq y$) : write on object y of version P[x].CR[y] is the last write on y that causally precedes x
    P[x].CR[x] : the version number of the write which created this x
- As described above, processor i maintains a single-dimensional array CR_i[Object_range].
- Processor i keeps track of a set of valid cache objects in $VALID_i$.
- For each object $x \in VALID_i$, processor i maintains cache area $C_i[x]$ consisting of:
  - $C_i[x]$.value
  - $C_i[x]$.version : the version number of write which created x.

Operations at processor i are described below:

Write(x,v)::
send [write, x, v, Live] to the primary copy site of x;
receive [x, Vn] from the primary copy site of x; /* Vn: version number */
$C_i[x]$.value := v; $C_i[x]$.version := Vn; $VALID_i := VALID_i \cup \{x\}$;
$CR_i[x]$:= Vn;

Read(x)::
if $x \notin VALID_i$
then
send [read, x] message to the primary copy site of x;

```
        receive [x, P[x]] message from the primary copy site;
        C_i[x].value := P[x].value;  C_i[x].version := P[x].CR[x];  VALID_i := VALID_i ∪ {x};
        for each y ∈ VALID_i, y ≠ x
a:        if C_i[y].version < P[x].CR[y] then VALID_i := VALID_i − {y} fi; /* invalidation */
b:      CR_i := update(CR_i, P[x].CR); /* update ≡ componentwise max */
    fi;
    return C_i[x].value;

  Process [write, x, v, CR_j] message from processor j::
    increment(P[x].CR[x]); CR_j[x] := P[x].CR[x]; /* generate the new version number */
c:  P[x].CR := CR_j;
    P[x].value := v;
    return [x, P[x].CR[x]] to processor j; /* return the new version number */

  Process [read, x] message from processor j::
    return [x, P[x]] to processor j;
```

In the above protocol, the steps labeled **a**, **b**, and **c** deserve special attention:

Step a: Invalidation of cache takes place only when the processor reads from primary copies, since this is the only situation which introduces new causal relations that may invalidate some of the values cached locally.

Step b: Update operation is performed because read operation merges two causal paths: (1) the causal path defined by the execution of read and write operations on the processor and (2) the causal path carried by the value read.

Step c: After a write operation is performed to the primary copy, the subsequent read operations on this new value receive the causal information associated with only the new value and do not receive any causal information associated with the previous (overwritten) value. Thus, instead of an update operation, the replace operation (except for the new version number) is performed at **Step c**.

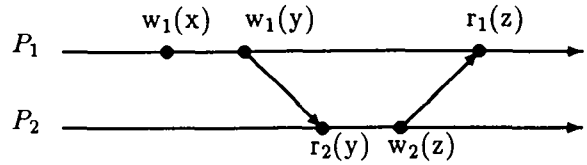
3.3 Properties

1. **Property 1**: A primary copy always provides a live value.
2. **Property 2**: At **Step a** in the protocol, $C_i[y].version < P[x].CR[y]$ implies that there is a write on y with a higher version number than $C_i[y].version$ on a causal path leading to this read operation. Note that it, however, does not necessarily mean that the value in $C_i[y]$ is invalidated by the write on y of version P[x].CR[y] (denoted w'[y]), since w'[y] (or a read from w'[y]) may not be on any causal path from the write on y of version $C_i[y].version$ (the write that produced the value stored in $C_i[y]$) to this read operation. However, for safety, the protocol invalidates such a local copy of y.
3. **Property 3**: At **Step a** in the protocol, $C_i[y].version \geq P[x].CR[y]$ implies that there is no write on y with higher version number than $C_i[y].version$ on any causal path leading to this read operation. This implies that the cache value y is still live.

4 Discussions and related work

We can view the protocol proposed by Ahamad et al. [1] to be another special case generalized by our framework. Their protocol is considered to have collapsed two dimensional arrays $CR2_i$

to one dimensional arrays VT_i[Processor_Range] in such a way that $VT_i[j] = \sum_{k=1}^{M} CR2_i[k, j]$, for $1 \leq j \leq N$. In this simplification, causal relations among different objects become unclear. Thus, the protocol tends to invalidate more live cache values than our protocol. Consider the following simple example.



If the protocol by Ahamad et al. is applied, r1(z) invalidates both x and y. This is because in their protocol, when P1 performs r1(z), it only knows that "P2 has updated *some* value" but does not know which value. Thus, for safety, it must invalidate both x and y, considering the worst case scenario in which P2 has updated x or y. However, if our protocol is applied, it knows that "P2 has updated z, not x or y." Therefore, our protocol does not invalidate either x or y.

Several definitions of consistency in multicache systems have been proposed. Traditionally, the consistency definition used in such systems is the *atomic consistency* [3, 6], in which a read always returns the last written value with respect to the global physical time. The notion of atomic consistency has been relaxed by Brown [2] to a weaker definition that corresponds to the consistency known as the *sequential consistency* [5]. These notions of consistency are stronger than the causal consistency and consequently allow less concurrency and less parallelism.

5 Conclusion

In this paper, we proposed a general framework of a protocol to capture causal relations among object values. Based on the general framework, we presented an efficient protocol to implement causal memory. We also showed that the protocol proposed by Ahamad et al. [1] is a special case described by our general framework.

References

[1] M. Ahamad, P.W.Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th ICDCS*, pages 274–281, 1991.

[2] G.M. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[3] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, 1978.

[4] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[5] L. Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[6] K.M. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[7] F. Mattern. Virtual time and global states of distributed systems. In Cosnard, Quinton, Raynal, and Robert, editors, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Bonas, France, 1989. North-Holland.

[8] D.S. Parker el al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–246, 1983.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 648 SET-THEORETIC GRAPH REWRITING
Jean-Claude RAOULT, Frédéric VOISIN
Mars 1992, 18 pages.

PI 649 UNE STRUCTURE D'INFORMATION POUR LES ALGORITHMES D'EXCLUSION MUTUELLE FONDES SUR UNE ARBORESCENCE
Jean-Michel HELARY, Achour MOSTEFAOUI, Michel RAYNAL
Mars 1992, 18 pages.

PI 650 BLOCK-ARNOLDI AND DAVIDSON METHODS FOR UNSYMMETRIC LARGE EIGENVALUE PROBLEMS
Miloud SADKANE
Avril 1992, 24 pages.

PI 651 COMPILING SEQUENTIAL PROGRAMS FOR DISTRIBUTED MEMORY PARALLEL COMPUTERS WITH PANDORE II
Françoise ANDRE, Olivier CHERON, Jean-Louis PAZAT
Avril 1992, 18 pages.

PI 652 CHARACTERIZING THE BEHAVIOR OF SPARSE ALGORITHMS ON CACHES
Olivier TEMAM, William JALBY
Avril 1992, 20 pages.

PI 653 MADMACS : UN OUTIL DE PLACEMENT ET ROUTAGE POUR LE DESSIN DE MASQUES DE RESEAUX REGULIERS
Eric GAUTRIN, Laurent PERRAUDEAU, Oumarou SIE
Avril 1992, 16 pages.

PI 654 SYSTEMES D'EQUATIONS RECURRENTES
Patrice QUINTON
Avril 1992, 20 pages.

PI 655 DIFFUSION ON SCALABLE HONEYCOMB NETWORKS
Dominique DESERABLE
Avril 1992, 24 pages.

PI 656 CAUSALITY ORIENTED SHARED MEMORY FOR DISTRIBUTED SYSTEMS
Michel RAYNAL, Masaaki MIZUNO, Mitch NEILSEN
Avril 1992, 8 pages.

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

ISSN 0249-6399