Static and Dynamic Adaptation of Transactional Consistency

Oliver Theel, Michel Raynal

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Static and Dynamic Adaptation of Transactional Consistency

Oliver Theel and Michel Raynal

N ° 2999

Octobre 1996

THÈME 1

ISSN 0249-6399

Static and Dynamic Adaptation of Transactional Consistency

Oliver Theel*and Michel Raynal

Thème 1 — Réseaux et systèmes
Projet Adp

Rapport de recherche n°2999 — Octobre 1996 — 30 pages

Abstract: Consistency criteria adopted for the management of persistent replicated objects in a distributed system define the degree of concurrency allowed among operations accessing objects. Several notions of consistency are known from the literature, among them are causal consistency, causal serializability, and serializability. In this paper, we propose a generalizing algorithm for concurrency control in a transaction system that exhibits a clean separation between policy and mechanism. A consistency criterion selected is manifested as a set of rules forming the policy. The mechanism, however, remains unchanged regardless of the currently used policy. The mechanism implements causally consistent message delivery and uses tokens and quorums of tokens to enforce access operation ordering according to the specified consistency criterion. Since a policy is implemented as a set of rules, switching on-the-fly from one consistency criterion to another one can easily be done whenever changes in access patterns or cost/availability requirements suggest a modification. An example of an application exploiting the advantages of switching among various consistency criteria concludes the paper.

Key-words: Distributed Systems, Distributed Transactions, Transactional Consistency, Causal Consistency, Causal Serializability, Serializability, Token-based Algorithms, Quorum Schemes.

(Résumé : tsvp)

* The author is supported by the Basic Research Action Program of the European Union, HCM project, under contract No. 3702 "CaberNet."

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

Mécanismes et Politiques de Cohérence pour la Gestion des Données Réparties

Résumé : Depuis la sérialisabilité jusqu'à la cohérence causale, plusieurs critères de cohérence sont envisageables pour les données gérées dans les systèmes répartis. Le choix d'un critère particulier relève à la fois de la nature des données et de l'application qui les manipule.

Cet article propose une approche modulaire pour mettre en oeuvre toute une famille de critères de cohérence. La première partie définit un ensemble de mécanismes généraux à partir desquels peuvent facilement être implantés des critères particuliers. Ces mécanismes sont fondés sur des jetons et des vecteurs de version. La seconde partie montre comment un critère de cohérence particulier peut être defini par un ensemble de règles de synchronisation. L'interprétation de ces règles par les mécanismes précédents offre alors une mise en oeuvre particulière du critère concerné. Un avantage de l'approche proposée réside dans la possibilité de changer statiquement ou dynamiquement le critère de cohérence. L'intérét de telles modifications du critère de cohérence est illustré à l'aide d'un exemple pertinent (positionnement et controle d'un véhicule en présence d'obstacles).

Mots-clé : Systèmes distribués, Transactions distribués, Cohérence des transactions, Cohérence causale, Sérialisabilité causale, Sérialisabilité, Algorithmes à jetons, Quorums

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 must be synchronized in order to ensure that they read correct values (i.e., satisfy the consistency criterion). Serializability and two-phase-locking, 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* [13], *serializability* [3], and *linearizability* [8]) 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 the first part of this paper, we give an introduction to weaker consistency criteria, namely causal consistency and causal serializability, in which the *causality relation* between read and write operations on shared objects plays a central role. In the second part, a framework implementing various consistencies, including weak consistency criteria, is presented. Weak consistency criteria reveal to be sufficient to match data consistency requirements of a class of applications. 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. A consistency criterion that is weaker than serializability and stronger than causal consistency can be defined to deal with such concurrent updates: causal serializability ensures 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 suppor-

ted 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 dependencies 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 that there is a unique last value of each object for all processes, at the end of the computation.

The main emphasis of this paper, however, is to present implementations that put the introduced consistency criteria to work. Contrary to presenting different protocols for each consistency criteria, we propose a *framework-based* approach. This framework exhibits a clean separation of *mechanism* and *policy*, i.e., a basic algorithm used for implementing all consistency criteria and a consistency criterion-specific setup enabling the algorithm to adopt a behavior that enforces a particular consistency criterion. The advantages offered by a framework-based approach as depicted above are various. First, through this framework, it becomes very clear, what different consistency criteria have in common and what separates them. Base concepts unspecific to any single consistency criterion but essential for implementing a range of criteria are identified and incorporated into the general algorithm. Consistency criteria-specific properties are singled out and presented as policies. Second, since a consistency criterion is represented as a policy within the framework that is not hard-coded, it can easily be modified on an application-to-application basis. Changing the consistency does not require modifying a single line of code.

Thus, it is possible to dynamically alter the consistency criterion currently used according to the requirements of applications. Third, even during the runtime of a single application, dynamic changes of the currently used consistency criterion can be exploited. And forth, the framework-based approach is open in the sense that new consistency notions yet to be defined can easily be added.

The framework presented in this paper assumes an environment where copies of shared objects are maintained at each node where they are accessed. This is not an integral assumption. The framework can easily be extended to handle only partially replicated objects, but for the sake of easy presentation, we assume full replication.

This paper consists of four main sections. Section 2 describes the shared object model and gives formal definitions for serializability, causal consistency, and causal serializability. Then, Section 3 defines a general framework for implementing the consistency criteria; tokens and vector clocks constitute the basic tools exploited by the framework. Section 4 gives implementations of the consistency criteria, based on the introduced framework and suited policies. It is stated how the framework can be used with a static consistency notion as well as with a notion of consistency that dynamically changes while the system is running. A sample application showing the usefulness of dynamic consistency adaptation is given in Section 5. Section 6 concludes the paper. It states the main achievements presented in this paper and sketches our current and future research.

2 Shared Objects Model

2.1 System Model

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 $W(t)$ are called *read* and *write set*. They denote the set of objects read and written, respectively, by transaction t. If $W(t) = \emptyset$, transaction t is called *query*; if $W(t) \neq \emptyset$, 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 $(h_i, \rightarrow_i)$.

Definition 2.1 (Execution History) An *execution history* (or simply a history) of a shared objects system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that:

- $H = \bigcup h_i$
- $t1 \rightarrow_H t2$ 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)
  - (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 $\neg(t2 \rightarrow_H t1)$.

2.2 Consistency of Shared Objects

This section defines three consistency criteria for shared objects accessed by processes through transactions, namely serializability, causal consistency, and causal serializability. Their definitions are based on the *legality* concept.

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

2.2.1 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 2.2 (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)

□

2.2.2 Serializability

Serializability is the classical consistency criterion for database transactions[2][3]. Informally, serializability expresses the fact that a history $\widehat{H}$ must be equivalent to some sequential execution of the same set of transactions in order to be consistent. Formally, it is defined in the following way.

Definition 2.3 (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 $\widehat{H1}$ given in 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.

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

[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 arrows with large heads and *read-from* edges by arrows with small ones. Additional edges that are due to transitivity are not indicated.

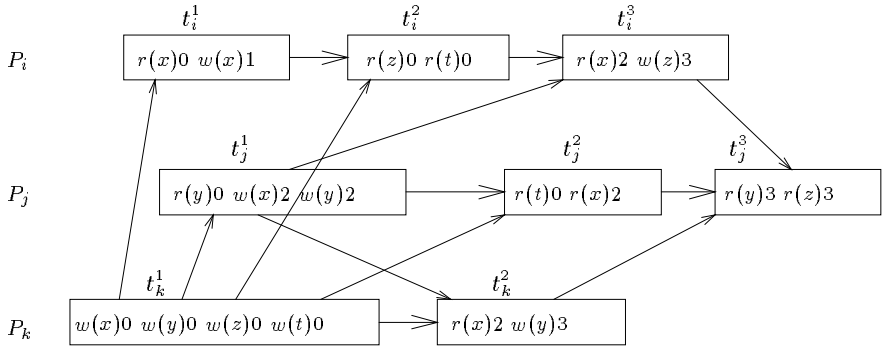

Figure 1: *A serializable history* $\widehat{H1}$

2.2.3 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 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 2.4 (Causal Consistency [16]) 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 $\widehat{H2}$ (see 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.

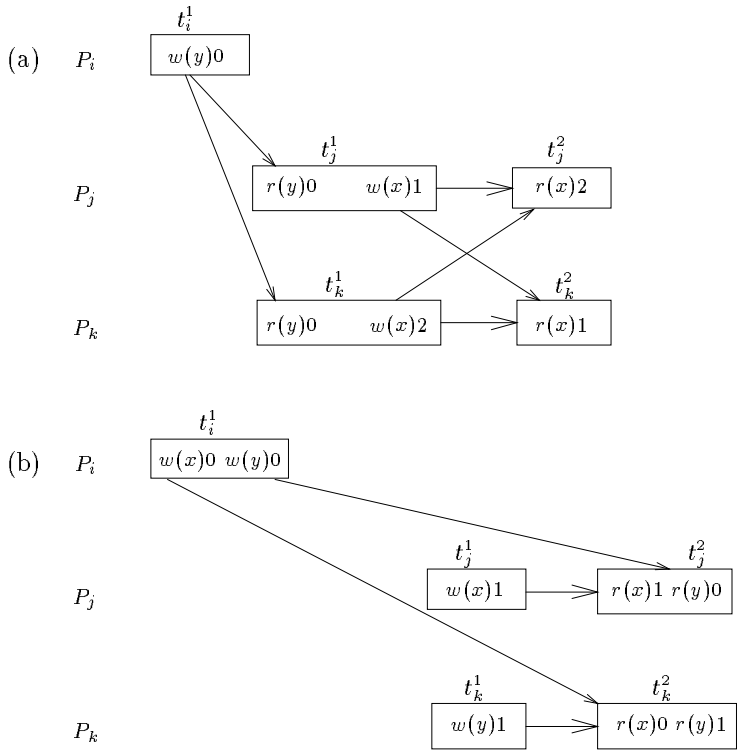

Figure 2: *(a) A causally consistent history* $\widehat{H2}$ *and (b) a causally serializable history* $\widehat{H3}$

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 (i.e., no process has hidden interaction with the other processes) cannot know whether the execution is serializable or only causally consistent.

2.2.4 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 2.5 (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 $\widehat{H3}$ (see 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 $\widehat{S3_k}$.

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 $\widehat{S2}_k$).

3 A Framework for Implementing Consistencies

To implement consistency criteria, one solution is to design a specific protocol for each criterion. Another solution, presented in this section, consists of developing a unifying framework from which the protocols can be modularly built. Such an approach exhibits a clean separation of mechanism and policy. It uses a single algorithm with varying "setups" (or *policies*) when implementing particular consistency criterion. The major advantage offered by such a separation is that the policy can be modified "on-the-fly," resulting in an adaptation of the consistency criterion used for transactions. This allows a dynamic switching between consistency criteria even while the system is running. Various applications can potentially benefit from this flexibility, among them are collaborative editing applications, support of disconnected operations for mobile users, and inventory control applications (e.g., a supermarket chain that uses the system with concurrent updates/queries during the opening hours but wants a consistent view for the inventory in the evening).

The description of the framework is done in two parts. First, the unifying algorithm is explained (Section 3.1). Then, in the second part, the format in which to specify a particular policy is stated (Section 3.2).

3.1 Basic Tools

Based on a reliable message-passing system, causal consistency for transactions can be implemented by using an adaptation of vector clocks [11], namely *version vectors*. It is used to force causally consistent delivery of messages to the sites' processes. Synchronization according to a particular consistency criterion is achieved through the management of *tokens*.

Version Vectors

Every site S_i maintains a version vector $vt^i[1,\ldots,m]$ where m is the total number of sites. An entry $vt^i[k]$, $1 \leq k \leq m$, represents site S_i's knowledge concerning the number of update transaction performed at site S_k.

Tokens
The copy of object a residing on site S_i has an associated token T_a^i. The site S_i is called the *home site* of token T_a^i. A Token T_a^i is a data structure made of three fields:

- the name a of the object it is associated with
- the owner's identity i
- an m-dimensional vector, denoted by $T_a^i[1, \ldots, m]$

Two tokens associated with copies of the same object but with different home sites are said to be *equal* if their vectors are identical. If all tokens of an object are equal then they are called *mutually consistent*. Initially, sites holding copies of objects also hold the corresponding tokens that are mutually consistent.

Basic Framework
The basic idea is the following one. Prior to any execution of an update transaction, tokens must be acquired. The number and identity of tokens required for a particular transaction t and a particular consistency criterion is specified by two sets $ReadSync(t)$ and $WriteSync(t)$. $ReadSync(t)$ is called *read synchronization set (of transaction t)* and $WriteSync(t)$ analogously *write synchronization set (of transaction t)*. When referring to both sets, we call them simply *synchronization sets*. As we will see later, synchronization sets have a strong relationship with transactional read and write sets [3], $R(t)$ and $W(t)$, respectively. The algorithm executed by the site issuing the transaction is described in Figure 3. First, according to the currently used consistency and based on the transaction's read and write set $R(t)$ and $W(t)$, read and write synchronization sets are derived (lines 2–3). How this is precisely achieved will be explained later on in this section. After all tokens specified by $ReadSync(t)$ and $WriteSync(t)$ have been obtained by the issuing site (lines 4–7), the transaction can be executed.[5] Thereby, we rely upon a token delivery honoring causal consistency [2] which is guaranteed by the algorithm (see Figure 4). At the end of the update transaction, tokens are updated and sent back to their home sites.[6] (lines 15–18). Only tokens of the $WriteSync(t)$ set are altered, tokens included in $ReadSync(t) \setminus WriteSync(t)$ remain unchanged. Altering a token means copying the

[5] Tokens can be requested by broadcasting [7] or though the use of hints and forwarding techniques [1].

[6] Within the release operation, alternative token handling strategies can be pursued. See the subsequent paragraph for more details.

```
01:  — algorithm is executed at site S_l
02:  < determine consistency notion to be used >
03:  < calculate sychronization sets >
04:  forall token T ∈ (ReadSync(t) ∪ WriteSync(t)) do
05:      < request token T >
06:  od
07:  < wait until all requested tokens have been delivered>
08:  ∀x ∈ R(t) : <read the value of x_i >
09:  < execute computation defined by the transaction >
10:  if (W(t) ≠ ∅) then
11:      ∀y ∈ W(t) :< update y with new value v^y >
12:      vt^l[l] := vt^l[l] + 1
13:      < multicast updates update(l, vt^l, {(y, v^y) : ∀y ∈ W(t)}) to copy holders >
14:  fi
15:  forall token T ∈ (ReadSync(t) ∪ WriteSync(t)) do
16:      if (token T ∈ WriteSync(t)) then T := vt^l fi
17:      < release token T >
18:  od
```

Figure 3: *Algorithm executed at a transaction processing site*

values of the version vector of the transaction processing site S_l into the vector part of the token (line 16). This reflects the fact that sites which have sent their tokens to the transaction processing site will honor the changes done by the execution of the current transaction as well as any update transaction previously seen by site S_l. Furthermore, due to the causal delivery of tokens, it is impossible for any other site that subsequently acquires one of those updated tokens to disregard the effects of this update transaction.

Figure 4 describes the causal delivery mechanism for update messages and tokens. It is important to note that tokens are delivered to token requesting processes only after causally related update messages have been delivered. This mechanism is quite powerful, as we will see later. The algorithm particularly enforces causal update

```
01:  — algorithm is executed at site S_l
02:  on receiving update(j, vt, S) do
03:      < delay processing of the message until ((vt^l[j] + 1 = vt[j]) ∧ (∀k ≠ j : vt^l[k] ≥ vt[k])) >
04:      vt^l[j] := vt^l[j] + 1
05:      ∀(y, v^y) ∈ S : y_l := v^y
06:  od
07:  on receiving token T do
08:      < delay processing of token T until (∀k : vt^l[k] ≥ T[k]) >
09:  od
```

Figure 4: *Algorithm for processing incoming messages at a site*

message delivery with respect to those tokens that have observed the most recent update transaction. Let us consider the following particular case. If always all tokens associated with copies of an object must be acquired prior to any update transaction of this particular object then all tokens are mutually consistent. They all carry the same information in their vectors, indicating that all of them have "witnessed" the same set of updates. Note that requiring to acquire all tokens of an object prior to its modification might be too costly in some situations. Furthermore, availability issues might quest for a more relaxed token acquiring policy. A potential solution is to only acquire the majority of tokens [18]. Thus, only a majority of tokens observe the update transaction and obtains updated vectors. Nevertheless, a subsequent update transaction is required to honor the previously executed update transaction. Using the algorithm stated above, this constraint is guaranteed, since within any majority of tokens there is at least one token that has observed the last update transaction. It is such a token that delays the execution of the new update transaction until the update messages of the former transaction have been delivered and applied to the objects. Particular care must be taken when dynamically switching between different consistencies. It must be guaranteed that a switch from one consistency to another one is done in a coordinated fashion by all transaction managers in order to prevent illegal accesses to some objects. How consistent switches can be performed by the use of object tokens is described later in the paper.

Token Fusion and Diffusion
As explained above, a token T_a^i can be regarded as a permission granted by the token's home site S_i to the current token holder (i.e., the site where the token currently resides) to perform operations on object a. If object a is replicated then this permission implies to read or update the particular replica of a at the home site. Consequently, when some transaction wants to modify a replicated object it may be necessary – depending upon the consistency criterion adopted – that multiple tokens of the same object but of different replicas must be acquired prior to processing the transaction. After such a transaction has finished (either successfully or unsuccessfully), the question arises of what to do with the tokens that are now located at the transaction processing site. Here, several approaches are possible. The tokens could be 1) sent back to their home site, 2) sent back to the sites they were acquired from, or 3) simple left where they currently are. It is even possible to "fuse" together multiple tokens of the same object and sent the resulting "super-token" within a single message to subsequent token-requesting transactions. Contrary to that, previously fused tokens could be "diffused," i.e., cut into smaller

pieces or even into single "basic" tokens and be disseminated one by one thereafter. It is interesting to note that the strategy adopted for the "granularity" management of tokens does not have an impact on the consistency of transactions, but on the efficiency of the system. If objects are exclusively managed using causal consistency then it is advantageous to permanently locate the tokens at their home sites. This way, tokens are always locally available without the need of requesting tokens and deliver them via the communication network. If, on the other hand, accesses to an object must be serializable (i.e., serializability is used as correctness criterion) then fusing together a sufficiently large number of tokens might reduce the communication overhead. Instead of requesting and receiving a certain number of basic tokens through multiple messages, all the transaction must acquire is a single fused super-token.

Whereas for a static use of the framework, i.e., using a single consistency criterion, an optimal token dissemination strategy can quite easily be identified and configured, token dissemination in the dynamic case is more complicated. Here, periods adopting different consistency criteria alternate. Consequently, a dynamic token dissemination management must be in place, trying to optimize token fusion and diffusion, as well as token placement. For the scope of this paper, we do not stress token dissemination any further. We simple assume that appropriate management is available, following the basic principles discussed above. Once again, we would like to emphasize that an adopted consistency is guaranteed solely by the interplay of the framework's policy and mechanism. It is not impacted by the token dissemination strategy chosen.

Detection and Regeneration of Lost Tokens

Since the availability of object tokens is a prerequisite for the accessibility of objects through the transaction system, special care must be taken to ensure that tokens are not lost due to communication and site failures, including network partitions. The loss of tokens must be detected and eventually new tokens generated [14, 12, 15]. A standard technique for the regeneration of tokens is through election algorithms [17]. In [1] an algorithm is presented that exploits the notion of logical time to detect the loss of tokens and to recover them together with token-specific status information. This approach eliminates the need of executing time- and communication cost-expensive election protocols. It can be adapted for efficiently solving the detection and regeneration problem of object tokens in the context of this paper, in particular, since our approach is also based on the concept of logical time. However,

within this paper, we do not further stress this problem due to its generality. We merely assume that an adequate mechanism is available.

3.2 Synchronization Rules

Additionally to the stated mechanism, a *policy* is needed. Such a policy is expressed through a set of rules. The rules specify how the synchronization sets $ReadSync(t)$ and $WriteSync(t)$ must be derived on a per-transaction basis. Typically, all transaction managers use the same set of rules at a given point in time, although they are not necessarily using the same set at two different times. Furthermore, we assume that a single consistency is used by the transaction managers.[7]

Informally, the rules describe which tokens must be acquired prior to executing transaction t. Thus, if for example $T_a^i \in ReadSync(t)$ holds then token T_a^i must first be acquired by the transaction processing site prior to the start of the transaction. If T_a^i is included in $WriteSync(t)$ then the token must also be obtained first but is additionally modified once the transaction is committed as shown by the algorithm in Figure 3. In order to facilitate the description of which tokens are included in these sets, we use the following definitions.

Definition 3.1 (Q_k-sets of an Object) Let a be an object of the distributed system with n replicas and T_a^i, $i = 1, \ldots, n$ are the associated tokens. Furthermore, $1 \leq k \leq n$ holds. A set $Q_k(a) \subseteq \{a^1, \ldots, a^n\}$ with $\mid Q_k(a) \mid = k$ is then called a *Q_k-set of (object) a.* □

Definition 3.2 (Rules) Let $\mathcal{C}$ and $\mathcal{E}$ be boolean expressions. $\mathcal{C} \rightarrow \mathcal{E}$ is called a *rule*. A rule is said to be *satisfied* if the formula "$\mathcal{C}$ implies $\mathcal{E}$" is true. □

For example, a typical rule describing which tokens must be collected depending upon a given read set $R(t)$ might look as follows:

$$(a \in R(t)) \rightarrow (Q_k(a) \subseteq ReadSync(t)) \tag{1}$$

Satisfying this rule means that if object a is contained in the readset of transaction t then at least k different tokens of object a must be included in the read synchronization set of t. We call a rule specifying the read (write) synchronization set *read (write) synchronization set rule*. When referring to both rules, we call them *synchronization set rules*. As we will show in the next section, different consistencies can be implemented within the framework, simply by using different synchronization set rules but the same algorithm.

[7] The approach can easily be generalized in order to support different notions of consistency on a per-application basis or for sets of objects.

4 Using the Framework

In this section, we present synchronization set rules for three consistencies, namely causal consistency, causal serializability, and serializability. Additionally, we describe how consistencies can dynamically be changed.

4.1 Causal Consistency

Causal consistency in a static framework setup is straightforwardly achieved. In fact, requesting any token prior to the execution of a transaction is not necessary at all due to the fact that the interplay of local version vectors together with the message delivery mechanism already guarantees causal consistency. Thus, the synchronization set rules are very simple.

Rules 4.1 (Synchronization Set Rules for Causal Consistency)

$$true \quad \rightarrow \quad (\emptyset \subseteq ReadSync(t)) \tag{2}$$

$$true \quad \rightarrow \quad (\emptyset \subseteq WriteSync(t)) \tag{3}$$

□

Rules (2) and (3) indicate that independent of the particular read and write sets of a transaction t, the synchronization sets $ReadSync(t)$ and $WriteSync(t)$ do not need to contain any token:[8] On top of a communication mechanism that already guarantees causal consistent (update) message delivery, no additional means need to be taken for enforcing causal consistency. It is easy to derive that the interplay of rules and interpreting mechanism leads to wait-free update transactions assuming that tokens and replicas are always locally available. If a token is not locally available, then it must be issued via the communication network, thus, delays may occur. In either case, applying the above rules for calculating the synchronization sets of a transaction, once the read and write sets have been identified, leads to a causally consistent transactional behavior.

4.2 Causal Serializability

Whereas causal consistency allows two processes a different perception of the effects of two concurrent update transactions modifying a particular object, causal

[8] This is reflected by the synchronization set rules since the empty set is always a subset of any set.

serializability enforces the same sequential perception of those update effects at all processes. Causal serializability is in this sense stronger a consistency criterion than causal consistency. This fact is reflected in our approach by a "stronger" synchronization rule set for causal serializability, meaning intuitively that by using fixed read and write sets stronger rules lead to synchronization sets having in total a higher cardinality (i.e., more tokens per object must be acquired prior to accessing the object).

Rules 4.2 (Synchronization Set Rules for Causal Serializability)

$$true \quad \rightarrow \quad (\emptyset \subseteq ReadSync(t)) \tag{4}$$

$$(a \in W(t)) \quad \rightarrow \quad (Majority(a) \subseteq WriteSync(t)) \tag{5}$$

with $Majority(a) := Q_m(a)$ and

$$m > n/2 \tag{6}$$

□

Rule (4) specifies a read synchronization set that does not need to contain any token. Rule (5) requires a majority of tokens of an object to be included in the write synchronization set if the corresponding object is included in $W(t)$. The fact that at least a majority of tokens of an object must be acquired guarantees, that there is at most a single update transaction that can proceed since at most one site can hold a majority of tokens at any given time. Therefore, generally speaking, modifying an object under causal serializability constraints leads to non-wait-free update transactions. Depending on the token dissemination strategy and the transaction workload of the entire distributed system, update transactions can be wait-free. If, for example, a particular site issues a series of update transaction without other sites questing for tokens then at least the second and all subsequent update transactions in a row are wait-free, assuming that the required tokens remain at the site and are not re-distributed.

4.3 Serializability

For enforcing serializability, both, query and update transactions must be synchronized in such a way that queries observe the modifications done by the most recent update transaction with respect to a total ordering among the transactions as defined by a common view. A total ordering can be achieved by preventing two update transactions to be executed concurrently. Additionally, no update transaction is allowed whenever query transactions are ongoing [3]. Read and write locks together

with two-phase-locking are well-known techniques to enforce such a behavior for non-replicated objects.

Enforcing Serializability through Votes, Version Numbers, and Quorums
In case of replicated objects, votes, version numbers, and read/write quorums together with two constraints with respect to read and write quorums, enforce a corresponding notion of consistency, called *one-copy-serializability* [3]. Here, a vote, a version number, and a replica are always co-located at a site. Prior to accessing a replicated object in a one-copy-serializable-compliant fashion, a read quorum of votes (or write quorum of votes depending on the operation) must be obtained. A site having a replica tries to grant its vote to a requester by first locking the replica in read (write) mode. If this succeeds (i.e., the replica is not already locked in a conflicting mode) then the value of the local replica together with its vote value and version number value is sent to the requester (the objects themselves remain at the site). Since at least the number of votes indicated by the read (write) quorum must be collected, the requester might contact several sites hosting replicas. Once the required quorum has been obtained, the requester can perform the operation. For a read access, the most recent value of the object can be derived through the version numbers: using the value of any replica exhibiting a maximal version number among those replicas contacted yields a correct result. In case of writing the replicated object, all contacted replicas must be atomically modified. Additionally, the version number of all modified replicas must be atomically set to a value higher than the maximal version number contained in the set of contacted replicas. In any case, all locks held by this transaction are finally released. The constraints imposed on the read quorum r and write quorum w are as follows. First, no concurrent read and write accesses are allowed, thus, $r + w > n$ with n being the total number of replicas and therefore sites in the distributed system. Second, no two concurrent write accesses should be possible, thus, $w > n/2$ [6]. A large number of data replication protocols known from the literature use the above algorithm for ensuring serializability of replicated objects [5, 4, 9, 19].

Enforcing Serializability through Tokens, Version Vectors, and Synchronization Sets
Within our framework, though, the vote of an object (or replica) a at site S_i is represented by its token T_a^i. Granting a vote therefore corresponds to sending the token to the requester. Satisfying a quorum (read or write quorum depending on the operation) finds its counterpart in the acquisition of all tokens that are inclu-

ded in the corresponding read or write synchronization set prior to performing the operation itself. Deriving the most recent value of the replicated object in case of a read access is simply achieved by reading the value of the local replica within the transaction. At this time, it is already guaranteed that the local replica is up-to-date. This is due to the fact that any token also has a version vector part. This version vector part can in fact be regarded as a logical vector timestamp. Since a token is withheld in a message buffer until all causally related update messages have been received and applied to the local object (see Figure 4), values of the local object are at least reflecting these modifications that acquired tokens of this particular replicated object have observed.[9] If the synchronization sets are defined in such a way that read and write synchronization sets as well as two write synchronization sets of an object always have a non-empty intersection then the value of the object is always up-to-date, once the issued operation accesses the object's value within the transaction. In case of an object update, the tokens included in the write synchronization set are tagged with the same version vector as the update messages for the object. Due to the intersection property of a write synchronization set with any other synchronization set, subsequent reads or writes to the object are delayed until theses update messages have been received and the changes have been applied to the local replica. Synchronization rules enforcing the intersection property are given below.

Rules 4.3 (Synchronization Set Rules for Serializability)

$$(a \in R(t)) \quad \rightarrow \quad (ReadQuorum(a) \subseteq ReadSync(t)) \tag{7}$$

$$(a \in W(t)) \quad \rightarrow \quad (WriteQuorum(a) \subseteq WriteSync(t)) \tag{8}$$

with $ReadQuorum(a) := Q_r(a)$, $WriteQuorum(a) := Q_w(a)$ and

$$r + w \quad > \quad n \tag{9}$$

$$w \quad > \quad n/2 \tag{10}$$

□

4.4 Switching between Consistencies

As already stated, one major advantage offered by the framework is to change the consistency on-the-fly, i.e., while the system is running. The main difficulty that

[9] A token is said to "observe an operation" if the token has been acquired for an operation (read or write). It is therefore physically present at the transaction processing site. In case of a modification of the associated object, the token is also modified prior to its release (see Figure 4).

must be overcome in this context is to guarantee a *coordinated* switch from one consistency to another one. Since consistencies are represented as synchronization set rules, changing the consistency consequently means exchanging one synchronization rule set against another one. Switching among consistencies can therefore be regarded as an update operation of some meta data used by transaction managers. There are several alternatives for the maintenance of synchronization rule sets.

Non-Replicated Synchronization Rule Sets
One possibility is to maintain a single, non-replicated object that stores the synchronization rule set. This object must be read-locked and read by a transaction manager prior to the execution of any transaction. In case of a consistency switch, the object has to be write-locked and modified. Since the object is not replicated, switching operations do not require the interaction of multiple transaction managers. Unfortunately, because a synchronization rule set object represents a centralized resource essential for the functioning of the transaction system, this approach has several drawbacks: First, the availability of the transaction system is limited by the availability of the synchronization rule set object. Even a single site failure can render the object unaccessible, leading to a situation where no transactions can be processed any more. Second, since the synchronization rule set object must be contacted by transaction managers prior to any transaction, it constitutes a severe performance bottleneck.

Replicated Synchronization Rule Sets
An second solution is to store a copy of an object containing the synchronization rule set locally at the transaction managers. Prior to calculating the synchronization sets for a transaction, the transaction manager simply read-locks and reads the local replica thereby deriving the current synchronization rules. A modification of this object (and therefore the consistency enforced by the transaction system) requires the participation of all transaction managers in order to consistently modify all replicas of synchronization rule sets. This approach overcomes the availability and performance bottleneck described above for the non-replicated case. It favorites normal operations, i.e., the processing of transactions, against exceptional operations like consistency switches. Due to the full replication and therefore the local accessibility of synchronization rule sets, no additional communication costs occur during transaction processing. For the much more seldom case of consistency switches, relatively high communication costs arise due to the fact that all copies must be updated. Additionally, the switch operation exhibits a relatively low availability. This behavior

must unfortunately be accepted in order to avoid an increase of communication costs and a decrease of availability for the processing of user transactions.

Eager and Lazy Switches
In our framework we use the latter approach which is based on a fully replicated synchronization rule set object. Figure 5 depicts the switches among the three consistencies defined earlier in the paper. The ovals represent the different consis-

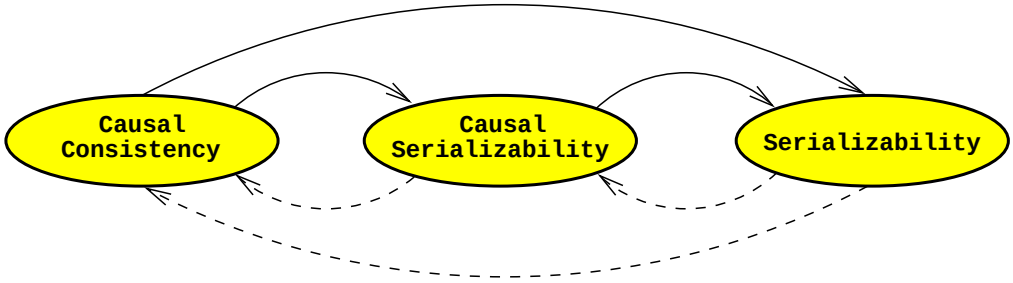

Figure 5: *Transitions between consistencies*

tencies. The arrows indicate a possible switch. For instance, the arrow starting at state `Causal Serializability` and ending at `Serializability` indicates a potential consistency switch from causal serializability to serializability. The type of arrow (solid or dashed) indicates the method used for performing the switch. In case of switching from a weaker consistency to a stronger one (transitions indicated by solid arrows), all synchronization rule sets copies must be eagerly updated: prior to the processing of any subsequent transaction, all synchronization rule set copies must already have been modified. For example when switching from causal serializability to serializability it must be guaranteed that all subsequent transactions are executed in a serializability-compliant fashion. Otherwise, consistency might be violated. Contrary to that, switching from a stronger to a weaker consistency notion (indicated by dashed transitions in the figure), eager updates of the synchronization rule set replicas are not compulsory. It suffices without sacrificing consistency to lazily reflect the changes [10]. This is because stronger consistencies cannot violate weaker ones. For example executing a transaction under causal serializability although a switch from causal serializability to causal consistency has already been initiated, does not result in a violation of causal consistency.

5 An Example

We assume a scenario where a vehicle moves in a two-dimensional space (called *exploration space* hereafter). As an example, one might imagine a semi-autonomous exploration vehicle on a planetary surface. Two remote sensors independently monitor the movement of the vehicle, one sensor the vehicle's movement along the x-dimension, the other sensor along the y-dimension. These sensors could be placed within the exploration space itself, e.g., at base camps or they are located at navigation satellites orbiting above the exploration space. An observer wants to monitor the vehicle's movement within the exploration space. This observer can be thought of as a control center or the vehicle itself. Generally, the observer does not need to have precise tracking data of the vehicle. It suffices to have some fuzzy knowledge about the vehicle's current position.[10] Within the exploration space, there exist an *a priori* known area that should be avoided by the vehicle by all means. One might think of an area presenting a threat to the vehicle or vice versa.

A possible solution to this monitor and control problem is the following one. The two sensor devices and the observer are each modeled as independent processes within a distributed system. An estimate of the current position of the vehicle is maintained in a shared object p that consists of two fields $p.x$ and $p.y$ storing the most recently observed x- and y-position. p is fully replicated, i.e., one replica is located at the sensor for the x-dimension, a second replica resides at the y-dimension sensor, and a third replica, finally, is maintained at the observer. Generally, the object is maintained under causal consistency constraints. However, when the vehicle approaches the restricted area, consistency switches are performed. In a first stage, a consistency switch from causal consistency to causal serializability is issued. If the vehicle continues to approach the particular area, a second switch from causal serializability to serializability is performed. Contrarily, when the vehicle moves away from the restricted area, these switches are undone. Thus, when the consistency has been serializability – reflecting the fact that the vehicle was quite close to the restricted area – a switch from serializability to causal serializability is performed. When moving even further away, then the consistency is finally altered to causal consistency. The idea behind these consistency switches is to more precisely monitor the movement of the vehicle when it approaches the dangerous zone. This offers the opportunity to be able to intervene in the vehicle's navigation if the vehicle is indeed about to enter the particular area.

[10] If the observer is the vehicle itself then there is a trade-off between precise tracking data and telecommunication costs such as battery power. The vehicle may want to sacrifice a certain degree of position correctness for saving energy.

Figure 6 shows the exploration space. The circle labeled with `A` represents the

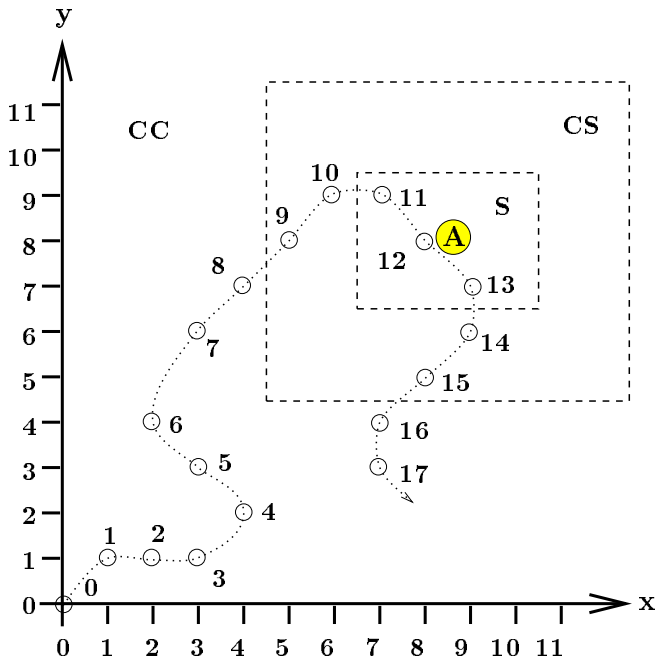

Figure 6: *The route of a vehicle through an exploration space*

restricted area. As a default, causal consistency, labeled `CC`, is used. While being in the area labeled with `CS`, causal serializability serves as consistency criterion. Finally, within the `S`-labeled area, serializability is used. The dotted line indicates the movement of the vehicle over a certain time period. Starting at the origin, the vehicle moves towards point 1, then point 2 etc. The points 0–17 indicate points in time and space where the sensors have measured the vehicle's position. For example, at point 3, the x-sensor has measured an x-coordinate of 3 and the y-sensor a y-coordinate of 1. Once measured, the sensor process updates the corresponding field of the p object with the new value. In this example, the x-sensor issues a transaction under causal consistency that sets $p.x = 3$. The y-sensor process, accordingly, issues a transaction using the same consistency notion for changing $p.y$ to 1. When either the x-sensor or the y-sensor recognizes an entry into a different "consistency area," then a consistency switch operation is executed. If the vehicle is about to enter an area with a stronger consistency than the currently used one, then the switch

is done first, followed by the update of the position. If the vehicle moves into an area with a weaker consistency then the position is updated first, followed by the corresponding consistency switch. The two sensor processes measure the vehicle's position at about the same time. Although, it cannot be predicted, which update transaction is processed first. Furthermore, the observer knows about the frequency with which p is updated, e.g., every five seconds.[11] We assume for the ease of presentation that the query transaction done by the observer at a certain time point is performed after the corresponding position updates have been executed. Figure 7 shows the update and query transactions as well as the consistency switches issued when the vehicle moves along the indicated path. $w(p.x)2$ in column 2 of line 5 (at time 2), for example, describes an update transaction issued by the x-sensor process at time point 2 that modifies the x field of object p to 2. Since the vehicle is in the CC-labeled area, this transaction is done under causal consistency constraints. An observer issuing a query immediately after point 2 can eventually observe a vehicle position of $p.x = 1$ and $p.y = 1$ as indicated by the entry in column four of line 6 (i.e., $r(p)1, 1$). This is due to the fact that, under causal consistency constraints, the query can correctly be processed by the transaction system without the need to block while causally unrelated updates are still to be received and applied to the observer's local replica (such as the update message for changing $p.x$ to 2 that originated from the x-server at time 2). The fifth column gives the consistency which is used for executing the related transactions. At time 9, for example, a consistency switch from causal consistency to causal serializability is performed. The last two columns of the table indicate whether the observed position is a real on, i.e., whether the vehicle has ever been present at this location (real?=yes). If a position is a real one then the last column informs whether the position is the current position of the vehicle, i.e., timely?=yes (at the time of the most recent update transaction).[12] A graphical presentation of the observed vehicle positions are given in Figure 8. In this figure, both, the true route of the vehicle (dotted line) as well as the observed route (solid line) are given. The points 0–17 represent the observed positions of the vehicle. They are not necessarily real locations in contrast to the corresponding points of Figure 6. It can particularly be derived that while using causal consistency, the observed route is an approximation of the real route taken by the vehicle. The vehicle was, for example, never present at the location given by point 5. Point 3, on the contrary, is a real and timely location. When using causal serializability,

[11] If more precise tracking data is needed, even in strong consistency areas, then this frequency must be increased.

[12] If a position is not real then it can never be timely, thus real?=no implies timely?=no.

time	x-sensor	y-sensor	observer	consistency	real?	timely?
0	$w(p.x)0$	$w(p.y)0$		CC		
			$r(p)0,0$	CC	yes	yes
1	$w(p.x)1$	$w(p.y)1$		CC		
			$r(p)1,1$	CC	yes	yes
2	$w(p.x)2$	$w(p.y)1$		CC		
			$r(p)1,1$	CC	yes	no
3	$w(p.x)3$	$w(p.y)1$		CC		
			$r(p)3,1$	CC	yes	yes
4	$w(p.x)4$	$w(p.y)2$		CC		
			$r(p)3,2$	CC	no	no
5	$w(p.x)3$	$w(p.y)3$		CC		
			$r(p)4,3$	CC	no	no
6	$w(p.x)2$	$w(p.y)4$		CC		
			$r(p)2,3$	CC	no	no
7	$w(p.x)3$	$w(p.y)6$		CC		
			$r(p)3,6$	CC	yes	yes
8	$w(p.x)4$	$w(p.y)7$		CC		
			$r(p)4,6$	CC	no	no
9	$w(p.x)5$	$w(p.y)8$		CC→CS		
			$r(p)4,7$	CS	yes	no
10	$w(p.x)6$	$w(p.y)9$		CS		
			$r(p)5,8$	CS	yes	no
11	$w(p.x)7$	$w(p.y)9$		CS→S		
			$r(p)7,9$	S	yes	yes
12	$w(p.x)8$	$w(p.y)8$		S		
			$r(p)8,8$	S	yes	yes
13	$w(p.x)9$	$w(p.y)7$		S		
			$r(p)9,7$	S	yes	yes
14	$w(p.x)9$	$w(p.y)6$		S→CS		
			$r(p)9,6$	CS	yes	yes
15	$w(p.x)8$	$w(p.y)5$		CS		
			$r(p)9,6$	CS	yes	no
16	$w(p.x)7$	$w(p.y)4$		CS→CC		
			$r(p)8,5$	CC	yes	no
17	$w(p.x)7$	$w(p.y)3$		CC		
			$r(p)7,4$	CC	yes	no

Figure 7: *Update and query transactions together with switching operations issued while the vehicle is moving*

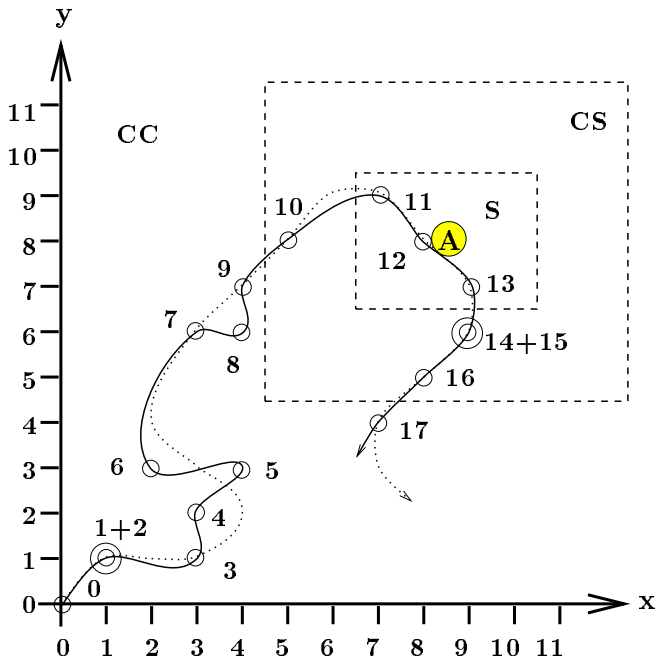

Figure 8: *The route of the vehicle as it can be monitored by an observer*

the real route of the vehicle can be observed, although the vehicles position at a certain point in time is not necessarily correct but merely an approximation. For example, point 10 is a real location, although it has been occupied by the vehicle at time 9. Finally, when using serializability, the real route of the vehicle is observed and correct information about the vehicle's position at the time of the last update transaction is given. Points 11, 12, and 13 serve as examples for those locations.

6 Conclusion and Future Work

In this paper, we introduced weak consistency criteria that are based on a causality relation among the read and write operations on persistent, shared objects. Causal consistency, being one of the weaker consistencies, guarantees that the user will always be able to observe all causally preceding updates of an object. The second weak consistency, causal serializability, extends this guarantee: it additionally ensures that there will be a single final value for the object at the end of a sequence of updates, thus preventing diverging modifications to the object. After stating formal definitions for causal consistency and causal serializability as well as for serializability, we presented a framework for the implementation of these transactional consistencies. The framework is cleanly separated into a mechanism and a policy. The mechanism, using base concepts like version vectors and tokens, can be used to implement all the consistencies alike. The policy, on the contrary, is consistency criterion-specific. A policy is formally described as a set of rules. Rules define the number of object tokens that have to be acquired by the transaction manager prior to the execution of the transaction. Since the policy can be changed without modifying the underlying mechanism, switching among consistencies is possible even while the transaction system is running. This gives the opportunity to exploit specific access patterns or varying demands of object availability and operation costs.

Our future work focuses on communicating information about the currently used consistency notion through the object tokens themselves. This is expected to result in an increased availability of the switching operation. Furthermore, we plan to develop adaptable token dissemination strategies that control the placement and granularity of tokens based on the observed workload and the currently used consistency, thereby reducing communication costs and increasing the system's throughput.

References

[1] 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.

[2] 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.

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

[4] S. Y. Cheung, M. Ahamad, and M. H. Ammar. The Grid Protocol: A High Performance Scheme for Maintaining Replicated Data. In *Proc. of the 6th International Conference on Data Engineering*, pages 438–445, February 1990.

[5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in Partitioned Networks. *Computing Surveys*, 17(3), 1985.

[6] D. K. Gifford. Weighted Voting for Replicated Data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[7] J. M. Hélary, A. Mostefaoui, and M. Raynal. A General Scheme for Token- and Tree-Based Mutual Exclusion Algorithms. *IEEE Trans. on Parallel and Distributed Systems*, 5(11):1185–1196, November 1994.

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

[9] A. Kumar. Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data. *IEEE Transactions on Computers*, 40(9):996–1004, 1991.

[10] R. Ladin, B. Liskov, and L. Shira. A Technique for Constructing Highly Available Services. *Algorithmica*, 3(2):393–420, 1988.

[11] F. Mattern. Virtual Time and Global States in Distributed Systems. In Cosnard, Quinton, Raynal, and Roberts, editors, *Proc. of the International Conference on Parallel and Distributed Computing*, pages 215–226. North-Holland, 1988.

[12] J. Misra. Detecting Termination of Distributed Computations Using Markers. In *Proc. of the 2nd Symposium on Principles of Distributed Computing*, pages 290–294. ACM, August 1983.

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

[14] M. Naimi and M. Trehel. How to Detect a Failure and Regenerate the Token in the log n Distributed Algorithm for Mutual Exclusion. In *Proc. of the 2nd International Workshop on Distributed Algorithms*, volume 312, pages 155–166, July 1987.

[15] S. Nishio, K. F. Li, and E. G. Manning. A Resilient Mutual Exclusion Algorithm for Computer Networks. *IEEE Transactions on Parallel Distributed Information Systems*, 1(3):344–355, July 1990.

[16] M. Raynal, G. Thia-Kime, and M. Ahamad. From Serializable to Causal Transactions (Brief Announcement). In *Proc. of the 5th Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, USA*, page 310. ACM, May 1996.

[17] A. Silberschatz, J. Peterson, and P. Galvin. *Operating Systems Concepts.* Addison-Wesley, Reading, MA, Fourth edition, 1993.

[18] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–207, 1979.

[19] C. Wu and C. G. Belford. The Triangular Lattice Protocol: A Highly Fault Tolerant and Highly Efficient Protocol for Replicated Data. In *Proc. of the 11rd Symposium on Reliability in Distributed Software and Database Systems.* IEEE, 1992.

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