Lifetime Based Consistency Protocols for Distributed Objects†

Francisco J. Torres-Rojas[1], Mustaque Ahamad[1] and Michel Raynal[2]

[1]College of Computing
Georgia Institute of Technology, USA
{torres, mustaq}@cc.gatech.edu
[2]IRISA, France
raynal@irisa.fr

Abstract. Techniques such as replication and caching of objects that implement distributed services lead to consistency problems that must be addressed. We explore new consistency protocols based on the notion of object value *lifetimes*. By keeping track of the lifetimes of the values stored in shared objects (i.e., the time interval that goes from the writing of a value until the latest time when this value is known to be valid), it is possible to check the *mutual consistency* of a set of related objects cached at a site. Initially, this technique is presented assuming the presence of physical clocks. Later, these clocks are replaced by *vector* clocks and then by *plausible* clocks. Lifetimes based on such clocks result in weaker consistency but do provide more efficient implementations.

1 Introduction

The popularity of the Internet and WWW has motivated new applications that allow widely distributed users to share and manipulate complex and rich information. For example, applications such as virtual or cyber communities that span large number of geographically distributed users are already being developed. As these applications scale, we get a shared virtual world that can be navigated by users from any part of the world. A system that implements such a virtual world will store and provide access to a huge number of objects that will model the features of the physical world. Other objects will correspond to users who will interact with objects in the virtual world as well as with each other. This application scenario leads to a widely distributed computing system that will store and provide access to a massive number of constantly evolving information objects across widely distributed users. Most of the objects that capture features of the world require just read-only access but others, that model users and active features of the physical world, will change. Furthermore, although a large number of objects may exist in the distributed system, a particular user or group of users will only access a very small fraction of the object space.

In this paper, we address a problem that is motivated by an application and system environment as described above. We assume that a user, modeled as a *computation*,

†. This work was supported in part by an INRIA/NSF grant and by NSF grant CDA-9501637 and CCR-9619371.

accesses n objects from an object space that consists of m objects ($n << m$). An access to an object may only require the reading of its state (e.g., representation of a graphical object is read for its rendering) or it may also result in an update to the state of the object. To ensure that a user interacts with a coherent view of the virtual world, it is necessary that a computation accesses a *consistent* set of object copies. Such consistency has to be ensured in a system environment where object replication and caching are employed to meet the interactive response time needs of the applications. In particular, a user may cache a local copy of an object to minimize the effects of communication latency on the access time of this object. Such a copy could be received from a number of servers where the object is replicated.

A number of techniques have been previously explored for maintaining the consistency of distributed objects which included ordered group communication [10], replicated data management protocols such as quorum protocols [1], and consistency protocols that have been developed in distributed shared memory systems. We believe that these techniques are inadequate in the application and system environment presented here because of the following reasons:

- Since a user may navigate through a part of virtual space quickly and may not return to it, there is poor locality of access. Thus, the cost of membership operations (to join a group when a user requests an object and delete when its access is complete) may not be amortized over a large number of accesses. If a deletion is not done quickly after the user's focus of attention moves, a user may receive unnecessary updates for objects that will not be locally accessed. Furthermore, although few users may be accessing an object at a given time, the potential number of users can be very large which makes it difficult to manage the data structures (e.g., vector clocks) used in group communication protocols.
- It is reasonable to assume that objects may be owned by certain users and groups who may want to control how and when the objects are modified. Such control can be provided using locks. However, most accesses of non-owner nodes will be read-only. It should be possible to use a nearby copy of an object without acquiring a lock to meet such a request. Many of the distributed shared memory protocols that use synchronization operations to drive consistency operations may not be appropriate when synchronization is not utilized by most object accesses.
- Consistency protocols that require synchronous communication between large number of widely distributed nodes cannot meet the response time needs of interactive applications.

We explore an approach for providing consistent access to related objects to a user[1] in the described environment, that does not suffer from the problems of existing techniques. To motivate our approach, assume that the system has a synchronized global clock T available to all user sites. Consider an operation that assigns value $\mathbf{v}$ to object

1. We do not address synchronous interactions across multiple users which are addressed in groupware applications. Thus, our focus is on consistent access to dynamic information for a given user.

X at time *t*. Furthermore, assume that it is known that object X will be written next at time *t'*. Thus, along with the new updated value of X, the time interval [*t*, *t'*] is also stored. This interval is the *lifetime* of the value **v** of object X. If a computation accesses two related objects X and Y, to ensure that values **v** and **w** of these objects are consistent, we require that the lifetime of **v** does not end before the start of the lifetime of **w** and vice versa. This guarantees that the lifetimes of these values overlap and that they coexisted at the same time, and the user is accessing the state of the virtual world at that time. Several problems arise in the implementation of this approach. Although the lifetime is easily known for write-once objects (they are valid until time *infinity* in the future) and for objects that are updated at regular intervals, in general object lifetimes are not known. Secondly, the assumption of a synchronized global clock for recording lifetimes may not be valid. We avoid this problem by using scalable logical clocks instead of real-time clocks in our implementations. We also develop practical techniques for determining lifetimes of objects.

As we present the consistency protocols based on the lifetime approach, it will become evident that our approach is well suited for scalable implementations of object sharing across widely distributed users. First, a site only needs to ensure consistency of objects that are accessed by local computations and does not need to be directly aware of computations at other sites and their access to objects, unless that a cache miss triggers communications with server sites. Second, it permits considerable flexibility in the choice of replication, caching and update propagation policies (e.g., push vs. pull). Thus, we believe that it is an important approach for meeting the object sharing and consistency requirements of future applications.

In Section **2**, we define the consistency criteria considered in this paper. We describe the lifetime approach in Section **3**. In Section **4**, we present an implementation that assumes the presence of synchronized physical clocks. We substitute the physical clocks by vector clocks and analyze the properties of the new scheme in Section **5**. In order to obtain a more efficient implementation, we replace the inherently non-scalable vector clocks by *plausible clocks* [21] and present some preliminary performance results of these protocols in Section **6**. The paper is concluded in Section **7**.

2 Consistency Criteria

The *global history* $\boldsymbol{H}$ of a distributed system is a partially ordered set of all operations occurring at all sites of the system. We assume that all the operations in $\boldsymbol{H}$ are either **read** or **write**, where a **read** operation retrieves the value of an object and a **write** operation updates an object by assigning it a new value. In order to simplify, it is assumed that each value written is unique. The *local history* $\boldsymbol{H}_i$ of site *i* is the sequence of operations that are executed on this site. If operation **a** occurs before operation **b** in $\boldsymbol{H}_i$ we say that **a** precedes **b** in program order. If $\boldsymbol{D}$ is a set of operations, then *S* is a *serialization* of $\boldsymbol{D}$ if *S* is a linear sequence containing exactly all the operations of $\boldsymbol{D}$ such that each **read** operation to a particular object returns the value written by the most recent (in the order of *S*) **write** operation to that same object.

Sequential Consistency (**SC**) requires that the result of any execution is the same as if the operations of all sites were executed in some sequential order, and the operations of each individual site appear in this sequence in the order specified by its program [15]. Therefore, global history $\boldsymbol{H}$ satisfies **SC** if there is at least one serialization S of $\boldsymbol{H}$ that respects the program order for each site in the system.

The *causality relation* "→" for message passing systems as defined in [14] can be modified to order the operations of $\boldsymbol{H}$. Let **a**,**b**,**c** ∈ $\boldsymbol{H}$. We say that **a** → **b**, i.e., **a** causally precedes **b**, if one of the following holds: (i) **a** and **b** are executed at the same site and **a** is executed before **b**; (ii) **b** reads an object value written by **a**; (iii) **a** → **c** and **c** → **b**. If none of the above conditions holds between two distinct operations **a** and **b**, **a** and **b** are "concurrent". We denote this situation as **a** || **b**.

Let $\boldsymbol{H}_{i+w}$ be the set of all the operations in $\boldsymbol{H}_i$ plus all the **write** operations in $\boldsymbol{H}$. History $\boldsymbol{H}$ satisfies *Causal Consistency* (**CC**) if for each site i there is a serialization S_i of $\boldsymbol{H}_{i+w}$ that respects causal order "→" [3]. This is equivalent to saying that if **a**,**b**,**c** ∈ $\boldsymbol{H}$ are such that **a** writes value **v** in object X, **c** reads the same value **v** from object X, and **b** writes value **v**' in object X, it is never the case that **a** → **b**→ **c**. **CC** requires that all the causally related operations be seen in the same order by all sites, but at the same time allows different sites to perceive concurrent operations in different order. **CC** has been shown to be sufficient for applications that support asynchronous sharing among distributed users. It has been explored both in message passing systems [5] and in shared memory and object systems [2, 4, 11, 12, 13, 19]. Relations between **SC** and **CC** have been studied in [3, 17].

3 Lifetime of Object Values

3.1 General Architecture

We assume a Client/Server architecture where each object has a set of *server* sites that provide long term storage for the object. Other sites must cache a local copy of the object before accessing it. At a given instant, each cache does not contain more than one copy of a given object. Notice, however, that as a result of caching and replication there may exist several different versions of the same object at different sites (either client or server sites) of the system. Each site's cache holds a finite number of object copies. Some kind of replacement algorithm is executed when the cache runs out of room for new objects. Cache misses are solved by communicating with a server site, which either has a copy of the requested object or can obtain it. As it was mentioned before, a **write** operation assigns a particular value to an object copy stored in the local cache. Eventually, this version of the object may be communicated to one or more server sites and from there to other sites in the system.

Let the cache of site i be modeled as a set of objects $\boldsymbol{C}_i$. In the context of a particular site cache there is at most one copy of any object of the system, thus let X_i denote the version of object X currently stored in $\boldsymbol{C}_i$. Similarly, when a cache miss occurs at site i while accessing object X, some server s provides a copy of X_s, i.e., its current version of X. Once that this copy is stored in $\boldsymbol{C}_i$, we denote it X_i.

3.2 Start and Ending Times

The *start time* of X_i, denoted as X_i^{α}, is the time when its corresponding **write** operation occurred. The start time of an object copy allows us to know since when the particular value of the object stored in the cache is valid. The latest time when the value stored in X_i is known to be valid is called its *ending time* and it is denoted as X_i^{ω}. For any object version X_i we have that $X_i^{\alpha} \leq X_i^{\omega}$. The interval $[X_i^{\alpha}, X_i^{\omega}]$ is the *lifetime* of the value stored in X_i.

The concept of lifetime allows us to evaluate the mutual consistency of the object copies stored in a site's cache. Two object copies are mutually consistent if it is possible that their values coexisted at some time in a distributed computation. Thus, the local copies X_i and Y_i are mutually consistent if the lifetime of one object value does not end before the start of the lifetime of the other object value, i.e., if $\mathbf{max}(X_i^{\alpha}, Y_i^{\alpha}) \leq \mathbf{min}(X_i^{\omega}, Y_i^{\omega})$. The cache of site i is consistent if every pair of objects in $\boldsymbol{C}_i$ is mutually consistent, or, equivalently, if the *maximum* start time of any object value in $\boldsymbol{C}_i$ is less than or equal to the *minimum* ending time of any object value in $\boldsymbol{C}_i$.

3.3 Cache Context Time

In general, the protocols presented in this paper induce the intended level of consistency in the global execution by checking the mutual consistency of the objects stored at each site's cache and by invalidating those objects which do not satisfy the consistency criteria. However, when the number of objects which can be held at a cache is less than the total number of objects in the system, it is not enough just to consider the current state of the cache because we are ignoring the mutual consistency with objects that *have been* in the cache previously [13]. For instance, if we don't take into account the past content of a cache with room for just a few objects or in the extreme case exactly one single object (which is always mutually consistent with itself), we may generate executions where an object value moves back and forth between two or more possible values.

In order to avoid any "flickering" of the object values between present and past values and to guarantee the liveness of our protocols, we associate a timestamp called $\mathbf{Context}_i$ to $\boldsymbol{C}_i$. The initial value of this timestamp is 0. When either a copy of object X is brought into $\boldsymbol{C}_i$ (becoming X_i) or when the local version X_i is updated, $\mathbf{Context}_i$ becomes $\mathbf{max}(X_i^{\alpha}, \mathbf{Context}_i)$. In other words, $\mathbf{Context}_i$ is the latest start time of any object value that is stored or that has been stored in $\boldsymbol{C}_i$. Whenever $\mathbf{Context}_i$ is updated all the local copies of objects $Y_i \in \boldsymbol{C}_i$ such that $Y_i^{\omega} < \mathbf{Context}_i$ are invalidated. Similarly, it is required that any object X brought from server s into $\boldsymbol{C}_i$ has an ending time not less than **Context**$_i$, i.e., $X_s^{\omega} \geq$ **Context**$_i$. If $\boldsymbol{C}_i$ is consistent, then site i is accessing the state of the objects at least at time **Context**$_i$.

4 Sequential Consistency Protocol based on Physical Clocks

In this section we explore Protocol **A**, a **SC** protocol based on the concept of lifetime of object values. This protocol guarantees **SC** by ensuring that the contents of the caches of every site are always mutually consistent. We presuppose the existence of a perfectly synchronized physical clock T available to all the sites of the system. We say that $T(\mathbf{a})$ is the real time when operation **a** is executed. It is assumed that $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}_i$, $\mathbf{a} \neq \mathbf{b} \Rightarrow T(\mathbf{a}) \neq T(\mathbf{b})$.

4.1 General Description

If site i has a cache miss when accessing object X, Protocol **A** executes the following steps:

1. It is requested that a server site s sends back a copy of X_s such that $X_s^{\omega} \geq$ **Context**$_i$. This copy of X_s is inserted in $\boldsymbol{C}_i$ and from now on it is denoted as X_i.
2. **Context**$_i$ is updated to **max**(X_i^{α}, **Context**$_i$).
3. All the local copies of objects $Y_i \in \boldsymbol{C}_i$ such that $Y_i^{\omega} <$ **Context**$_i$ are invalidated. This implies that all objects $Y_i \in \boldsymbol{C}_i$ such that (**min**(X_i^{ω}, Y_i^{ω}) < **max**(X_i^{α}, Y_i^{α})) are invalidated.

Since a **write** operation executed at site i establishes a new value for **Context**$_i$ (see Section **3.3**), it is necessary to repeat step **3** on all the objects in $\boldsymbol{C}_i$ after an object value is updated. Notice that it is possible that site i is accessing a consistent state of the virtual world at some point in the past, specifically at time **Context**$_i$ even when the current time is much later than **Context**$_i$ (notice that **Context**$_i$ is never decreased).

If the computation of site i can be modeled as a series of *sessions*, where each one has a well defined set of objects $\boldsymbol{O}$ to be accessed, we may consider an interesting "lazy" variation of Protocol **A**. In this new protocol, we assume the existence of some kind of *begin session* operation that declares the set of objects $\boldsymbol{O}$ associated with the current session. At this point, it is verified that the values of the objects $\boldsymbol{O} \cap \boldsymbol{C}_i$ are consistent, i.e., the set of object values to be accessed by the session that currently reside in the cache must be consistent. Any object $Y_i \in (\boldsymbol{O} \cap \boldsymbol{C}_i)$ such that $Y_i^{\omega} <$ **Context**$_i$ is invalidated. When an object X is brought into the cache, instead of testing it against all the objects $Y_i \in \boldsymbol{C}_i$, we test it just against the objects $Y_i \in (\boldsymbol{O} \cap \boldsymbol{C}_i)$. This protocol can be a more efficient alternative to Protocol **A**, specially if the caches are able to store a large number of objects, many of them unrelated to the current session.

4.2 Estimation of Lifetimes

Let's assume that when a local copy of object X is updated by site *i* at time *t*, the time *t*' when any copy of object X is written next at any site of the system is known beforehand. This knowledge would allow us to establish with precision the lifetime of an object value. For write-once objects, the ending time is trivially known to be *infinity* in the future. Similarly, the ending time for objects that are updated periodically, such as dynamic content web pages with time-to-live (TTL) fields, can be easily estimated. If a site holds a lock with time-out (also known as a fault-tolerant lock or *lease*) on a particular object, the time-out of the lock is a conservative estimate of the ending time of the value of this object.

However, in general, we don't know the lifetime of arbitrary object values. When site *i* updates object version X_i at time *t*, we start by assigning *t* to both X_i^α and X_i^ω. Thus, site *i* only knows the lifetime of this object value in an incomplete way. We have to discover as we go that no object copy X_j ($i \neq j$) has been overwritten and use this information to advance X_i^ω. For this we need a protocol that determines if the object copy X_i is valid at time $t' > X_i^\omega$.

When site *i* requests that a server *s* provides a copy of X such that $X_s^\omega \geq$ **Context**$_i$, it may happen that X_s does not satisfy this condition. In this case, this server communicates with other sites that could potentially have updated X, to determine if X_s is still valid at a time not less than **Context**$_i$. If it is, X_s^ω is advanced. Otherwise, some client or server site *j* has a more recent copy of X such that X_j^ω is not less than **Context**$_i$; in this case X_j is copied into X_s. Eventually, a copy of an appropriate version of X is provided to site *i*. Notice that with this protocol server *s* is effectively advancing X_s^ω and X_i^ω.

It is reasonable to assume that even if many sites read an object, it will only be written by a small number of sites. In the extreme case of a single site that writes an object, communication with this site can be used to advance lifetime. Otherwise, several ways are possible. For instance, server may keep track of last writer, or asynchronous, gossip style communication discovers that an object has not been overwritten and its lifetime is increased.

4.3 Protocol A and Sequential Consistency

If $\mathbf{a} \in \boldsymbol{H}_i$, we say that *Context*(**a**) is the value of **Context**$_i$ when operation **a** is executed, and that *Pos*(**a**) is the relative position of **a** in the sequence $\boldsymbol{H}_i$. Let *L*(**a**) be a structure of the form <*Time*, *Position*> where:

If **a** is a **write** operation then *L*(**a**) = <*T*(**a**), 0>.
If **a** is a **read** operation then *L*(**a**) = <*Context* (**a**), *Pos*(**a**)>.

If $\mathbf{a},\mathbf{b} \in \boldsymbol{H}$ are such that $L(\mathbf{a}) = \langle Time_{\mathbf{a}}, Position_{\mathbf{a}} \rangle$ and $L(\mathbf{b}) = \langle Time_{\mathbf{b}}, Position_{\mathbf{b}} \rangle$, we say that $L(\mathbf{a}) < L(\mathbf{b})$ if:

$(Time_{\mathbf{a}} < Time_{\mathbf{b}}) \vee$
$((Time_{\mathbf{a}} = Time_{\mathbf{b}}) \wedge$
($\mathbf{a}$ and $\mathbf{b}$ are executed at the same site) $\wedge$ $(Position_{\mathbf{a}} < Position_{\mathbf{b}})) \vee$
$((Time_{\mathbf{a}} = Time_{\mathbf{b}}) \wedge$ ($\mathbf{b}$ reads the value written by $\mathbf{a}$))

Otherwise, $L(\mathbf{b}) < L(\mathbf{a})$.

Theorem 1. *Protocol* **A** *generates execution histories that satisfy* **SC**.

Proof. Let the sequence S be the product of sorting all the operations $\mathbf{a} \in \boldsymbol{H}$ by the values of $L(\mathbf{a})$. It is easy to see that S includes all the operations in $\boldsymbol{H}$ and that S respects the program order for all the local histories. Now, we have to prove that each **read** returns the value of the most recent **write** in the order of S.

Let $\mathbf{a},\mathbf{b},\mathbf{c} \in \boldsymbol{H}$ be such that $\mathbf{a}$ writes value $\mathbf{v}$ in object X, $\mathbf{b}$ reads the same value $\mathbf{v}$ from object X and $\mathbf{c}$ writes value $\mathbf{v}$' in object X. Since we assume that each value written is unique, the fact that $\mathbf{b}$ reads the value that $\mathbf{a}$ writes implies that $L(\mathbf{a}) < L(\mathbf{b})$. If $T(\mathbf{c}) < T(\mathbf{a})$, then $L(\mathbf{c}) < L(\mathbf{a}) < L(\mathbf{b})$, i.e., operation $\mathbf{c}$ does not affect the value $\mathbf{v}$ read by operation $\mathbf{b}$.

If $T(\mathbf{a}) \le T(\mathbf{c})$, then, assuming that operation $\mathbf{a}$ is executed at site i, operation $\mathbf{a}$ assigned a value to X_i^{ω} such that $X_i^{\omega} \le T(\mathbf{c})$. This implies that $T(\mathbf{a}) \le Context(\mathbf{b}) \le X_i^{\omega} \le T(\mathbf{c})$ and since $\mathbf{b}$ does not read the value written by $\mathbf{c}$, then $L(\mathbf{a}) < L(\mathbf{b}) < L(\mathbf{c})$. Once again, $\mathbf{c}$ does not affect the value $\mathbf{v}$ read by operation $\mathbf{b}$.

Thus, S is a serialization of the global history $\boldsymbol{H}$ that respects the program order for each site in the system and, in conclusion, Protocol **A** generates an execution that satisfies **SC**. ❑

If the lifetimes of the object values are estimated incompletely, unnecessary invalidations and communications will occur, reducing the efficiency of the system. In the next sections we consider the relaxation of certain assumptions which lead to weaker consistency models, but with more efficient implementations.

5 Causal Consistency Protocol based on Logical Clocks

The assumption of perfectly synchronized physical clocks made by protocol **A** is not realistic in general, specially if we consider widely distributed systems. Furthermore, as it was mentioned in Section **4**, if the ending times of the object values cannot be determined efficiently, the cost of guaranteeing sequential consistency can become high. In this section we consider a protocol that uses logical clocks [18] instead of physical clocks. This change reduces the level of consistency from **SC** to **CC**.

5.1 Background

A *Time Stamping System* (TSS) defines a particular format of timestamp, assigns these timestamps to each operation in $\boldsymbol{H}$ and provides tests for comparing timestamps [21]. Let **a**, **b** $\in \boldsymbol{H}$ and let X be an arbitrary TSS. The timestamps of **a** and **b** are $X(\mathbf{a})$ and $X(\mathbf{b})$, respectively. X reports the causal relationship (not necessarily correct) between two operations by comparing their assigned timestamps, in this way:

$\mathbf{a} \overset{X}{=} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{=} X(\mathbf{b}) \Leftrightarrow X$ "believes" that **a** and **b** are the same operation.
$\mathbf{a} \overset{X}{\rightarrow} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{\rightarrow} X(\mathbf{b}) \Leftrightarrow X$ "believes" that **a** causally precedes **b**.
$\mathbf{a} \overset{X}{\leftarrow} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{\leftarrow} X(\mathbf{b}) \Leftrightarrow X$ "believes" that **b** causally precedes **a**.
$\mathbf{a} \overset{X}{\parallel} \mathbf{b} \Leftrightarrow X(\mathbf{a}) \overset{X}{\parallel} X(\mathbf{b}) \Leftrightarrow X$ "believes" that **a** and **b** are concurrent.

We define TSS V to be the well known *vector clocks* technique [7, 8, 9, 16, 18]. Each site i keeps an integer vector $\mathbf{V}_i$ of N entries (N is number of sites), where $\mathbf{V}_i[j]$ is the knowledge that site i has of the activity at site j. Entry $\mathbf{V}_i[i]$ is incremented each time that an operation is executed at site i. If site i reads an object value whose start time is the vector clock **W**, $\mathbf{V}_i$ becomes the component-wise maximum of $\mathbf{V}_i$ and **W**. $V(\mathbf{a})$ is the value of $\mathbf{V}_i$ when operation **a** is executed at site i. Given **a**, **b** $\in \boldsymbol{H}$, we have the following tests:

$\mathbf{a} \overset{V}{=} \mathbf{b} \Leftrightarrow V(\mathbf{a}) \overset{V}{=} V(\mathbf{b}) \Leftrightarrow \forall\ \mathrm{k}\ V(\mathbf{a})[\mathrm{k}] = V(\mathbf{b})[\mathrm{k}]$
$V(\mathbf{a}) \leq V(\mathbf{b}) \Leftrightarrow \forall\ \mathrm{k}\ V(\mathbf{a})[\mathrm{k}] \leq V(\mathbf{b})[\mathrm{k}]$
$\mathbf{a} \overset{V}{\rightarrow} \mathbf{b} \Leftrightarrow V(\mathbf{a}) \overset{V}{\rightarrow} V(\mathbf{b}) \Leftrightarrow V(\mathbf{a}) \leq V(\mathbf{b})$ and $\exists\mathrm{k}$ such that $V(\mathbf{a})[\mathrm{k}] < V(\mathbf{b})[\mathrm{k}]$
$\mathbf{a} \overset{V}{\parallel} \mathbf{b} \Leftrightarrow V(\mathbf{a}) \overset{V}{\parallel} V(\mathbf{b}) \Leftrightarrow \exists\mathrm{k,j}$ such that $V(\mathbf{a})[\mathrm{k}] < V(\mathbf{b})[\mathrm{k}]$ and $V(\mathbf{a})[\mathrm{j}] > V(\mathbf{b})[\mathrm{j}]$.

Vector clocks characterize completely the causality relation between operations in $\boldsymbol{H}$ [16]. Given vector clocks a and b, we define the functions $\mathbf{max}_V$ and $\mathbf{min}_V$ in this way:

$\mathbf{max}_V(a, b)$ is the vector t such that $\forall\ \mathrm{k}\ t[\mathrm{k}] = \mathbf{max}(a[\mathrm{k}], b[\mathrm{k}])$
$\mathbf{min}_V(a, b)$ is the vector t such that $\forall\ \mathrm{k}\ t[\mathrm{k}] = \mathbf{min}(a[\mathrm{k}], b[\mathrm{k}])$

5.2 Consistency Protocol based on Vector Clocks (Protocol B)

We now propose Protocol **B**, whose purpose is to produce executions that satisfy **CC** by verifying that all the objects stored in a cache are mutually consistent among themselves according to **CC**. This protocol basically follows the same rules of Protocol **A** with the main difference that instead of physical clocks, we use vector clocks (i.e., TSS V) to timestamp the operations and that, since there is not a total order of these timestamps, the tests are modified slightly.

Let **Clock**$_i$ be the current logical time of site i. **Clock**$_i$ is copied to X_i^{α} and X_i^{ω} when the local copy X_i is updated. As before, $\boldsymbol{C}_i$ has an associated timestamp called **Context**$_i$. When object copy X_s is brought from server s into $\boldsymbol{C}_i$ (becoming X_i) or when object X_i is updated at site i, **Context**$_i$ becomes $\mathbf{max}_V(X_i^{\alpha}, \mathbf{Context}_i)$. Several mechanisms, similar to the ones described in Section **4.2**, can be used to update the ending vector time of an object value [12]. Any later start time of a different value of the same object will be after or concurrent with the assigned ending time. For any object copy $X_i \in \boldsymbol{C}_i$ it is true that $(X_i^{\alpha} \xrightarrow{V} X_i^{\omega}) \vee (X_i^{\alpha} \stackrel{V}{=} X_i^{\omega})$.

CC allows that for all objects $X_i \in \boldsymbol{C}_i$, X_i^{ω} advances as the logical time at the site i progresses. Therefore, unlike Protocol **A**, no objects are invalidated as consequence of a local **write** operation. We assume the existence of an asynchronous component responsible for sending updated versions of the objects to the servers. Before sending its copy of X_i back to the server, site i makes $X_i^{\omega} = \mathbf{max}_V(X_i^{\omega}, \mathbf{Clock}_i)$. The server updates its own clock with the information provided by X_i^{α} and updates X_i^{ω} with this clock.

If site i has a cache miss when accessing object X, Protocol **B** executes the following steps:

1. It is requested that a server s sends back a copy of X_s such that $\neg(X_s^{\omega} \xrightarrow{V}$ **Context**$_i$). This copy of X_s is inserted in $\boldsymbol{C}_i$ and from now on it is denoted as X_i.
2. **Context**$_i$ is updated to $\mathbf{max}_V(X_i^{\alpha}, \mathbf{Context}_i)$.
3. $Y_i^{\omega} = \mathbf{max}_V(Y_i^{\omega}, \mathbf{Clock}_i)$ for all the local copies of objects $Y_i^{\omega} \in \boldsymbol{C}_i$.
4. All the local copies of objects $Y_i \in \boldsymbol{C}_i$ such that $\neg(X_i^{\alpha} \stackrel{V}{\parallel} Y_i^{\alpha}) \wedge (\mathbf{min}_V(X_i^{\omega}, Y_i^{\omega}) \xrightarrow{V} \mathbf{max}_V(X_i^{\alpha}, Y_i^{\alpha}))$ are invalidated.

5.3 Protocol B and Causal Consistency

Given two object copies X_i, $Y_i \in \boldsymbol{C}_i$ such that $X_i^{\alpha} \stackrel{V}{\parallel} Y_i^{\alpha}$, it is easy to see that their values are mutually consistent because there exists no operation **o** such that **o** alters either X_i or Y_i and that $X_i^{\alpha} \xrightarrow{V} V(\mathbf{o}) \xrightarrow{V} Y_i^{\alpha}$ or that $Y_i^{\alpha} \xrightarrow{V} V(\mathbf{o}) \xrightarrow{V} X_i^{\alpha}$.

Now, let two object copies X_i, $Y_i \in \boldsymbol{C}_i$ be such that $\neg(X_i^{\alpha} \stackrel{V}{\parallel} Y_i^{\alpha})$, i.e., their start times are causally related. Without loss of generality, let's say that $X_i^{\alpha} \xrightarrow{V} Y_i^{\alpha}$. If the values stored in X_i and Y_i are mutually consistent it has to be the case that $\neg(X_i^{\omega} \xrightarrow{V} Y_i^{\alpha})$, and vice versa.

In conclusion, two object copies X_i, $Y_i \in \boldsymbol{C}_i$ are mutually consistent according to Causal Consistency if $(X_i^{\alpha} \overset{V}{\parallel} Y_i^{\alpha}) \vee \neg(\mathbf{min}_V(X_{\omega}, Y_{\omega}) \xrightarrow{V} \mathbf{max}_V(X_{\alpha}, Y_{\alpha}))$. A cache $\boldsymbol{C}_i$ is consistent if every pair of elements in $\boldsymbol{C}_i$ is mutually consistent. Protocol **B** starts with an empty cache and systematically checks each new cached object against the current contents, invalidating those objects that may compromise the consistency of the cache.

Theorem 2. *Protocol* **B** *generates execution histories that satisfy* **CC**.

Proof. We have to prove that for each site i there is a serialization S_i of $\boldsymbol{H}_{i+w}$ that respects causal order "$\rightarrow$". We start by making $S_i = \boldsymbol{H}_i$. Notice that the sequence $\boldsymbol{H}_i$ respects the causal order "$\rightarrow$".

Let $\mathbf{w} \in \boldsymbol{H}$ be a **write** operation not included in $\boldsymbol{H}_i$ such that $\mathbf{w}$ assigns value $\mathbf{v}$ to an arbitrary object X and that this value is read by operation $\mathbf{a} \in \boldsymbol{H}_i$, Protocol **B** guarantees that $V(\mathbf{w}) \xrightarrow{V} V(\mathbf{a})$ and that there cannot exist another **write** operation $\mathbf{w}' \in \boldsymbol{H}$ that assigns value $\mathbf{v}'$ to object X and $V(\mathbf{w}) \xrightarrow{V} V(\mathbf{w}') \xrightarrow{V} V(\mathbf{a})$, because if this were the case the value $\mathbf{v}$ of object X would have been invalidated by step **3** of Protocol **B**. Therefore, $\mathbf{w}$ can be inserted into its proper position in S_i, i.e., in the rightmost position in S_i such that there is not operation $\mathbf{a}$ in S_i to the left of $\mathbf{w}$ such that $V(\mathbf{w}) \xrightarrow{V} V(\mathbf{a})$.

The rest of the **write** operations in $\boldsymbol{H}$ are either concurrent with the operations inserted in S_i or causally related to some other **write** operation already in S_i. In both cases these operations can be inserted into S_i in the rightmost position such that the causal order is not violated and that there is no interference between a **write** and a **read**. ❑

6 Causal Consistency Protocol based on Plausible Clocks

It is proved in [16] that vector clocks capture completely the causality relation between all the operations in $\boldsymbol{H}$ and therefore they are able to detect all the cases where two operations are concurrent. However, [6] proved that given a distributed system with N sites, causality can only be captured by vector clocks with N entries. If N is large, several problems arise such as growing storage costs, considerable communications overhead and extended processing time. Thus, vector clocks have poor scalability. *Plausible clocks* strive to provide a high level of accuracy in ordering events but they do not guarantee that certain pairs of concurrent events are not ordered. These clocks can be constructed with a constant number of elements independent of the number of sites and yet, under a Client/Server communication pattern, they can decide the causal relationship between arbitrary pairs of operations with an accuracy close to V. Examples of plausible clocks are presented in [20, 21] and it is proved that if TSS P is plausible then $\forall\, \mathbf{a}, \mathbf{b} \in \boldsymbol{H}$:

$$\mathbf{a} \overset{P}{=} \mathbf{b} \Leftrightarrow \mathbf{a} = \mathbf{b}$$
$$\mathbf{a} \xrightarrow{P} \mathbf{b} \Rightarrow (\mathbf{a} \rightarrow \mathbf{b}) \vee (\mathbf{a} \parallel \mathbf{b})$$
$$\mathbf{a} \xleftarrow{P} \mathbf{b} \Rightarrow (\mathbf{a} \leftarrow \mathbf{b}) \vee (\mathbf{a} \parallel \mathbf{b})$$
$$\mathbf{a} \overset{P}{\parallel} \mathbf{b} \Rightarrow \mathbf{a} \parallel \mathbf{b}$$

In general, if v and w are two timestamps generated by P, we say that timestamp t is $\mathbf{max}_P(v, w)$, if all the following conditions hold [22]:

- $v \xrightarrow{P} t \vee v \overset{P}{=} t$
- $w \xrightarrow{P} t \vee w \overset{P}{=} t$
- There is no timestamp u such that $v \xrightarrow{P} u \wedge w \xrightarrow{P} u \wedge u \xrightarrow{P} t$

Similarly, timestamp $t \in \mathbf{S}$ is $\mathbf{min}_P(v, w)$, if:

- $v \xleftarrow{P} t \vee v \overset{P}{=} t$
- $w \xleftarrow{P} t \vee w \overset{P}{=} t$
- There is no timestamp u such that $v \xleftarrow{P} u \wedge w \xleftarrow{P} u \wedge u \xleftarrow{P} t$

We now propose Protocol **C**, which follows exactly the *same rules* of Protocol **B** but, in order to use a logical clock whose size is constant and independent of the number of sites in the system, vector clocks are replaced with a plausible clock P. For instance, if site i has a cache miss when accessing object $\mathtt{X}$, Protocol **C** executes the following steps:

1. It is requested that the server s sends back a copy of $\mathtt{X}_s$ such that $\neg(\mathtt{X}_s^{\omega} \xrightarrow{P} \mathbf{Context}_i)$. This copy of $\mathtt{X}_s$ is inserted in $\boldsymbol{C}_i$ and from now on it is denoted as $\mathtt{X}_i$
2. $\mathbf{Context}_i$ is updated to $\mathbf{max}_P(\mathtt{X}_i^{\alpha}, \mathbf{Context}_i)$.
3. $\mathtt{Y}_i^{\omega} = \mathbf{max}_P(\mathtt{Y}_i^{\omega}, \mathbf{Clock}_i)$ for all the local copies of objects $\mathtt{Y}_i^{\omega} \in \boldsymbol{C}_i$.
4. All the local copies of objects $\mathtt{Y}_i \in \boldsymbol{C}_i$ such that $\neg(\mathtt{X}_i^{\alpha} \overset{P}{\parallel} \mathtt{Y}_i^{\alpha}) \wedge (\mathbf{min}_P(\mathtt{X}_i^{\omega}, \mathtt{Y}_i^{\omega}) \xrightarrow{P} \mathbf{max}_P(\mathtt{X}_i^{\alpha}, \mathtt{Y}_i^{\alpha}))$ are invalidated.

A number of pairs of concurrent timestamps may be reported by P as being ordered, but it can be seen that there is no case where Protocol **C** decides that a set of objects is mutually consistent and that Protocol **B** decides that this same set is *not* mutually consistent. However, for a mutually consistent set, **C** may decide that the set is not consistent and issues unnecessary invalidations of local copies of objects. In any case, since the use of plausible clocks affects only the efficiency of the approach and not its correctness, Protocol **C** induces executions that satisfy Causal Consistency. It is expected that the reduction in the size of the logical clocks and their associated overhead will compensate positively for the unnecessary invalidations and communications with server sites generated by Protocol **C**. Furthermore, under the assumption of a very high number of users sharing a virtual world through the Internet, vector clocks are not feasible.

For purposes of illustration, Table **1** shows some results of a simple simulation of a distributed system where 50 sites sharing 5 objects executed 20 operations each one. We collect the number of messages (either from client to server or vice versa) originated when an object was updated, a cache miss occurred, an object was invalidated or when a server verifies whether an object is still valid up to certain time. The same history was executed under different protocols. The first row shows the results when a sequentially consistent *invalidations scheme* is used (i.e., all the copies of a particular object are invalidated when it is updated at any site). As expected, Protocol **B** generates significantly less messages than Protocol **A**. The number of messages generated by Protocol **C**, which utilizes the plausible clock *REV* [20, 21] with 5 entries per clock, is not much bigger than the number of messages generated by Protocol **B**. However, each one of the 752 messages generated by Protocol **B** must carry a 50 entry vector, i.e., an overhead of 37600 integers, while each one of the 960 messages generated by Protocol **C** must include a 5 entry vector, i.e., an overhead of just 4800 integers.

Table 1. Causal Consistency with Vector Clocks and Plausible Clocks.

	Updates	*Cache Misses*	*Invalidations*	*Validations*	*Total*
Invalidations Scheme	319	1310	1260	0	2890
Protocol **A**	319	1114	771	118	2322
Protocol **B**	319	382	51	0	752
Protocol **C**	319	488	153	0	960

7 Conclusions

We have presented mechanisms to ensure that a set of objects stored in the cache of a site in a distributed system is mutually consistent. The start time of an object value corresponds to the instant when this particular value is written and its ending time indicates the latest time when this value is known to be valid. These two times are updated in such a way that the ending time is always posterior to the start time and previous (or concurrent) to any future update of the object. The lifetime of an object value goes from its start time to its ending time. The basic idea of the presented techniques is to check if the lifetimes of a set of object values overlap, which implies that they are mutually consistent.

Protocol **A** assumes the existence of synchronized physical clocks and generates executions that satisfy Sequential Consistency. Protocol **B** replaces the physical clocks by vector clocks and redefines slightly the rules of Protocol **A**, which requires a less expensive implementation but relaxes the consistency level to Causal Consistency. Finally, Protocol **C** follows exactly the same rules of Protocol **B** but replaces the expensive vector clocks by constant-size plausible clocks, which makes sense if we consider that vector clocks require an entry for each one if the sites in the system. Protocol **C** may execute unnecessary invalidations of local copies of objects, but in the long term this is compensated by the use of shorter timestamps.

Currently we are exploring optimizations to protocols for lifetime estimation and evaluating our mutual consistency techniques under different workloads and applications. At the same time, different families of plausible clocks are being considered. We are interested in characterizing the cases where a "lazy" protocol may be more convenient than guaranteeing the consistency of the complete contents of local caches. This work extends the schemes presented in [2, 4, 12, 13], solving in particular the problem of caches with capacity for just a subset of the shared objects [13] and focusing on implementations with constant-size logical clocks.

References

1 M. Ahamad, M. Ammar and S.Y. Cheung, "Replicated Data Management in Distributed Systems", in *Advances in Distributed Computing: Concepts and Design*, IEEE ComputerSociety Press, T.L. Casavant and M. Singhal eds., 1993.

2 M. Ahamad, F. Torres-Rojas, R. Kordale, J. Singh, S. Smith, "Detecting Mutual Consistency of Shared Objects". Proc. of Intl. Workshop on Mobile Systems and Appl., 1994.

3 M. Ahamad, G. Neiger, J. Burns, P. Kohli and P. Hutto. "Causal memory: definitions, implementation, and programming". Distributed Computing. September 1995.

4 M. Ahamad, S. Bhola, R. Kordale, F. Torres-Rojas. "Scalable Information Sharing in Large Scale Distributed Systems". Proc. of the Seventh SIGOPS Workshop, August 1996.

5 M. Ahamad, M.Raynal, and G. Thiakime, "An adaptive architecture for causally consistent services". Proc. of the 18th IEEE Int. Conf. on Distributed Computer Systems (ICDCS98), pp. 86-95, Amsterdam, 1998.

6 B. Charron-Bost, "Concerning the size of logical clocks in Distributed Systems", Information Processing Letters 39, pp. 11-16. 1991.

7 C.J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering", Proc. 11th Australian Comp.Science Conference, Univ. of Queensland, pp. 55-66, 1988.

8 C.J. Fidge, "Logical Time in Distributed Computing Systems", Computer, vol 24, No. 8, pages 28-33, August 1991.

9 C.J. Fidge, "Fundamentals of Distributed Systems Observation", IEEE Software, vol 13, No. 6, November 1996.

10 U. Friztke, P. Ingels, A. Mostefaoui and M. Raynal, "Fault-Tolerant Total Order Multicast to Asynchronous Groups", Proc. 17th IEEE Symposium on Reliable Distributed Systems (SRDS98), Purdue, October 1998.

11 R. John and M. Ahamad, "Evaluation of Causal Distributed Shared Memory for Data-race-free Programs", Tech. Report, College of Computing, Georgia Institute of Technology, 1991.

12 R. Kordale and M. Ahamad. "A Scalable Technique for Implementing Multiple Consistency Levels for Distributed Objects". Proc. of the 16th. International Conference in Distributed Computing Systems. May 1996.

13 R. Kordale. "System Support for Scalable Services". Ph.D. dissertation, College of Computing, Georgia Institute of Technology. January 1997.

14 L. Lamport, “Time, clocks and the ordering of events in a Distributed System”, Communications of the ACM, vol 21, pp. 558-564, July 1978.

15 L. Lamport, “How to make a multiprocessor computer that correctly executes multiprocess programs”, IEEE Transactions on Computers, C-28(9), 690-691, 1979.

16 F. Mattern, “Virtual Time and Global States in Distributed Systems”, Conf. (Cosnard et al (eds)) Proc. Workshop on Parallel and Distributed Algorithms, Chateau de Bonas, Elsevier, North Holland, pp. 215-226. October 1988.

17 M. Raynal and A. Schiper, “From Causal Consistency to Sequential Consistency in Shared Memory Systems”, Proc. 15th Int. Conference FST & TCS (Foundations of Software Technology and Theoretical Computer Science), Springer-Verlag LNCS 1026, pp. 180-194. Bangalore, India, Dec. 1995.

18 M. Raynal and M. Singhal, "Logical Time: Capturing Causality in Distributed Systems", IEEE Computer, Vol. 29, No. 2, Feb., 1996.

19 M. Raynal and M. Ahamad, “Exploiting write semantics in implementing Partially Replicated Causal Objects”, Proc. of 6th Euromicro Conference on Parallel and Distributed Systems, pp. 175-164, Madrid, January 1998.

20 F. Torres-Rojas, “Efficient Time Representation in Distributed Systems”, MSc. Thesis, College of Computing, Georgia Institute of Technology, 1995.

21 F. Torres-Rojas and Mustaque Ahamad, “Plausible Clocks: Constant Size Logical Clocks for Distributed Systems”, Proc. 10th International Workshop on Distributed Algorithms, WDAG’96. Bologna, Italy, October 1996.

22 F. Torres-Rojas and Mustaque Ahamad. “Computing Minimum and Maximum of Plausible Clocks”, Technical Report, Georgia Institute of Technology, 1998.