From Serializable to Causal Transactions for Collaborative Applications

Michel Raynal, G. Thia-Kime, M. Ahamad

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

From Serializable to Causal Transactions for Collaborative Applications

M. RAYNAL , G. THIA-KIME , M. AHAMAD

N ° 2802

Février 1996

THÈME 1

ISSN 0249-6399

INRIA
RENNES

From Serializable to Causal Transactions for Collaborative Applications

M. RAYNAL*, G. THIA-KIME*, M. AHAMAD†

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n ° 2802 — Février 1996 — 22 pages

Abstract: Serializability is the traditional consistency criterion when shared objects are accessed concurrently. Its main drawback lies in the strong synchronization constraints it imposes on execution of applications when they are run on distributed systems. In this paper we examine weaker consistency criteria for computations in which accesses to shared objects are grouped to form transactions. In particular, we explore causal consistency and causal serializability for transaction based computations. These criteria turn out to be sufficient for a class of applications (*e.g.*, collaborative applications) and their implementation results in greater availability of data and improved performance. These criteria are formally defined and protocols implementing them are presented. We demonstrate that causal consistency allows both read and update transactions to be executed in a wait-free manner. Although write accesses in causal serializability require synchronization with other nodes, read accesses appearing in update transactions can be executed locally. Finally, fault-tolerance problems are addressed in the context of process crash failures.

Key-words: Transactions, consistency, causality, serializability, fault-tolerance.

(Résumé : tsvp)

An abstract of this paper appeared in the Proceedings of the 15th ACM Symposium on Principles of Distributed Computing, May 1996, under the title: "From Serializable to Causal Transactions".

* IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France, {raynal, thiakime}@irisa.fr
† College of Computing, Georgia Tech, Atlanta, GA 30332, U.S.A, mustaq@cc.gatech.edu.

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

Des transactions sérialisables aux transactions causales pour le travail coopératif

Résumé : La sérialisabilité est le critère de cohérence traditionnel lorsque des objets partagés sont accédés de façon concurrente. Son principal inconvénient réside dans les fortes contraintes de synchronisation qu'elle impose sur l'exécution des applications lorsqu'elles sont exécutées sur des systèmes distribués. Dans cet article, nous examinons des critères de cohérence plus faibles pour des calculs dans lesquels les accès aux objets partagés sont groupés pour former des transactions. En particulier, nous examinons la cohérence causale et la sérialisabilité causale pour des calculs utilisant les transactions. Ces critères s'avèrent être suffisants pour une classe d'applications (par exemple les applications coopératives) et leur implantation résulte en une plus grande disponibilité des données et une meilleure efficacité. Ces critères sont formellement définis et les protocoles les mettant en œuvre sont présentés. Enfin, la question de tolérance aux défaillances est traitée dans le contexte des pannes de processus.

Mots-clé : Transactions, cohérence, causalité, sérialisabilité, tolérance aux défaillances.

1 Introduction

In a shared object model, a consistency criterion defines which is the value that must be returned to a process when it reads an object, and a protocol implementing a consistency criterion describes how processes have to be synchronized in order to ensure that they read correct values (*i.e.*, satisfy the consistency criterion). Serializability and the two phase locking protocol, mainly studied and used in the database field, are the best known examples of a consistency criterion and its associated implementation protocol.

Traditional consistency criteria (namely, *atomicity* [19], *serializability* [6] and *linearizability* [11]) require that all processes have the same sequential view of the computation. This view is formally defined as a total order on operations issued by processes and an execution is correct if any read of an object gets the last value previously written into this object (the words 'last' and 'previously' refer to the total order of operations defined by the common view). These criteria have largely been studied. But, if they are natural and easy to use, their implementations are based on strong synchronization constraints that severely limit efficiency of distributed applications as soon as these applications are composed of many processes or cover a large geographic area.

In this paper, we are interested in exploring weaker consistency criteria in which the *causality relation* between read and write operations on shared objects plays a central role. These criteria reveal to be sufficient to match data consistency requirements of a class of applications and their implementations result in greater availability of data and better performance. For example, in a collaborative editing application, asynchronously interacting users may want to access a shared document that is composed of many chapters. Each user corresponds to a process that executes one or more transactions. A query transaction reads chapters of interest to the user. Causal consistency guarantees that the user will always get a set of chapters that include all causally preceding updates. It is possible that concurrent transactions initiated by different users update and define new versions of some chapters. Consistency criterion that is weaker than serializability can be defined to deal with such concurrent updates. It can ensure that a user has a causally consistent view of all chapters and all users get the same 'last' version of each chapter at the end of an editing session. Many other applications from the domain of computer supported cooperative work have data consistency requirements that are naturally met by causality based consistency criteria.

A novel aspect of our work is that we consider an abstraction level at which read and write operations are encapsulated inside transactions. Thus, rather than a single

operation on a single object, the consistency criteria must address transactions that may manipulate many objects. Two new consistency criteria, causal consistency and causal serializability, are introduced in the context of systems composed of sequential processes that execute transactions.

Causal consistency is the weaker criterion considered: in addition to the sequentiality on transactions issued by each process, it considers only dependency on transactions due to a *read-from* relation. This relation is defined in the following way: a transaction that reads a value written by another transaction is dependent on it. So, with causal consistency, two concurrent transactions that write into the same object can be perceived in a different order by two processes (with serializability, they are perceived in the same order). The second criterion considered, *causal serializability*, lies between causal consistency and serializability and is a consistency criterion strong enough to satisfy a wide range of applications (*e.g.* inventory control, distributed dictionaries, reservation systems or cooperative work). Causal serializability is causal consistency plus the following constraint: all transactions writing into the same object must be perceived by all processes in the same sequential order. This ensures there is always one and only one 'last' value for each object (in particular, there is a unique last value of each object for all processes, at the end of the computation).

We develop implementations of the consistency criteria in an environment where copies of shared objects are maintained at each node where they are accessed. Causal consistency allows both *query* (*i.e.*, read-only) and *update* (*i.e.*, read-write) transactions to be executed in a wait-free manner because they can complete by accessing the local copies of the objects. Causal serializability requires synchronization for write accesses of update transactions but read accesses issued by update or query transactions can be completed with local copies of shared objects. In contrast, serializability requires that both read and write accesses appearing in update transactions synchronize with other transactions. Furthermore, serializability imposes stronger ordering requirements for messages that are used to propagate updates to shared objects.

The paper is composed of five main sections. Section 2 introduces the computational model (basically an execution is a partially ordered set of transactions executed by processes). Section 3 revisits serializability and formally defines causal consistency and causal serializability. Section 4 presents protocols implementing these criteria in a distributed system. Section 5 addresses fault-tolerance in the case of crash failures.

2 Shared Objects Model

2.1 Preliminary definitions

We consider a system composed of a finite set of sequential processes $P_1, P_2, \ldots, P_n$ which interact through a finite set X of shared objects. Each object $x \in X$ can be accessed by a read or a write operation. A write into an object defines a new value for the object; a read allows a process to obtain a value of the object. The execution of a write operation that assigns the value v into object x is denoted $w(x)v$ (for simplicity, and without loss of generality, we assume all values written into an object are different). The execution of a read operation of the object x, that returns value v is denoted $r(x)v$.

A process P_i executes transactions. A *transaction* t is a "procedure" composed of read and write operations. It is assumed that every transaction is structured in the following way: first it reads shared objects, then it does internal computation (*i.e.*, computation not involving shared objects), and finally it issues write operations on shared objects; moreover, an object is read (written) at most once by a transaction[1]. $R(t)$ and $W(t)$ denote the set of objects read and written, respectively, by transaction t. If $W(t) = \phi$, transaction t is called *query*; if $W(t) \neq \phi$, t is called *update*.

At the abstraction level defined by transactions, the execution of a process P_i is modeled as the sequence $t_i^1 t_i^2 \ldots t_i^k \ldots$ where t_i^k denotes the k-th transaction executed by P_i. Such a sequence defines the local history $\widehat{h_i}$ of P_i. Let h_i denote the set of transaction executions issued by P_i and $\rightarrow_i$ be the total order relation on transactions issued by P_i. $\widehat{h_i}$ is the totally ordered set $(h_i, \rightarrow_i)$.

2.2 Execution Histories

An *execution history* (or simply a history) of a shared objects system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that:

- $H = \bigcup_i h_i$
- $t1 \rightarrow_H t2$ if:
  - (i) $\exists\ P_i$: $t1 \rightarrow_i t2$ (in that case $\rightarrow_H$ is called *process-order* relation)
  - (ii) $\exists\ w(x)v$, $r(x)v$ such that $w(x)v \in t1$ and $r(x)v \in t2$ (in that case $\rightarrow_H$ is called *read-from* relation)

[1] This restriction can easily be overcome by reading shared object values in private variables and computing with them.

(iii) $\exists\, t3$: $t1 \rightarrow_H t3$ and $t3 \rightarrow_H t2$ (transitivity)

As we can see, an execution history is defined at the transaction abstraction level. As in database transaction systems, read and write operations induce precedence on transactions but do not appear explicitly in a history.

Two transactions $t1$ and $t2$ are *concurrent* in $\widehat{H}$ if $\neg(t1 \rightarrow_H t2)$ and $\neg(t2 \rightarrow_H t1)$.

3 Consistency of Shared Objects

This section defines three consistency criteria for shared objects accessed by processes through transactions. These definitions are based on the *legality* concept.

3.1 Legal Transaction

Let us consider a history $\widehat{H}$. Informally, a transaction $t \in H$ is legal if it does not read overwritten values. More formally, legality of a transaction is defined in the following way.

Definition. Legal transaction. A transaction t is *legal* if $\forall r(x)v \in t$: $\exists\, t'$ such that :

- $t' \rightarrow_H t$ (t' precedes t)
- $w(x)v \in t'$ (t' is the transaction that wrote v into x)
- $\forall\ t''$ such that $t' \rightarrow_H t'' \rightarrow_H t$: $w(x) \notin t''$ (there is no overwriting transaction)

3.2 Serializability

This is the classical consistency criterion for database transactions[2] [6]. Informally, serializability expresses the fact that a history $\widehat{H}$ has to be equivalent to some sequential execution of the same set of transactions in order to be consistent. Formally, it can be defined in the following way.

[2] When considering the database context, processes in our model correspond with transaction managers that would execute transactions serially.

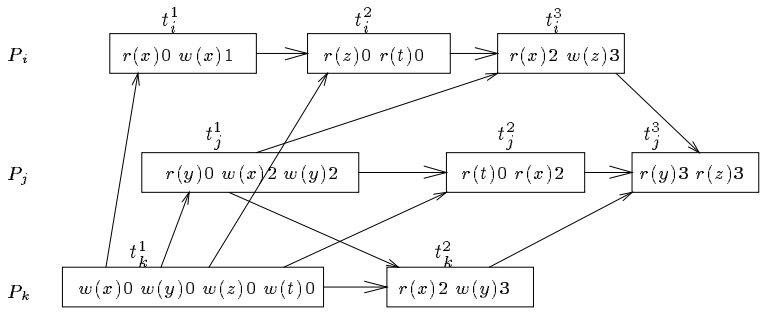

Figure 1: A serializable history $\widehat{H1}$

Definition. Serializability. A history $\widehat{H} = (H, \rightarrow_H)$ is *serializable* if it admits a linear extension[3] $\widehat{S}$ in which all transactions are legal. (Such a linear extension $\widehat{S}$ constitutes the common *view* perceived by every process.)

As an example, let us consider an execution modeled by history $\widehat{H1}$ (Figure 1)[4]. $\widehat{H1}$ is serializable since there exists a linear extension $\widehat{S1} = t_k^1\ t_i^1\ t_j^1\ t_k^2\ t_i^2\ t_j^2\ t_i^3\ t_j^3$ in which all transactions are legal. As we can see, this is the traditional consistency criterion for shared data.

Remark. Let us note that if every transaction is reduced to include either a single read or a single write operation, serializability is the same as *sequential consistency* [14], which is the most used criterion to define semantics of memories in shared memory systems [1, 20].

3.3 Causal Consistency

While serializability considers that all processes must have the same sequential view of the whole execution $\widehat{H}$ (the view defined by a legal linear extension), causal consistency is weaker in the following sense: it allows each process to have its own sequential view of the execution $\widehat{H}$ as long as the individual views preserve the causality relation $\rightarrow_H$. The set of operations that can affect the behavior of a

[3] A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{H} = (H, \rightarrow_H)$ is a topological sort of this partial order, *i.e.*, (i) $S = H$, (ii) $t1 \rightarrow_H t2 \Rightarrow t1 \rightarrow_S t2$ ($\widehat{S}$ maintains the order of all ordered pairs of $\widehat{H}$), and (iii) $\rightarrow_S$ defines a total order.

[4] In all figures, *process-order* edges are denoted by thick arrows and *read-from* edges by thin arrows. Additional edges that are due to transitivity are not indicated.

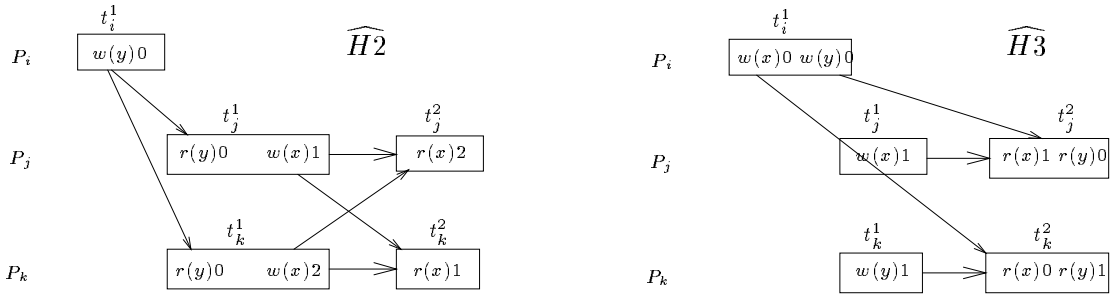

Figure 2: (a) A causally consistent history $\widehat{H2}$ and (b) a causally serializable history $\widehat{H3}$

process P_i are all operations of its own transactions plus the set of all writes issued by transactions executed by other processes. More precisely, causal consistency requires that, for each process P_i, there exists a linear extension of $\widehat{H}$ in which all transactions of P_i are legal.

Definition. Causal Consistency. Let $\widehat{H} = (H, \rightarrow_H)$ be a history. $\widehat{H}$ is *causally consistent* if, for each process P_i, there exists a linear extension of $\widehat{H}$ in which all transactions issued by P_i are legal. (Let $\widehat{S_i}$ be such a linear extension from which all queries not issued by P_i have been removed; $\widehat{S_i}$ is called P_i's *view* of history $\widehat{H}$.)

As an example, let us consider history $\widehat{H2}$ (Figure 2a). $\widehat{H2}$ is not serializable since there does not exist a linear extension of $\widehat{H2}$ in which all transactions are legal. However, $\widehat{H2}$ is causally consistent as there exists, for each process P_i, a linear extension including all update transactions plus all query transactions issued by P_i, in which all transactions issued by P_i are legal. More precisely, these linear extensions are: $\widehat{S2_i} = t_i^1\ t_j^1\ t_k^1$ for P_i, $\widehat{S2_j} = t_i^1\ t_j^1\ t_k^1\ t_j^2$ for P_j, and $\widehat{S2_k} = t_i^1\ t_k^1\ t_j^1\ t_k^2$ for P_k.

This example shows the main difference between causal consistency and serializability: with causal consistency, concurrent updates can be perceived in a different order by two processes (t_j^1 and t_k^1 are concurrent in $\widehat{H2}$ and perceived differently by P_j and P_k in $\widehat{S2_j}$ and $\widehat{S2_k}$ respectively) while they must be perceived in the same order by all processes with serializability. It is important to note that, a process

considered alone[5] cannot know whether the execution is serializable or only causally consistent.

Remark. Let us note that if every transaction is reduced to a single read or single write operation, then causal consistency becomes identical to *causal memory* [3].

3.4 Causal Serializability

For some applications, serializability is too strong a consistency criterion while causal consistency is too weak. With causal consistency, when two update transactions that write into the same object are concurrent, they can be ordered differently by two processes in their views of the execution. The aim of causal serializability is to prevent such a possibility by adding the following constraint to causal consistency: all transactions that update the same object must be perceived in the same order by all processes. This constraint ensures that, for each object, there is a unique 'last' value on which all processes agree. It follows from this definition that causal serializability lies between causal consistency and serializability. Formally, causal serializability is defined in the following way.

Definition. Causal Serializability. A history $\widehat{H} = (H, \rightarrow_H)$ is *causally serializable* if:

(i) it is causally consistent (let $\widehat{S_i}$ be the linear extension representing P_i's view of $\widehat{H}$), and

(ii) for every object x and for any pair of transactions $t1$ and $t2$ that write into x: $t1$ is ordered before $t2$ in all linear extensions $\widehat{S_i}$ or $t2$ is ordered before $t1$ in all these linear extensions.

As an example, let us consider an execution modeled by history $\widehat{H3}$ (Figure 2b). It is easy to see that $\widehat{H3}$ is not serializable as there is no linear extension including all transactions in a legal way. $\widehat{H3}$ is causally serializable since (1) it is causally consistent as there exists a legal linear extension for each process, namely $\widehat{S3_i} = t_i^1\ t_j^1\ t_k^1$, $\widehat{S3_j} = t_i^1\ t_j^1\ t_j^2\ t_k^1$, $\widehat{S3_k} = t_i^1\ t_k^1\ t_k^2\ t_j^1$ and (2) any pair of transactions writing into the same object are ordered in the same way ($t_i^1\ t_j^1$ for x and $t_i^1\ t_k^1$ for y) in the view of each process, *i.e.*, in $\widehat{S3_i}$, $\widehat{S3_j}$ and $\widehat{S3_k}$.

[5] *i.e.*, no process has hidden interaction with the other processes.

Note that $\widehat{H2}$, which is causally consistent, is not causally serializable since updates t_j^1 and t_k^1 which write into the same object x cannot be ordered in the same way by P_j and P_k (they are ordered in one way in $\widehat{S2_j}$ and in another way in $\widehat{S2_k}$).

Remark. Let us replace constraint (ii) by the following one:

(ii') all update transactions of $\widehat{H}$ are totally ordered.

It is possible to show that constraints (i)+(ii') imply serializability. This generalizes to transaction systems a result presented in [3, 24] for shared memory systems (*i.e.*, systems where processes read and write memory locations). These references establish sufficient conditions under which a program, run on a causal memory, behaves as if it was executed on a sequentially consistent memory. One of these sufficient conditions is that all write operations be Concurrent-Write Free (in short CWF, *i.e.*, no two write operations are executed concurrently). In the particular case where transactions reduce to a single read or write operation, CWF is identical to constraint (ii').

4 Implementation Protocols

A protocol implementing a consistency criterion must ensure all execution histories will satisfy the constraint defining the criterion considered. Till now, no particular assumption on the way objects are implemented has been made in defining these criteria. So, a family of protocols can be designed for systems in which each object has a single copy, while another family can be designed for systems where each object has several copies.

We consider here a full replication scheme: each object x is replicated on each process P_i; x_i will denote the copy of x located on P_i. In order to correctly implement a particular consistency criterion, each P_i is superimposed on a protocol that manages its local copies of the objects. A broadcast primitive is used by the protocol to send updated values of objects. We assume that this primitive provides reliable delivery and hence a copy of the message is delivered to all processes except its sender. We do not assume any message ordering guarantees from the broadcast primitive.

4.1 Causal Consistency

The main point of a protocol that has to ensure causal consistency is the tracking of causal dependencies between transactions. To do such a tracking, each process

P_i is endowed with a control variable $vt_i[1 \ldots n]$ of integers, initialized to 0. The set of all these vectors is managed in the following way :

- each time P_i issues an update transaction, it increments $vt_i[i]$ by 1.
- each time a message is sent, it carries the value of the vector vt of its sender.
- when a process P_i receives a message m carrying vector vt, it delays the processing of m until all updates revealed by vt have been applied to its local copies.

We can see that vectors vt_i constitute an adaptation of vector clocks ([18]) to our problem. Actually, $vt_i[j]$ represents the number of update transactions executed by P_j, to P_i's knowledge.

When P_i executes a transaction, it atomically does the following sequence of actions (remember all read operations are done at the beginning of t and write operations at its end). If the transaction is a query, only steps 1 and 2 are executed.

1. $\forall x \in R(t)$: read the value of x_i
2. execute the computation defined by the transaction
3. $\forall y \in W(t)$: update y_i with its new value v^y
4. $vt_i[i] := vt_i[i] + 1$
5. **broadcast** *update(i, vt_i, {(y, v^y): $\forall y \in W(t)$})*

As indicated previously, when a message *update(j, vt, S)* is received by P_i, the protocol delays its processing until all causally preceding updates have been applied to local copies. Then it updates atomically the local copies with the new values reported in the message.

On receiving *update (j, vt, S)*
delay the processing of this message until $((vt_i[j] + 1 = vt[j]) \wedge (\forall k \neq j$: $vt_i[k] \geq vt[k]))$;
$vt_i[j] := vt_i[j] + 1$;
$\forall$ $(y, v^y) \in S$ *do* $y_i := v^y$ *od*

Actually, this protocol is similar to the one proposed in [3] to implement causally consistent memories; its correctness proof follows the same principle. It is also interesting to note that this protocol is the same as the one proposed in [23] to implement causally ordered communications on top of asynchronous systems. Additionally, this implementation of causal consistency is *wait-free*: a process executing a transaction

is never suspended due to a synchronization constraint added by the protocol. From this property, we can conclude that causal consistency copes naturally with partitions (in that case it is only necessary to ensure *update* messages are not lost and will be delivered when partitions merge).

4.2 Causal Serializability

Causal serializability must ensure (i) causal consistency and (ii) for each object, a total ordering of all update transactions writing into it. As the previous algorithm (described in Subsection 4.1) guarantees point (i) we have only to augment it with synchronization rules guaranting point (ii). A simple way to do this is by associating a token with each object and requiring that an update transaction first acquire the tokens for the objects it wants to write and retain these tokens till it has broadcast its *update* message.

Tokens transfer should be done in a manner that is consistent with the delivery of other messages so causality is not violated. This can be easily realized by using a delivery rule for tokens similar to the one used for the delivery of *update* messages. So, a token carries the value of the vector vt of its last owner (the last process that used this token) and is delivered to requesting process P_i only when all updates known by the token have been applied to P_i's local copies (*i.e.*, when $\forall k \ : vt_i[k] \geq vt[k]$).

The previous rules are translated in the following statements. Let *token_x* be the unique token associated with object x. When P_i executes a transaction, it executes the following steps (as before, steps 3-8 are executed atomically and, if a transaction is a query, it executes only steps 3 and 4).

1. $\forall y \in W(t)$: request *token_y*
2. wait till all requested tokens have been delivered
3. $\forall x \in R(t)$: read the value of x_i
4. execute the computation defined by the transaction
5. $\forall y \in W(t)$: update y_i with its new value v^y
6. $vt_i[i] := vt_i[i] + 1$
7. **broadcast** *update(i,* vt_i*,* $\{$*(y,* v^y*):* $\forall y \in W(t)\}$*)*
8. $\forall y \in W(t)$: release *token_y*

When P_i receives an *update* message it executes the same steps as in the previous algorithm. As indicated, the delivery of a token is delayed to ensure correct tracking of causal dependencies:

On receiving *token_x(vt)*
delay the delivery of token_x until $(\forall k \ : \ vt_i[k] \geq vt[k])$

Management of tokens. The management of tokens has to ensure that any process requesting a token will eventually get it. This can be accomplished by associating Lamport's time-stamps [13] with update transactions and by piggy-backing these time-stamps on requests issued by processes to get tokens. Due to the uniqueness of the time-stamp associated with an update, all conflicts for tokens will be solved in the same way, guaranteeing the absence of deadlocks. Moreover, monotonically increasing time-stamps ensure absence of starvation. (More details on such an use of Lamport's time-stamps to avoid deadlock and starvation when allocating several resources -here, tokens- can be found in [22].)

Correctness of the protocol.

Theorem 1: The previous protocol ensures causal serializability.

Proof (sketch): Let $\widehat{H}$ be an execution that obeys the synchronization rules described by the protocol that implements causal serializability. To prove $\widehat{H}$ is causally serializable, we show that (i) $\widehat{H}$ is causally consistent and (ii) all update transactions accessing the same object are ordered in the same way in every view $\widehat{S_i}$ of each P_i.

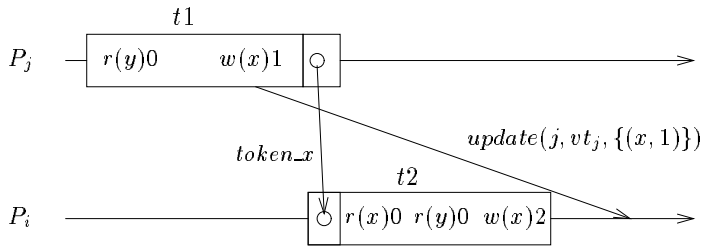

Figure 3: The *update* message cannot bypass the token

Proof of (i): This proof is based on the fact that this algorithm is built on top of the algorithm described in Section 4.1 which ensures causal consistency. It differs from it in the addition of tokens. So, we only need to verify that tokens management

preserves causal consistency. This follows from the fact that delivery of tokens to processes is done according to causal ordering [23] and so does not violate causal consistency.
Proof of (ii): Point (ii) could be violated if after a write on x (transaction $t1$) by a process P_j, process P_i first receives *token_x* allowing it to write x (transaction $t2$) and later receives from P_j the *update* message with the new value of x related to $t1$ (see the space-time diagram depicted in Figure 3). In that case, P_i will (incorrectly) perceive $t2 \rightarrow_{S_i} t1$ while P_j will (correctly) perceive $t1 \rightarrow_{S_j} t2$. This is impossible as the vector vt piggy-backed by *token_x* constraints its delivery to occur only after the *update* message related to the new value of x. Let $\rightarrow_x$ be the total order relation created by *token_x* on all updates transactions writing x. It follows from the previous discussion that, if $t1 \rightarrow_x t2$, then $t1$ is ordered before $t2$ in every view $\widehat{S_i}$.

4.3 Sketch of a Protocol for Serializability

A protocol implementing serializability can be devised in a way similar to the one used for causal consistency and causal serializability. Its design follows the remark of Section 3.4 which stated that a history $\widehat{H}$ is serializable if (i) it is causally consistent and (ii') all update transactions of $\widehat{H}$ are totally ordered.

Point (i) is implemented by the protocol defined in Section 4.1. Point (ii') could be (inefficiently) implemented by a unique token that would order all update transactions. A more efficient implementation of point (ii') is provided by the two following rules R1 and R2. Basically, R1 ensures serializability for all update transactions supposing objects are not replicated. Rule R2 ensures one-copy equivalence for the set of all objects. Legality of query transactions is ensured by the underlying protocol implementing causal consistency.

- Rule R1.
  Two update transactions are conflicting if one of them writes into an object that is read or written by the other. Rule R1 requires all conflicting update transactions be ordered. This can be easily accomplished by requiring every update transaction acquires first a read (or write) token for the object it wants to read (or write)[6]. A read (respt. write) token is a shared (respt. exclusive) lock: such locks can easily be implemented by read and write quorums [28, 9].

[6] Actually, if there are no query transactions, Rule R1 implements linearizability [11] on a set of non duplicated objects. Linearizability is a consistency criterion more constraining than serializability [4]; it is equivalent to *strict serializability*.

- Rule R2.
  This rule ensures one-copy equivalence for the set of all objects. It requires dissemination of *update* messages be consistent, *i.e.*, all *update* messages be delivered to every process in the same total order and without violating causal consistency. This can be realized by using a broadcast primitive that sends messages to all processes (including their senders) and delivers messages to processes in the same total order. The Isis ABCAST primitive provides such a delivery property[7]. This broadcast primitive is used by the underlying protocol (described in Section 4.1) that has to be modified accordingly. An update transaction executed by process P_i finishes when P_i is delivered the corresponding *update* message it sent with the ABCAST primitive (a similar protocol, called *fast-read*, is used in [4] to implement sequential consistency).

It is important to note that, due to the underlying causal consistency layer, query transactions are not constrained by acquisition of read tokens, and so are wait-free as in previous protocols. Moreover, let us note that classical lock-based protocols implementing serializability require that any (query or update) transaction first obtains appropriate read or write tokens (quorums) for the objects it wants to access. With this stronger synchronization, vectors are no more necessary to track causality relation, version numbers associated with objects are sufficient [6]. This shows there is a tradeoff between synchronization imposed on an execution and size of control information necessary to track causal dependencies: the more synchronized is the execution, the less control information is necessary.

4.4 Discussion

Many implementations are possible for the three consistency criteria. The ones that we have developed in the previous sections demonstrate the important differences between the various criteria and also show why causal consistency and causal serializability allow higher availability compared to serializability. Causal consistency allows both query and update transactions to be completed locally and hence in a wait-free manner. Wait-free implementations of query transactions are also possible with causal serializability and serializability. However, the synchronization imposed on update transactions distinguishes these consistency criteria from each other as

[7] As noted in the "group multicast" literature, such a protocol is more expensive than a protocol implementing causal delivery of messages; in his thesis [26], Schmuck showed that causal order protocols need basically one phase while total order protocols require two phases in asynchronous distributed systems.

well as from causal consistency. Only write accesses on each object need to be synchronized in causal serializability whereas both read and write accesses issued by update transactions must incur such synchronization in serializable executions. In transactions in which large number of objects are read but only few are updated, the performance of causal serializable transactions can be significantly better compared to their execution with the serializability consistency criterion.

If we consider each process has a vote for every object x, owning a read or a write token amounts to get a sufficient number of votes. Let r_x (respt. w_x) the number of votes a process has to obtain in order to read (respt. write) x. Table 1 depicts the number of votes required by a read (respt. write) issued by a transaction in the three previous protocols (n is the number of processes).

number of votes required to access object x	protocol ensuring causal consistency	protocol ensuring causal serializability	protocol ensuring serializability
read by a query	$r_x = 0$	$r_x = 0$	$r_x = 0$
read by an update	$r_x = 0$	$r_x = 0$	$r_x + w_x > n$
write by an update	$w_x = 0$	$w_x \geq (n+1)/2$	$w_x \geq (n+1)/2$

Table 1: Synchronization Cost of Protocols

5 Fault-tolerance

5.1 Fault Model

Without loss of generality, we assume a one-to-one mapping between processors and processes. We consider the following failure model. Communication channels are asynchronous and reliable. Asynchrony means transfer delays are arbitrary (as are process speeds) and reliability means that a message sent by a process P_i is eventually received by its destination process P_j if P_i and P_j are correct (*i.e.*, do not fail)[8]. Processes fail by crashing, *i.e.*, by prematurely halting. A non-crashed process follows its assigned protocol.

5.2 Causal Consistency

When implementing causal consistency, crash failures can give rise to the two following problems:

[8] Reliable channels are implemented by retransmitting lost or corrupted messages and by repairing link failures.

- If a process crashes during the broadcast of an *update* message, it is possible that some non-crashed processes receive this message while some others do not.
- If the system becomes partitioned, then the *update* messages broadcast in a partition cannot be delivered to processes of an other partition.

Dissemination of *update* messages The *uniform reliable broadcast* primitive, as described by Hadzilacos and Toueg in [12], provides a solution to the first problem. This primitive has the following specification:

- Validity: if a correct process broadcasts a message m, then all correct processes eventually deliver m.
- Uniform agreement: if a process delivers a message m, then all correct processes eventually deliver m.
- Uniform integrity: any message m is delivered to a process at most once, and only if it has been broadcast by some process.

It is important to see that if a process fails during the broadcast of an update message, two outcomes are possible: either all non-crashed processes deliver it or none of them.

A protocol implementing such a broadcast primitive is described in [12]. Basically, when it broadcasts a message, a process sends it to all its neighbors; when a process receives a message for the first time, it propagates it to its neighbors and only then processes the message. In case of no partition, this simple protocol implements a uniform reliable broadcast.

Remark. It is possible to incorporate the causal delivery of *update* messages (necessary to implement causal consistency) within this broadcast primitive by adding the following requirement to the three previous properties:

- Causal delivery: if the broadcast of m precedes causally [13] the broadcast of m', then no process delivers m' unless it has previously delivered m.

This constitutes the specification of the *uniform causal broadcast* [12].

Partitions As indicated, if the system suffers from partitioning, all messages broadcast in one partition cannot be delivered to processes belonging to another

partition. It is important to realize that partitioning cannot create violation of causal delivery of messages as messages sent in one partition are not causally related to messages sent in another one. The only problem that has to be solved is to ensure that messages delivered to processes of a partition that are not crashed when this partition merges with another one, are to be delivered to non-crashed processes of the other partition. This requires the saving of delivered messages till partition merging. The memory size necessary to keep copies of delivered messages till partition merging can be bounded by limiting the number of broadcasts that can occur in a partition [25].

5.3 Causal Serializability

Implementation of causal consistency amounts to implement an appropriate protocol to disseminate information. Implementation of causal serializability, additionally, requires to synchronize processes when they access shared variables. As we have seen previously, tokens have been introduced to realize these synchronizations.

Let us first observe that only update transactions entail such a synchronization. So, crash of a process does not affect query transactions. Consequently, the problem occurs when update transactions need a token owned by a crashed process.

Token-based protocols Several token protocols resilient to processors failures have been proposed [2, 17]. These protocols detect loss of the token and recover the state of the lost token. It is important to note that these protocols do not use an underlying election protocol. Moreover, when the system suffers from partitioning, they guarantee that there is at most one copy of the token. These protocols assume that each processor has, associated with it, a stable storage and that channels are FIFO. Finally, it is important to note that these protocols rely on time-out mechanisms and in some sense assume a "synchronous" model of computation.

Asynchronous systems If the system is totally asynchronous, it can be augmented with a failure detector [7]. A failure detector is an entity attached to a process that informs it about the set of other processes it suspects to have crashed; it is perfect if it does not make mistakes when it does a suspicion. It has been shown in [25] that perfect failure detectors are necessary to solve resource (here token) allocation problems. Not perfect failure detectors can suspect the owner of the token to have crashed while it has not; this is because even if they run a consensus protocol [7], each one proposing as consensus value the set of processes it suspects, they can mistakenly agree on the crash of the owner of the token. As perfect failure

detectors cannot be implemented in an unreliable asynchronous system[9], it follows that, in real systems such as Isis [5], a majority of processes that agree on the crash of some other process exclude it from the system (if the excluded process has not crashed, it has to execute a *join* procedure to enter again the system). It follows that, in a totally asynchronous system, the liveness property (an update transaction will eventually be executed) cannot be ensured as a previously excluded processor joining the system can be excluded again if it is still suspected to be crashed.

6 Related Work

Many systems have advocated consistency criteria weaker than serializability. These include database systems where performance of query transactions is improved by allowing them to access objects even when such accesses are not serializable. Examples of systems that allow weak consistency include read-only transactions [10], epsilon serializability [21] and many others. In these systems, update transactions must still be serializable. In contrast, causal consistency and causal serializability criteria allow concurrent update transactions to be ordered differently in the views of different processes.

Weaker consistency has also been explored in distributed shared memory (DSM) systems and distributed file systems. Although many consistency criteria exist for DSM systems, they are defined at the level of memory operations[10] while we define them at the level of transactions. In file systems such as Coda [16], disconnected operation due to mobility can lead to weak consistency. The Coda *isolation-only-transactions* (IOT) are related to our work. However, IOT's are validated to be serializable. Our consistency criteria can also be applied to IOTs. The Bayou system [27] addresses the problem of consistency between replicated servers and provides various levels of consistency via session guarantees. The issues related to transactions that include accesses to multiple objects are not directly addressed.

Several protocols have been proposed to manage distributed dictionaries [29, 15]. It is worth noting that all these protocols actually implement variants of causal consistency at the level of dictionary operations rather than transactions.

[9] As shown in [7], their implementation would contradict the impossibility result of distributed consensus with one faulty process [8].

[10] Many memory models consider explicit synchronization operations in defining the consistency model. In our system, the necessary synchronization (if any) is implemented by the system.

7 Conclusion

This paper has presented consistency criteria weaker than serializability for processes sharing objects through transactions. These criteria are based on a *read-from* relation on transactions. While serializability guarantees all processes have the same sequential view of the execution, causal consistency only ensures that every process, taken individually, has a consistent view of the execution (so two processes can have distinct views of the execution). Causal serializability adds to causal consistency the guarantee that all updates on each object are seen in the same order by all processes. These criteria have been formally defined and protocols implementing them have been introduced. These protocols demonstrate the high availability and improved performance that is possible with causal consistency and causal serializability. The paper has addressed fault-tolerance problems in the case of crash failures.

References

[1] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Trans. on Programming Languages and Systems*, 15(1):182–205, 1993.

[2] D. Agrawal and A. El Abbadi. A Token-based Fault-tolerant Distributed Mutual Exclusion Algorithm. *Journal of Parallel and Distributed Computing*, 24:164–176, 1995.

[3] M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. *Distributed Computing*, 9:37-49, 1995.

[4] H. Attiya and J. L. Welch. Sequential Consistency versus Linearizability. *ACM Trans. on Computer Systems*, 12(2):91–122, May 1994.

[5] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, (9, 3):272–314, 1991.

[6] P.A. Bernstein, V. Hadzilacos and N. Goodman. Concurrency Control and Recovery in Database Systems. *Addison-Wesley*, 370 pages, 1987.

[7] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Proc. 10th ACM Symposium on Principles of Distributed Computing*, Montreal, pp. 325–340, 1991. (To appear in the Journal of the ACM, 1996).

[8] M.J. Fischer, N. Lynch and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985.

[9] H. Garcia-Molina and D. Barbara. How to Assign Votes in a Distributed System. *Journal of the ACM*, 32(4):841–850, 1985.

[10] H. Garcia-Molina and G. Wiederhold. Read-only Transactions in a Distributed Database. *ACM Trans. on Database Systems*, 7(2):209–234, June 1982.

[11] M. Herlihy and J. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant Broadcasts and Related Problems. *Chapter 5, Distributed Systems (2nd Edition), S. Mullender Ed., Addison-Wesley and ACM Press*, 1993, pp. 97–145.

[13] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[14] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C28(9):690–691, 1979.

[15] R. Ladin, B. Liskov, L. Shrira and S. Ghemawat. Providing Availability using Lazy Replication. *ACM Trans. on Computer Systems*, 10:4, pp. 360–392, 1992.

[16] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review*, 28(2):81–87, April 1994.

[17] D. Manivannan and M. Singhal. A Decentralized Token Generation Scheme for Token-Based Mutual Exclusion Algorithms. *Int'l Journal of Computer Systems: Science and Engineering*, 11(1):45–54, January 1996.

[18] F. Mattern. Virtual Time and Global States in Distributed Systems. *Proc. Int. Conf. on Parallel and Distributed Computing*, (Cosnard, Quinton, Raynal and Robert Eds), North-Holland, pp. 215-226, 1988.

[19] J. Misra. Axioms for Memory Access in Asynchronous Hardware Systems. *ACM Trans. on Programming Languages and Systems*, 8(1):142–153, 1986.

[20] M. Mizuno, M. Raynal, and J.Z. Zhou. Sequential Consistency in Distributed Systems. *Proc. Int. Workshop "Theory and Practice in Dist. Systems"*, Dagstuhl, Germany, Springer-Verlag LNCS 938, (K. Birman, F. Mattern and A. Schiper Eds), pp.227-241, 1994.

[21] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *ACM Special Interest Group on Management of Database*, May 1991.

[22] M. Raynal. A Distributed Solution to the k-out of-m Resources Allocation Problem. *Proc. Int. Conf. ICCI*, Ottawa, Springer-Verlag LNCS 497, pp. 407-412, 1991.

[23] M. Raynal, A. Schiper and S. Toueg. The Causal Ordering Abstraction and A Simple Way to Implement it. *Information Processing Letters*, 39:343–350, 1991.

[24] M. Raynal and A. Schiper. From Causal Consistency to Sequential Consistency in Shared Memory Systems. *Proc. 15th Int. Conf. FST&TCS (Foundations of Software Technology and Theoretical Computer Science)*, Bangalore, India, Springer-Verlag LNCS 1026, (P.S. Thiagarajan Ed.), pp. 180-194, dec. 1995.

[25] A. Ricciardi. Perfect Failures Detectors and (repeated) Reliable Broadcast. *Proc. 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, May 1996, to appear. (Also Tech. Report TR-PDS-1995-016, University of Texas at Austin).

[26] F. Schmuck The Use of Efficient Broadcast in Asynchronous Distributed Systems. *Ph. D. Thesis*, Cornell University, TR88-928, 124 pages, 1988.

[27] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proc. of the Third Int. Conf. on Parallel and Distributed Data Systems*, Austin, Texas, pp. 140–149, Sept. 1994.

[28] R.H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copies Databases. *ACM Trans. on Database Systems*, 4(2):180–209, 1979.

[29] Wuu G.T. and Bernstein A.J. Efficient Solutions to the Replicated Log and Dictionary Problems. *Proc 3rd ACM Symposium on Principles of Distributed Computing*, pp. 233-242, 1984.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655 avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e 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