Scalable Protocols for Causal Consistency based on Vector Clocks and Plausible Clocks

Francisco J. Torres-Rojas
Instituto Tecnológico de Costa Rica, Escuela de Ingeniería en Computación
COSTA RICA
torres@ic-itcr.ac.cr

Abstract

Causal Consistency is a weak consistency model with very interesting properties that allows efficient implementations. In the same spirit as [20], we use the notion of object value lifetimes (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) to build a scalable and efficient protocol that by checking the mutual consistency of a set of related objects cached at a site induces Causal Consistency on the executions. This technique is presented using only logical clocks (vector clocks initially and plausible clocks later).

Keywords: Distributed Computing, Causal Consistency, Lifetime, Vector Clocks, Plausible Clocks, Distributed Objects.
1. Introduction
In [20], we presented a scalable protocol that induces Sequential Consistency [15] on distributed executions. That protocol (identified as Protocol \( A \) in [20]) ensures that a computation accesses a mutually consistent set of object copies. Such consistency has to be guaranteed 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.

Protocol \( A \) assumes that the system has a synchronized global clock \( T \) available to all user sites. Consider an operation that assigns value \( v \) to object \( X \) 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 \( X \), i.e. the lifetime of an object value goes from its start time, which corresponds to the instant when this particular value is written, until its ending time, which is the latest time when this value is known to be valid. If a computation accesses two related objects \( X \) and \( Y \), to ensure that values \( v \) and \( w \) of these objects are mutually 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. Thus, through the use of synchronized physical clocks, Protocol \( A \) figures out the lifetimes of each object value and then verifies that the set of objects cached at a site are such that their lifetimes overlap. Protocol \( A \) locally invalidates old objects that do not satisfy this requirement, which eventually generates faults that must be solved by a server site. We prove in [20] that this is enough to guarantee that the global executions follow the definition of Sequential Consistency [15]. Besides, since most of the required consistency actions are local, the technique shows great scalability.

However, the assumption of perfectly synchronized clocks made by Protocol \( A \) in [20] is not realistic in general, specially if we consider widely distributed systems. In this paper, we consider a protocol that uses logical vector clocks [18] instead of physical clocks. We call it Protocol \( B \). This protocol induces Causal Consistency [2] on distributed histories. Most of the logic of Protocol \( B \) is similar to the logic of Protocol \( A \), but since we are using logical clocks, there are interesting mathematical properties (or constraints) worthy of analyzing. Besides, since Causal Consistency is able to execute local writes and reads without losing its consistency, this allows less expensive implementations in terms of messages between client and server sites. Later, we briefly propose Protocol \( C \), which is almost identical to Protocol \( B \), with the only difference that instead of using vector clocks, Protocol \( C \) uses plausible clocks, which are scalable clocks independent of the number of sites in the distributed system.

In Section 2, we give general definitions necessary to develop the rest of the paper. The basic concepts of Causal Consistency are explained in Section 3. We present the lifetime approach in Section 4. Section 5 describes the details of Protocol \( B \), an implementation of the technique that uses vector clocks [8, 9, 16, 18] and proves that such protocol induces Causal Consistency on the distributed executions. Protocol \( C \), a variant of Protocol \( B \) that uses plausible clocks instead of vector clocks, is explained in Section 6. The paper is concluded in Section 7.

2. General Definitions

2.1 Distributed Histories and Causality
The global history \( \mathcal{H} \) of a distributed system is a partially ordered set of all operations occurring at all sites of the system. Let \( \mathcal{H}_i \) be the sequence of operations that are executed on site \( i \), also called the local history of site \( i \). In order to simplify, we assume that all the operations in \( \mathcal{H} \) are either read or write, and that each value written is unique, besides, we do not consider speculative executions which allow that a read operation returns a value that has not been written yet. Let \( \mathcal{H}_{i+w} \) be the set of all the operations of \( \mathcal{H}_i \) together with all the write operations in \( \mathcal{H} \).

If \( \mathcal{S} \) is a set of read and write operations, then a serialization of \( \mathcal{S} \) is a linear sequence \( S \) containing exactly all the operations of \( \mathcal{S} \), in such a way that each read operation to a particular object returns the value written by the most recent (in the order of \( S \)) write operation to the same object. Let \( \sim \) be an arbitrary partial order relation defined over \( \mathcal{S} \), we say that serialization \( S \) “respects” \( \sim \) if \( \forall a, b \in \mathcal{S} \) such that \( a \sim b \), it also happens that \( a \) precedes \( b \) in \( S \).

The causality relationship “\( \rightarrow \)” for message passing systems as defined in [14] can be modified to order the operations in
Let \( a, b, c \in \mathcal{H} \). We say that \( a \rightarrow b \), i.e. \( a \) causally precedes \( b \) if one of the following holds:

- \((i)\) \( a \) and \( b \) are executed at the same site and \( a \) precedes \( b \) in program order;
- \((ii)\) \( b \) reads an object value written by \( a \);
- \((iii)\) \( a \rightarrow c \) and \( c \rightarrow b \).

If none of the previous conditions holds between two distinct operations \( a \) and \( b \), we say that \( a \) and \( b \) are concurrent and denote this situation as \( a \parallel b \).

### 2.2 Logical Clocks

The main purpose of logical clocks as defined by Lamport in [14] is to establish the causal relationship between a pair of events or operations in a distributed history. A good survey of logical clocks and their evolution can be found in [5]. A Time Stamping System (TSS) defines a particular format of timestamp, assigns these timestamps to each operation in \( \mathcal{H} \) and provides tests for comparing timestamps [21]. Let \( a, b \in \mathcal{H} \) and let \( X \) be an arbitrary TSS. The timestamps of \( a \) and \( b \) are \( X(a) \) and \( X(b) \), respectively. \( X \) reports the causal relationship (not necessarily correct) between two operations by comparing their assigned timestamps, in this way:

\[
\begin{align*}
& a =_X b \iff X(a) =_X X(b) \iff \text{“believes” that } a \text{ and } b \text{ are the same operation.} \\
& a \rightarrow_X b \iff X(a) \rightarrow_X X(b) \iff \text{“believes” that } a \text{ causally precedes } b. \\
& a \leftarrow_X b \iff X(a) \leftarrow_X X(b) \iff \text{“believes” that } b \text{ causally precedes } a. \\
& a \perp_X b \iff X(a) \perp_X X(b) \iff \text{“believes” that } a \text{ and } b \text{ are concurrent.}
\end{align*}
\]

We define TSS \( V \) to be the well-known vector clocks technique [8, 9, 16, 18]. Each site \( i \) keeps an integer vector \( V_i \) of \( N \) entries (\( N \) is the number of sites), where \( V_i[j] \) is the knowledge that site \( i \) has of the activity at site \( j \). Entry \( V_i[j] \) 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 \( W_i \), \( V_i \) becomes the component-wise maximum of \( W_i \) and \( V_i \). \( V(a) \) is the value of \( V_i \) when operation \( a \) is executed at site \( i \). Given \( a, b \in \mathcal{H} \), we have the following tests:

\[
\begin{align*}
& a =_F b \iff V(a) =_F V(b) \iff \forall k \ V(a)[k] = V(b)[k] \\
& V(a) \leq V(b) \iff \forall k \ V(a)[k] \leq V(b)[k] \\
& a \rightarrow_F b \iff V(a) \rightarrow_F V(b) \iff V(a) \leq V(b) \text{ and } \exists k \text{ such that } V(a)[k] < V(b)[k] \\
& a \parallel_F b \iff V(a) \parallel_F V(b) \iff \exists k, j \text{ such that } V(a)[k] < V(b)[k] \text{ and } V(a)[j] > V(b)[j].
\end{align*}
\]

Vector clocks characterize completely the causality relation between operations in \( \mathcal{H} \) [8, 9, 16, 18]. Therefore, when appropriate we will drop the subindex \( F \) from the operators defined above (i.e., \( \rightarrow_F \) is always equivalent to \( \rightarrow \), and \( \parallel_F \) is always equivalent to \( \parallel \)). Given vector clocks \( t \) and \( v \), we define the functions \( \max_F \) and \( \min_F \) over vector clocks as:

\[
\begin{align*}
& \max_F(t, v) \text{ is the vector } w \text{ such that } \forall k \ w[k] = \max(t[k], v[k]) \\
& \min_F(t, v) \text{ is the vector } w \text{ such that } \forall k \ w[k] = \min(t[k], v[k])
\end{align*}
\]

Vector clocks have the disadvantage of not being scalable since its size depends on the size of the Distributed System. Plausible Clocks [21] are a class of efficient logical clocks that can be represented with a constant number of elements \( i.e., \) independent of the size of the distributed system). However, these clocks can sometimes be wrong about the actual causal relationship between two events in a distributed history. A TSS \( P \) is plausible if \( \forall a, b \in \mathcal{H} \):

\[
\begin{align*}
& a =_P b \iff a = b \\
& a \rightarrow b \Rightarrow a \rightarrow_P b
\end{align*}
\]

### 3. Causal Consistency

The model known as Causal Consistency (CC) was originally defined by Ahamad et al. [2]. It provides a level of consistency weaker than Sequential Consistency as defined by Lamport in [15] and Linearizability as defined by Herlihy
CC considers the causal relationships between operations in a distributed history and it requires that each site perceives its local history together with all the external writes as a serialization that respects causal order as defined in the previous Section. Formally:

**Definition 1.** Global history \( \mathcal{H} \) satisfies CC if for each Site \( i \) there is a serialization \( S_i \) of the set \( \mathcal{H}_{i+w} \) that respects the order “\( \rightarrow \)” [2].

CC requires that all the causally related operations be seen in the same order by all the sites, but at the same time allows different sites to perceive concurrent operations in different orders. 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 [4], and in distributed shared memory (DSM) and object systems [1, 3, 6, 11, 12, 13, 19]. Relations between CC and SC have been explored in [2, 17]. Perhaps, an easier way to understand the meaning of CC is by realizing that, under this consistency model, if a read operation \( r \) finds value \( v \) in object \( X \), and such value was written by a previous operation \( w \) (which according to our previous definition means that \( w \rightarrow r \)), then there cannot exist another write operation \( w' \) that writes a different value \( v' \) into the same object \( X \) and that \( w \rightarrow w' \rightarrow r \) (because if that were the case, then \( r \) should have seen the value \( v' \)).

Just for posterior comparisons, remember that Sequential Consistency (SC) is defined in [15] as:

**Definition 2.** Global history \( \mathcal{H} \) satisfies SC if there is a serialization \( S \) of the set \( \mathcal{H} \) that respects the program order of each site in the distributed system.

Notice that SC is a stronger consistency model than CC, since every distributed history that satisfies SC also satisfies CC, but the contrary is not always true.

### 4. Lifetime of Object Values

#### 4.1 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. Thus, an object might be replicated at several server sites. Client 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. A write operation assigns a particular value to an object copy stored in the local cache. Eventually, this version of the object is 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 \( \mathcal{C}_i \). In each 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 \( \mathcal{C}_i \). 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 this copy is stored in \( \mathcal{C}_i \), we denote it by \( X_s \).

#### 4.2 Start and Ending Times

Let \( \text{Clock}_i \) be the local vector clock of Site \( i \). This logical clock is updated every time that a local event occurs or when a new object is brought into the local cache (most likely, the value currently stored in this object was written at another site) following the standard rules for vector clocks [????].

The start time of the current value \( v \) of \( X_i \), denoted as \( X_i^0 \), is the value of the vector clock that the site that executed the corresponding write operation (i.e., the operation that assigned this value \( v \)) had at that instant. The start time of an object copy allows us to know since which logical instant the particular value of the object stored in the cache is valid. Our readers may notice why the importance of the previous simplification of assuming that each value written is unique.
Now, the latest logical time when the value stored in \( X_i \) is known to be valid is called its *ending time* and it is denoted as \( \omega_i \). For any object version \( X_i \) we have that \( \alpha_i \rightarrow \omega_i \) or \( \alpha_i = \omega_i \). The interval \([\omega_i, \alpha_i]\) is the *lifetime* of the value stored in \( X_i \). Notice that any later\(^1\) start time of a different value of the same object \( X \) will be causally after \( \omega_i \) or concurrent with \( \alpha_i \).

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, in principle, the local copies \( X_i \) and \( Y_i \) are mutually consistent if their lifetimes overlap, i.e., the lifetime of one object value does not end before the start of the lifetime of the other object value, or if the frontier events are concurrent.

### 4.3 Cache Context Time (\( \text{Context}_i \))

The protocol presented in this paper induces 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 locally invalidating those objects which do not satisfy the consistency criteria (which forces, in case that the locally invalidated object is needed later, that the object be provided by a server site).

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 *previously* in the cache \([13]\). For instance, if we don’t take into account the past contents of a cache with room for just a few objects or in the extreme case exactly one single object\(^2\), we may generate executions where an object value moves back and forth between two or more possible values. In order to avoid the *flickering* of the object values between present and past values and to guarantee the liveness of our protocols, we associate a logical timestamp called \( \text{Context}_i \) to \( \omega_i \). This timestamp is updated with the rules:

- The initial value of the vector clock \( \text{Context}_i \) is \( <0, 0, \ldots, 0> \) for every site in the distributed system.
- When a copy of object \( X \) is brought into \( \omega_i \) (becoming \( \omega_i \)), or when the local copy of object is updated (which copies \( \text{Clock}_i \) to \( \omega_i \)) then \( \text{Context}_i \) becomes \( \max(\text{Context}_i, \omega_i) \).

\( \text{Context}_i \) combines the information of all the *start times* of any object values that are stored or that *have been* stored in \( \omega_i \). The purpose of the timestamp \( \text{Context}_i \) will become evident when we explain the CC protocol proposed.

### 5. Causal Consistency and Protocol B

In this section we explore Protocol B, a CC protocol based on the concept of lifetime of object values. This protocol guarantees CC by ensuring that the contents of the caches of every site are always mutually consistent. One of the many interesting properties of logical vector clocks is that each site establishes its own private “line of time” which can advance independently from the rest of the system, unless that, because of a cache miss, a site gets forced to communicate with a server site (which indirectly may mean communications with another client site) connecting the private times of two or more different sites. Let’s point out two consequences of this:

1. For all objects \( X_i \in \omega_i \) (*i.e.*, the objects of Site \( i \)), their ending times \( \omega_i \) advance as the local vector clock at Site \( i \) progresses. Therefore, the lifetimes of all the values of local objects in \( \omega_i \) are actually being “stretched” or updated and becoming larger, at the very least in the local line of time.
2. Since the ending times of all the local objects are advancing at the same pace, they are never inconsistent among themselves and therefore are not locally invalidated, even when one of them is updated with a local write. However, as we will explain later, local invalidations in Protocol B are possible when a new object is brought into the local cache.

On the other hand, Protocol A, as explained in \([20]\), might trigger local invalidations as a consequence of a local write.

---

1. In relation to physical time.
2. Which is always mutually consistent with itself
operation in order to guarantee that the global history \( \mathcal{H} \) satisfies SC (see Definition 2), because in ordinary time we cannot assure that their current values are mutually consistent.

5.1 General Description of Protocol B

Before site \( i \) can access object \( X \), a local copy of this object must reside in its local cache \( \mathcal{C}_i \). If this is the case, the operation (either read or write) is executed locally without further immediate communications. We can 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 a server, site \( i \) makes \( X_i' = \max_i(\mathcal{V}_i, \text{Clock}_i) \). When this object is received by the server, the server updates its own logical clock with \( X_i' \) and then updates \( X_i'' \) with this new time.

Now, if site \( i \) has a cache miss when accessing object \( X \), Protocol B executes the following steps:

1. It is requested that a server site \( s \) sends back a copy of \( X_i \) such that \( \neg(\mathcal{V}_s' \rightarrow \text{Context}_i) \), i.e., the version provided by a server cannot have a start time that is causally before the local Context. This copy of \( X_s \) is inserted in \( \mathcal{C}_i \) and, from now on, it is denoted as \( X_i' \). A server \( s \) can always provide a copy of \( X \) that satisfies this requirement, either from its local information or by contacting other server or client sites.

2. Context becomes \( \max_i(\text{Context}_i, X_i') \), and Clock becomes \( \max_i(\text{Clock}_i, X_i') \). Thus, the local line of time is updated with the time information brought by the new object.

3. \( X_i'' = \max_i(\mathcal{V}_i, \text{Clock}_i) \) for all the local copies of objects \( Y_i \in C_i \). Lifetimes are always stretched under CC.

4. All the local copies of objects \( Y_i \in C_i \) such that \( \neg(\mathcal{V}_i' \parallel \mathcal{V}_i) \land (\min_i(\mathcal{V}_i', \mathcal{V}_i) \rightarrow \max_i(\mathcal{V}_i', \mathcal{V}_i)) \) are invalidated.

It is possible that site \( i \) is accessing a consistent state of the virtual world at some point in the past\(^3\). We know that all the objects currently stored in cache \( C_i \) are mutually consistent at logical time Context.

Given two object copies \( X_i \) and \( Y_i \in C_i \) such that \( \mathcal{V}_i' \parallel \mathcal{V}_i \), i.e., its start times are concurrent, it should be easy to see that their values are mutually consistent because there cannot exist an operation \( o \) such that \( o \) alters either \( X_i \) or \( Y_i \) and that \( X_i \rightarrow V(o) \rightarrow Y_i \) or that \( Y_i \rightarrow V(o) \rightarrow X_i \). Now, let \( X_i \) and \( Y_i \in C_i \) be such that \( \neg(\mathcal{V}_i' \parallel \mathcal{V}_i) \), i.e., their start times are causally related. Without loss of generality, let’s say that \( X_i \rightarrow Y_i' \). If the values stored in \( X_i \) and \( Y_i \) are mutually consistent it has to be the case that \( \neg(\mathcal{V}_i' \rightarrow \mathcal{V}_i') \), and vice versa.

In conclusion, two object copies \( X_i \) and \( Y_i \) are mutually consistent according to CC if \( (\mathcal{V}_i' \parallel \mathcal{V}_i) \lor \neg(\min_i(\mathcal{V}_i', \mathcal{V}_i) \rightarrow \max_i(\mathcal{V}_i', \mathcal{V}_i)) \). A cache \( C_i \) is consistent if every pair of elements in \( C_i \) 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.

Let’s formalize the previous discussion:

**Theorem 1.** Protocol B generates execution histories that satisfy CC.

**Proof.** We have to prove that if all the sites in a distributed system follow strictly Protocol B, our Definition 2 will be satisfied, i.e., for each site \( i \) there is a serialization \( S_i \) of the set \( \mathcal{H}_i+\mathcal{W} \) that respects the order “→”. Let’s make a proof by construction. We start by making \( S_i = \mathcal{H}_i \). Notice that the sequence \( \mathcal{H}_i \) respects the causal order “→”. Let \( w \in \mathcal{H}_i \) be a write operation not included in \( \mathcal{H}_i \) such that \( w \) assigns value \( v \) to an arbitrary object \( X \) and that this value is read by operation \( a \in \mathcal{H}_i \). Protocol B guarantees that \( V(a) \rightarrow V(w) \) and that there cannot exist another write operation \( w' \in \mathcal{H}_i \) that assigns value \( v' \) to object \( X \) and that \( V(w') \rightarrow V(w) \) → \( V(a) \), because if this were the case the value \( v \) of object \( X \) would have been invalidated by step 4 of Protocol B. Therefore, \( w \) can be inserted into its “proper” position in \( S_i \), i.e., in the rightmost

\(^3\) Again, “past” according to physical time.
position in $S_i$, such that there is not operation $a$ in $S_i$ to the left of $w$ such that $V(w) \rightarrow V(a)$. The rest of the write operations in $\gamma_i$ (whose values written are not read by any read operation in $\gamma_i$) are either concurrent with every operation inserted in $S_i$ or are causally related to some other write operation already in $S_i$. In both cases these write operations can be “easily” 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 [8, 9, 16, 18] that vector clocks capture completely the causality relation between all the operations in $\gamma_i$ and therefore they are able to detect all the cases where two operations are concurrent. However, [7] proved that given a distributed system with $N$ sites, causality can only be captured by vector clocks with $N$ entries. If $N$ is large, severe problems arise such as growing storage costs, considerable communications overhead and extended processing time. Thus, vector clocks have poor scalability. Plausible clocks [21] 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 $\epsilon$. Examples of plausible clocks are presented in [5, 21].

In this section, we briefly 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 $P$. 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, given the imperfections of plausible clocks, 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 $C$ still induces executions that satisfy Causal Consistency. Thus:

**Theorem 2.** Protocol $C$ generates execution histories that satisfy $CC$.

**Proof.** The properties of plausible clocks [21] allows us to reutilize shamelessly the proof of Theorem 1.

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.

Just for purposes of illustration, Table 1 shows some results of a very 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 Protocols $A$ [20], $B$ and $C$. 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 $REV$ [21] with 5 entries per clock, is not much bigger than the number of messages generated by Protocol $C$. 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>
<thead>
<tr>
<th></th>
<th>Updates</th>
<th>Cache Misses</th>
<th>Invalidations</th>
<th>Validations</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalidation</td>
<td>319</td>
<td>1310</td>
<td>1260</td>
<td>0</td>
<td>2890</td>
</tr>
<tr>
<td>Protocol $A$</td>
<td>319</td>
<td>1114</td>
<td>771</td>
<td>118</td>
<td>2322</td>
</tr>
<tr>
<td>Protocol $B$</td>
<td>319</td>
<td>382</td>
<td>51</td>
<td>0</td>
<td>752</td>
</tr>
<tr>
<td>Protocol $C$</td>
<td>319</td>
<td>488</td>
<td>153</td>
<td>0</td>
<td>960</td>
</tr>
</tbody>
</table>

Table 1: Number of Messages with Different Consistency Protocols
7. Conclusions
The lifetime of an object value goes from its start time, which corresponds to the logical instant when this particular value is written, until its ending time, which is the latest logical time when this value is known to be valid. Protocol B follows almost the same structure as Protocol A [20] but using vector clocks instead of physical clocks which relaxes Sequential Consistency to a weaker (but less expensive in communications) Causal Consistency on the executions. Similarly, Protocol C is almost identical to Protocol B, but replaces vector clocks with constant-size plausible clocks. 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.

The scalability of both Protocols B and C derives from the fact that many consistency actions are decided locally by each site. Once that a site has collected (or hoarded) a mutually consistent set of objects, its local computations can access these objects with read and write operations, in an undisturbed fashion, for as long as they need, and the protocol still guarantees Causal Consistency. Of course, as it was mentioned, the site can be accessing a consistent but old view of the universe which can be updated by appropriate communications with sever sites (operation which could cause the local invalidation of some objects). Notice the interesting property of Causal Consistency of allowing the disconnected activities of sites in a distributed system (assuming of course that the application works properly under Causal Consistency).

References