Timed Consistency for Shared Distributed Objects†

Francisco J. Torres-Rojas
College of Computing
Georgia Institute of Technology
U.S.A.
torres@cc.gatech.edu

Mustaque Ahamad
College of Computing
Georgia Institute of Technology
U.S.A.
mustaq@cc.gatech.edu

Michel Raynal
IRISA
University of Rennes
France
raynal@irisa.fr

ABSTRACT

Ordering and time are two different aspects of consistency of shared objects in a distributed system. One avoids conflicts between operations, the other addresses how quickly the effects of an operation are perceived by the rest of the system. Consistency models such as sequential consistency and causal consistency do not consider the particular time at which an operation is executed to establish a valid order among all the operations of a computation. Timed consistency models require that if a write operation is executed at time t, it must be visible to all nodes by time $t+\Delta$. Timed consistency generalizes several existing consistency criteria and it is well suited for interactive and collaborative applications, where the action of one user must be seen by others in a timely fashion.

1 INTRODUCTION

Non-local access to objects in a distributed system is expensive due to high communication costs. In order to alleviate this inefficiency and to improve the reliability, availability and scalability of the system, techniques such as caching and replication are normally used. However, these methods lead to the coexistence of many, possibly different, versions of the same object. This problem is addressed by defining convenient consistency models for the state of the distributed objects. A consistency model establishes a series of guarantees about the valid relationships among the operations executed by the sites of the system, which in turn constrain the possible values that shared objects can return when **read** operations are executed.

In many consistency models, real-time is not explicitly captured. For example, sequential consistency only requires that it is possible to serialize all the operations executed in the distributed system [25]. Causal consistency, a weaker model, only requires that causally related operations be seen in the same order by all the sites of the system, while concurrent operations may be perceived in different orders by different sites [2]. On the other hand, strict serializability [14, 30] and linearizability [20] require that the serializations respect real-time ordering between transactions and operations, respectively. The notion of time has also been explored in communications systems such as delta causal broadcast [7, 8], in memory systems that have a temporal consistency model (e.g., delta consistency [34]) and in several World-Wide Web (WWW) cache consistency protocols [10, 11, 19].

We propose a *timed consistency* model which defines a maximum acceptable threshold of time after which the effects of a **write** operation must be observed by all the sites of a distributed system. Timed consistency not only meets the needs of many applications that must observe updates to dynamically changing objects in a timely fashion, but also unifies existing consistency models such as sequential consistency and linearizability. We focus our attention on *timed serial consistency* (**TSC**) and *timed causal consistency* (**TCC**) models. We will show that sequential consistency and linearizability are special cases of **TSC**.

In Section **2**, we define the consistency criteria contemplated in this paper. We present the concept of timed consistency and the particular cases of **TSC** and **TCC** in Section **3**. Some applications are presented in Section **4**. In Section **5**, we describe a possible implementation of these consistency levels. The paper is concluded in Section **6**.

2 CONSISTENCY CRITERIA

The *global history* H of a distributed system is a partially ordered set of all operations occurring at all sites of the system. H_i is the sequence of operations that are executed on site i. If **a** occurs before **b** in H_i we say that **a** precedes **b** in program order. We assume that all the operations in H are either **read** or **write** (in order to simplify, it is assumed that each value written is unique). These operations take a finite, non-zero time to execute, hence there is an interval that goes from the time when a **read** or **write** "starts" to the time when

† This work was supported in part by NSF grants CCR-9619371 and INT-9724774.

PODC '99 Atlanta GA USA

such an operation "finishes". However, for the purposes of timed consistency, we associate an instant to each operation called the *effective time* of the operation. The effective time of an operation corresponds to some instant between its start and its end. For short, if the effective time of **a** is t, we say that **a** was "executed" at time t.

If D is a set of operations, then a *serialization* of D is a linear sequence S containing exactly all the operations of 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. Let ~ be an arbitrary partial order relation defined over D, we say that serialization S "respects ~" if $\forall \mathbf{a},\mathbf{b} \in D$ such that $\mathbf{a} \sim \mathbf{b}$, it happens that **a** precedes **b** in S.

History H satisfies *linearizability* (**LIN**) if there is a serialization of H that respects the order induced by the effective times of the operations [20]. *Strict serializability* is defined over histories formed by *transactions*, and it requires the existence of a serialization of H that respects the real-time order of the transactions [14, 30]. **LIN** can be seen as a particular case of strict serializability where each transaction is a predefined operation on a single object. *Normality*, as proposed in [17], is equivalent to **LIN** when operations on objects are unary, but it is strictly weaker than **LIN** when operations can span several objects.

A weaker level of consistency is offered by *sequential consistency* (**SC**). This model, defined by Lamport in [25], does not guarantee that a **read** operation returns the most recent value with respect to real-time, but just 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. History H satisfies **SC** if there is a serialization of H that respects the program order for each site in the system. The power and implementation costs of **SC** and **LIN** are compared by Attiya and Welch in [6].

The *causality relation* "$\rightarrow$" for message passing systems as defined in [26] can be modified to order the operations of H. Let $\mathbf{a},\mathbf{b},\mathbf{c} \in H$, we say that $\mathbf{a} \rightarrow \mathbf{b}$, i.e., **a** causally precedes **b**, if one of the following holds:

(i) **a** and **b** are executed on the same site and **a** is executed before **b**;
(ii) **b** reads an object value written by **a**;
(iii) $\mathbf{a} \rightarrow \mathbf{c}$ and $\mathbf{c} \rightarrow \mathbf{b}$.

Two distinct operations **a** and **b** are *concurrent* if none of these conditions hold between them. Let H_{i+w} be the set of all the operations in H_i plus all the **write** operations in H. History H satisfies *Causal Consistency* (**CC**) if for each site i there is a serialization of H_{i+w} that respects causal order "$\rightarrow$" [2]. Thus, if $\mathbf{a},\mathbf{b},\mathbf{c} \in H$ 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 $\mathbf{a} \rightarrow \mathbf{b} \rightarrow \mathbf{c}$.

CC requires that all the causally related operations be seen in the same order by all sites, while different sites may perceive concurrent operations in different orders. **CC** is a model of consistency weaker than **SC**, but it can be efficiently implemented [2]. **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 [9] and in shared memory and object systems [3, 4, 5, 21, 23, 24, 39]. Relations between **SC** and **CC** have been studied by Ahamad et al. [2] and by Raynal and Schiper [32].

3 TIMED CONSISTENCY

In neither **SC** nor **CC**, real-time is explicitly captured. In **SC**, operations may appear out of order in relation to their effective times. In **CC**, each site can see concurrent **write** operations in different orders. On the other hand, **LIN** requires that the operations be observed in an order that respects their real-time ordering [20]. Ordering and time are two different aspects of consistency. One avoids conflicts between operations, the other addresses how quickly the effects of an operation are perceived by the rest of the system.

Timed consistency requires that if the effective time of a **write** is t, the value written by this operation must be visible to all sites in the distributed system by time $t + \Delta$, where Δ is a parameter of the execution. We can see that when Δ is 0, timed consistency becomes **LIN**, i.e., timed consistency is a generalization (or a weakening) of **LIN**.

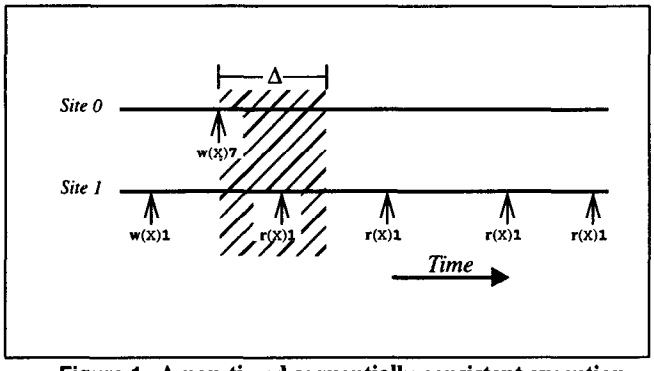


Figure 1. A non-timed sequentially consistent execution.

The execution showed in Figure 1 satisfies **SC** and **CC** but not **LIN**. Up to the second operation of site *1*, the execution satisfies timed consistency for the value of Δ represented in this figure, but, by that same instant, **LIN** is no longer satisfied. After this point, the execution is not even timed because there are **read** operations in site *1* that start more than Δ units of real-time after site *0* writes the value **7** to object X and these **read** operations do not return this value.

3.1 Reading on Time

In *timed* models, the set of values that a **read** may return is restricted by the amount of time that has elapsed since the preceding **write**s. A **read** occurs *on time* if it does not return stale values when there are more recent values that have been available for more than Δ units of time. This definition depends on the properties of the underlying clock used to assign timestamps to the operations in the execution. First, we assume perfectly synchronized physical clocks, where $T(\mathbf{a})$ is the real-time instant corresponding to the effective time of operation **a**.

Definition 1. Let $D \subseteq H$ be a set of operations and S a serialization of D. Let $\mathbf{w},\mathbf{r} \in D$ be such that **w** writes a value into object X that is

later read by **r**, i.e., **w** is the closest **write** operation into object X that appears to the left of **r** in serialization S. We define the set W_r, associated with **r**, as:

$$W_r = \{\mathbf{w}' \in D : (\mathbf{w}' \text{ writes a value into object X}) \wedge (T(\mathbf{w}) < T(\mathbf{w}') < (T(\mathbf{r}) - \Delta))\}$$

We say that operation **r** *occurs or reads on time* in serialization S, if $W_r = \emptyset$. S is *timed* if every **read** operation in S occurs on time. ❑

Figure 2 illustrates Definition **1** presenting a possible arrangement of **read** and **write** operations over the same object. Operation **r** reads a value previously written by operation **w**. Since operation $\mathbf{w}_1$ was executed before **w**, it has no effect on whether **r** is reading on time or not. Similarly, although $\mathbf{w}_4$ is more recent than **w**, the interval Δ has not elapsed yet when **r** is executed, and, thus, it is acceptable that **r** does not observe the value written by $\mathbf{w}_4$. On the other hand, operations $\mathbf{w}_2$ and $\mathbf{w}_3$ occur after **w**, and the values written by them have been available in the system for more than Δ units of time when **r** is executed. Thus, $\mathbf{w}_2$ and $\mathbf{w}_3$ are in W_r and, therefore, operation **r** does not occur on time. The shaded area between $T(\mathbf{w})$ and $T(\mathbf{r}) - \Delta$ represents the interval of time associated with the set W_r, which, according to Definition **1**, must be empty if **r** reads on time (i.e., no **write** operation to the same object read by **r** can occur in this interval).

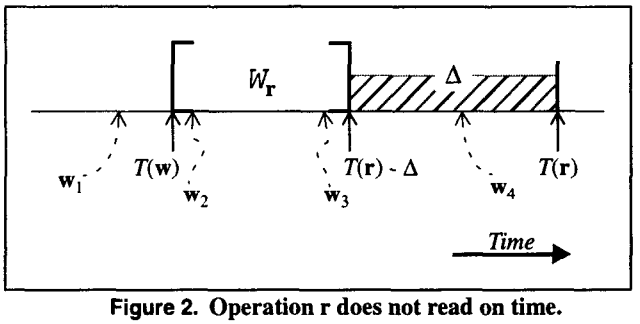


Figure 2. Operation r does not read on time.

3.2 Approximately-synchronized real-time clocks

Physical clocks do not keep perfect time and there will always be differences among the clocks at different nodes. In approximately-synchronized real-time clocks, periodic resynchronizations guarantee that no two clocks differ by more than ε units of time. Typically, it is also guaranteed that the difference between any clock and the "real" time (as maintained by a time server) is never more than $\varepsilon/2$ [12, 13, 22, 28, 29]. Therefore, if the effective time of operation **a** was reported by a particular site as $T(\mathbf{a})$, then, from the point of view of the time server, this effective time corresponded to some instant in the interval $[T(\mathbf{a}) - \varepsilon/2, T(\mathbf{a}) + \varepsilon/2]$.

Similarly to [35], we say that, $\forall \mathbf{a}, \mathbf{b} \in H$, **a** *definitely occurred* before **b** if $T(\mathbf{a}) + \varepsilon/2 < T(\mathbf{b}) - \varepsilon/2$, or, equivalently, if $T(\mathbf{a}) + \varepsilon < T(\mathbf{b})$. If neither **a** nor **b** definitely occurred before the other one, we say that their timestamps are non-comparable or concurrent (i.e., the imprecision of the clocks does not allow us to decide which operation occurred earlier).[1]

Let $\mathbf{w}, \mathbf{r} \in D \subseteq H$ be such that **w** writes a value into object X that is later read by **r**. Let $\mathbf{w}' \in D$ also update object X. If $T(\mathbf{w}')$ definitely occurred before $T(\mathbf{w})$, or if $(T(\mathbf{r}) - \Delta)$ definitely occurred before $T(\mathbf{w}')$, then it is clear that **w'** does not affect the fact that **r** occurs on time. Now, if either $T(\mathbf{w}')$ and $T(\mathbf{w})$ are concurrent, or if $(T(\mathbf{r}) - \Delta)$ and $T(\mathbf{w}')$ are concurrent, there is no evidence of **w'** being more recent than **w**, or of **w'** occurring more than Δ units of real-time before **r**, respectively. In these cases, we can still claim that **r** occurs on time. Conversely, **r** is not reading on time if $T(\mathbf{w})$ definitely occurred before $T(\mathbf{w}')$ and if $T(\mathbf{w}')$ definitely occurred before the instant $(T(\mathbf{r}) - \Delta)$. Therefore:

Definition 2. Let $D \subseteq H$ be a set of operations and S a serialization of D. Let $\mathbf{w}, \mathbf{r} \in D$ be as presented in Definition **1**. We define the set W_r, associated with **r**, as:

$$W_r = \{\mathbf{w}' \in D : (\mathbf{w}' \text{ writes a value into object X}) \wedge (T(\mathbf{w}) + \varepsilon < T(\mathbf{w}')) \wedge (T(\mathbf{w}') + \varepsilon < T(\mathbf{r}) - \Delta)\}$$

We say that operation **r** *occurs or reads on time* in serialization S, if $W_r = \emptyset$. S is *timed* if every **read** operation in S occurs on time. ❑

Figure 3 shows the same example presented in Figure 2, but assuming approximately-synchronized clocks. As before, operations $\mathbf{w}_1$ and $\mathbf{w}_4$ do not affect whether or not **r** reads on time. Given the imprecision of the clocks used to timestamp the events, operations **w** and $\mathbf{w}_2$ are considered concurrent. Besides, it cannot be decided if operation $\mathbf{w}_3$ occurred more than Δ units of time before **r**. Hence, the set W_r is empty, and **r** occurs on time. Notice how the interval associated with the set W_r is 2ε units of time shorter than the one presented in Figure 2. When $\varepsilon = 0$, Definition **2** reduces to Definition **1**.

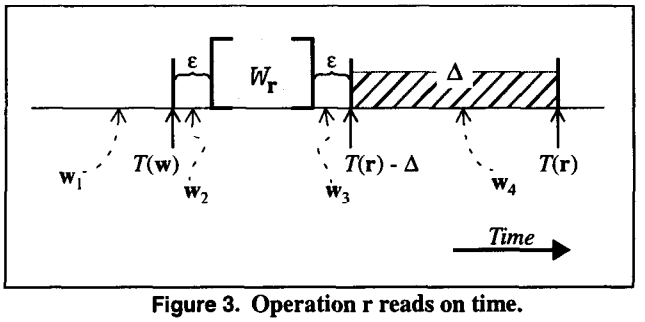


Figure 3. Operation r reads on time.

3.3 TSC and TCC

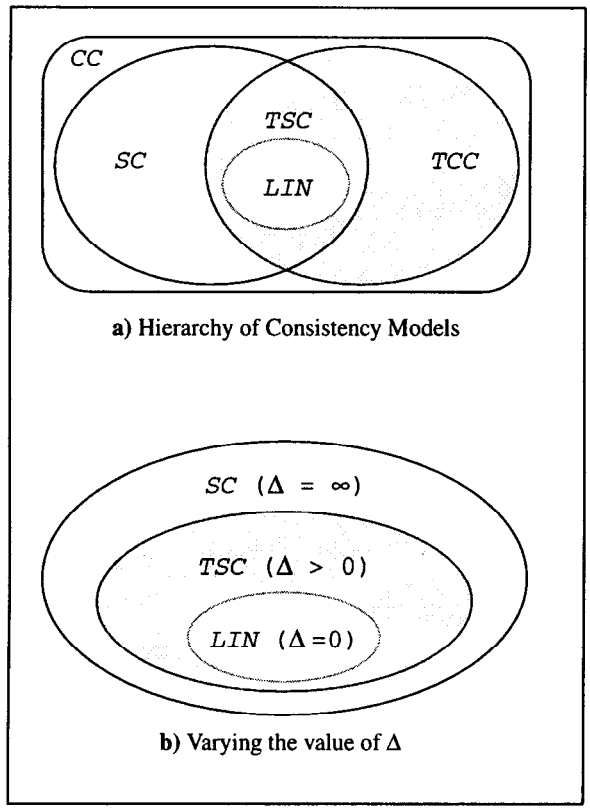
We now combine the requirements of well-known consistency models such as **SC** and **CC** with the requirements of reading on time:

Definition 3. History H satisfies *timed serial consistency* (**TSC**) if there is a **timed** serialization S of H that respects the program order for each site in the system. ❑

1. In this context "definitely occurs before" only refers to the order of the operations in real-time and not necessarily implies any causal relationship.

Definition 4. History H satisfies *timed causal consistency* (**TCC**) if for each site i there is a **timed** serialization S_i of H_{i+w} that respects causal order "$\rightarrow$". ❑

Let LIN, SC, CC, TSC and TCC be the sets of all the executions that, respectively, satisfy **LIN**, **SC**, **CC**, **TSC** and **TCC**. Figure 4.**a** presents the hierarchy of these sets. If an execution satisfies **LIN** it satisfies **SC** as well, but the contrary is not always true. If a set of operations D satisfies **LIN**, then it is always possible to produce a serialization S of D such that all the operations are ordered by their respective effective times. In turn, S satisfies Definition **1**, even for $\Delta = 0$. Then, **LIN** is a case of **TSC** where $\Delta = 0$, and therefore $LIN \subset TSC$. It is easy to see that $SC \subset CC$.



a) Hierarchy of Consistency Models

b) Varying the value of Δ

Figure 4.

Let T be the set of all the timed executions. From Definition **3**, we have that:

$$TSC = T \cap SC \Rightarrow TSC \subset SC \subset CC$$

From Definition **4**, we have that:

$$TCC = T \cap CC \Rightarrow TCC \subset CC$$

Then:

$$TCC \cap SC = T \cap CC \cap SC = T \cap SC = TSC$$

Figure 4.**b** shows, for the case of **TSC**, the effects of different values of Δ (i.e., both **SC** and **LIN** can be seen as particular cases of **TSC**).

3.4 Examples

Figure 5.**a** presents a sequentially consistent execution. Operation $\mathbf{w}_i(\mathrm{X})\mathbf{v}$ is executed at site i and writes value **v** into object X; operation $\mathbf{r}_i(\mathrm{X})\mathbf{v}$ is executed at site i and reads value **v** from object X. Each operation has an associated real-time instant that corresponds to its effective time. Let **0** be the initial value for any object. Figure 5.**b** presents a possible serialization of all the operations that respects the program order of each site. This serialization proves that the execution satisfies **SC**, but it does not capture the real-time ordering of the effective times of the operations since, in fact, several are observed in reversed order (e.g., $\mathbf{w}_0(\mathrm{C})\mathbf{6}$ and $\mathbf{w}_2(\mathrm{B})\mathbf{5}$, or $\mathbf{r}_4(\mathrm{C})\mathbf{6}$ and $\mathbf{w}_2(\mathrm{C})\mathbf{7}$).

The definition of **TSC** depends on the particular value of Δ. For instance, site *0* updates the value of object C to **6** at instant 338 and site *2* updates the value of the same object C to **7** at instant 340. At instant 436, site *4* reads object C and the value returned is **6**. If, for instance, $\Delta = 50$ this execution does not satisfy **TSC** because by instant 436, site *4* must be aware of the **write** operation $\mathbf{w}_2(\mathrm{C})\mathbf{7}$. On the other hand, for $\Delta > 96$ this execution satisfies **TSC**. Similarly, site *2* updates the value of object B to **5** at instant 274, but, at instant 301, site *3* reads value **2** from object B; if $\Delta < 27$ then this execution does not satisfy **TSC**.

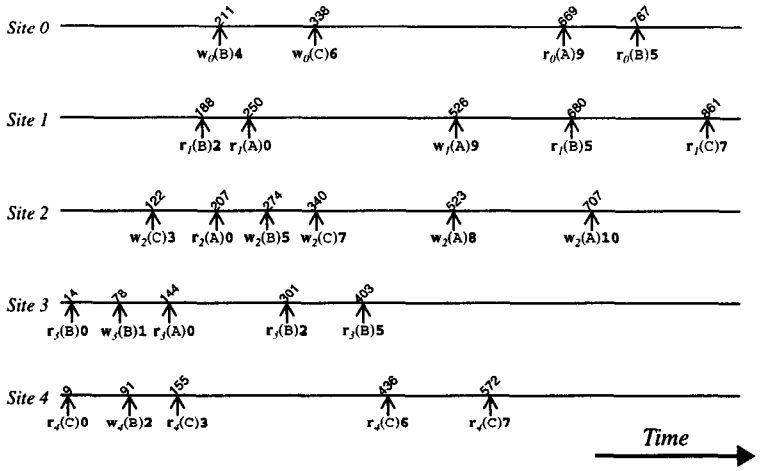
Figure 6.**a** presents an execution that satisfies **CC** but not **SC** (among other things, operation $\mathbf{r}_0(\mathrm{B})\mathbf{4}$ disallows a serialization of all the operations that respects the program order of each site[2]). Figure 6.**b** shows serializations for each local history, marked by the shaded boxes, plus all the external **writes**, that respect the causality of the operations. These serializations do not necessarily follow the real-time ordering of the operations and, in several of the serializations, concurrent **write** operations are perceived in different orders. The chosen value of Δ establishes whether or not the execution satisfies **TCC**. For instance, if $\Delta = 30$ then operation $\mathbf{r}_4(\mathrm{C})\mathbf{0}$ executed at instant 155 violates **TCC** because it ignores operation $\mathbf{w}_2(\mathrm{C})\mathbf{3}$ that was executed at instant 122.

4 APPLICATIONS OF TIMED CONSISTENCY

In spite of the usefulness and importance of consistency models such as **SC** and **CC**, their formal definition and practical implementation permit executions that ignore the real-time occurrence of certain events. In many cases this could be considered an advantage that allows efficient and convenient implementations. For instance, **CC** is well suited to mobility applications and has the ability to handle disconnections smoothly [3, 4]. Mechanisms to gracefully weaken the consistency requirements (e.g. from **SC** to **CC**) are described in [23], giving control to the application to voluntarily disconnect.

Many applications require more support to detect and exploit the correct real-time relationships between events. The trivial execution presented in Figure 1 satisfies **SC**, since it is possible to serialize all the operations corresponding to site *2* before the **write** operation of

2. In general, determining whether or not an execution satisfies SC is a NP-complete problem [18, 36].

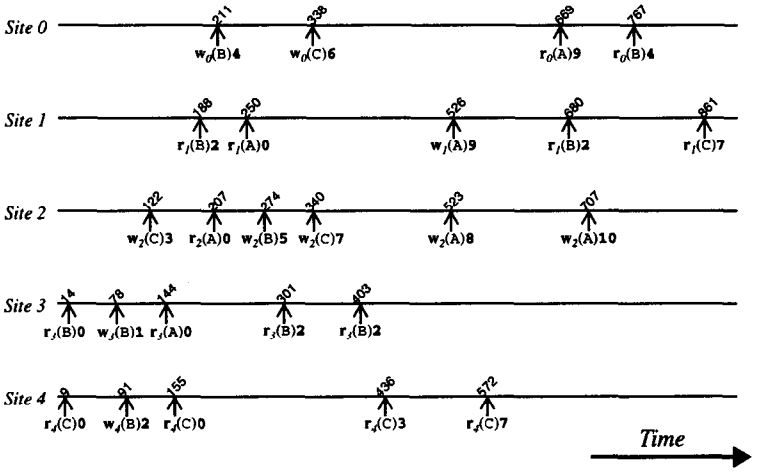


a) Sequentially Consistent Execution

$r_4(C)0$ $r_3(B)0$ $w_0(B)4$ $w_2(C)3$ $r_2(A)0$ $w_3(B)1$ $r_3(A)0$ $w_4(B)2$ $r_4(C)3$ $r_3(B)2$ $r_1(B)2$ $r_1(A)0$ $w_0(C)6$
$w_1(A)9$ $r_0(A)9$ $w_2(B)5$ $r_1(B)5$ $r_0(B)5$ $r_3(B)5$ $r_4(C)6$ $w_2(C)7$ $r_1(C)7$ $r_4(C)7$ $w_2(A)8$ $w_2(A)10$

b) Serialization that respects program order

Figure 5.



a) Causally Consistent Execution

S_0 : $w_4(B)2$ $w_0(B)4$ $w_0(C)6$ $w_1(A)9$ $r_0(A)9$ $r_0(B)4$ $w_2(C)3$ $w_2(B)5$ $w_2(C)7$ $w_2(A)8$ $w_2(A)1$ $w_3(B)1$

S_1 : $w_2(C)3$ $w_2(B)5$ $w_4(B)2$ $r_1(B)2$ $r_1(A)0$ $w_1(A)9$ $r_1(B)2$ $w_2(C)7$ $r_1(C)7$ $w_0(B)4$ $w_0(C)6$ $w_2(A)8$ $w_2(A)10$ $w_3(B)1$

S_2 : $w_2(C)3$ $r_2(A)0$ $w_2(B)5$ $w_2(C)7$ $w_2(A)8$ $w_2(A)10$ $w_4(B)2$ $w_0(B)4$ $w_0(C)6$ $w_1(A)9$ $w_3(B)1$

S_3 : $r_3(B)0$ $w_3(B)1$ $r_3(A)0$ $w_4(B)2$ $r_3(B)2$ $r_3(B)2$ $w_0(B)4$ $w_0(C)6$ $w_1(A)9$ $w_2(C)3$ $w_2(B)5$ $w_2(C)7$ $w_2(A)8$ $w_2(A)10$

S_4 : $r_4(C)0$ $w_4(B)2$ $r_4(C)0$ $w_2(C)3$ $w_2(B)5$ $r_4(C)3$ $w_2(C)7$ $r_4(C)7$ $w_0(B)4$ $w_0(C)6$ $w_1(A)9$ $w_2(A)8$ $w_2(A)10$ $w_3(B)1$

b) Serializations that respect causal order

Figure 6.

site *1* or vice versa. However, the last **read** operation of site *2* may occur several hours after the **write** operation of site *1* and the value **7** is yet to be seen. Evidently, this violates **TSC** and **TCC** (the shaded area represents the span of Δ). If this execution corresponded to an application of, for instance, multi-user virtual environments, this behavior may not be acceptable because the most recent **write** could imply a serious alteration of the environment that is not perceived on time by site *2*.

Timed consistency models are useful in applications where the observance of the passing of real time and their effects is part of the definition of correctness. In the same way as it occurs in nature, we can allow the passing of a finite period of time before the effects of an operation are known to all the sites in the system. This period is precisely the parameter Δ. For instance, it is valid that the first **read** executed by site *2* in Figure 1 returns the value **1**, because the new value of X has not yet been made visible to the entire system.

The simulation of interactive virtual environments is explored by Singla et al. in [34], where the concept of "delta consistency" is defined as a correctness criterion to control accesses to a shared memory. This timed consistency model is particularly appropriate for expressing the requirements of interactive application domains.

The notion of Δ-causality in unreliable networks is presented by Baldoni et al. in [7, 8], this protocol supports multimedia real-time collaborative applications. In order to deliver a message to a process, it is verified that the causal order of messages is met (similar to [9]) and that the lifetime Δ of the message has not yet expired. Their approach is slightly different than the one expressed in Definition **3** because late messages are never delivered, and it is assumed that a more updated message will eventually be received.

The features of *Semantic Distributed Shared Objects* (**S-DSO**) are presented by West et al. in [41]. In order to improve performance, this system exploits application-level temporal and spatial constraints on shared objects, allowing applications to specify when and which processors should see the effects of **write** operations to shared objects. A multimedia application based on **S-DSO** is also described in [41].

Web cache consistency protocols can be modeled as timed consistency protocols, and if the causal relationships between documents are considered then **TSC** or **TCC** are appropriate models. The use of an adaptive time-to-live (TTL) protocol based on the Alex file system [11] for World-Wide Web documents is favored by the results of Gwertzman and Seltzer [19]. They found that this cache consistency protocol reduces network bandwidth consumption and server loads in comparison to other protocols. On the other hand, the results of Cao and Liu [10] advocate a consistency protocol based on invalidations of outdated cached web documents that are initiated by the server sites. Both [19] and [10] distinguish between "weak" and "strong" consistency of web documents, which can be modeled with different values of Δ.

The level of consistency provided by **TCC** is useful even in applications in which clients cache objects for read-only access. Assume that an user cached a web page containing the Dow Jones index and a CNN web page, with no causal relations between them. This state of the cache satisfies **CC**. At a later point in time, the user downloads a more recent version of the CNN web page describing a sudden fall in the Dow Jones index, with a link to the page containing the Dow Jones index. Clearly, accessing the outdated web page with the old Dow Jones index after reading the CNN article violates **CC**, therefore the old Dow Jones page must be invalidated. On the other hand, if the user does not download new versions of any of the original web pages for weeks, the cache still satisfies **CC**, but, for a value of Δ set to a few hours, does not satisfy **TCC**.

5 IMPLEMENTATION

SC and **CC** can be implemented in a variety of ways, with varying degrees of flexibility and performance [1, 2, 3, 4, 6, 9, 20, 23, 25]. We explore implementations of **TSC** and **TCC** based on the *lifetime* of distributed objects as presented in [39].

5.1 Lifetime Based Consistency Protocol

This technique provides consistency across different but related set of objects. We assume an architecture where each object has a set of server sites that provide long term storage for the object and where client sites cache objects before accessing them. Cache misses are handled by a server site, which either has a copy of the requested object or can obtain it.

Let the set C_i denote the cache of site i, in which it stores copies of objects that have been accessed recently. If a cache miss occurs when accessing object X, some server provides a copy of its current version of X. Once this copy is stored in C_i, we denote it as X_i. The *start time* of X_i, denoted as X_i^{α}, is the time when the value of X_i was written. The latest time when the value stored in X_i is known to be valid is its *ending time* and it is denoted as X_i^{ω}. The interval $[X_i^{\alpha}, X_i^{\omega}]$ is the currently known *lifetime* of the value stored in X_i.

The values of X_i and Y_i (cached in C_i) are mutually consistent if $max(X_i^{\alpha}, Y_i^{\alpha}) \leq min(X_i^{\omega}, Y_i^{\omega})$, i.e., their lifetimes overlap and, thus, they coexisted at some instant. C_i is consistent if the *maximum* start time of any object value in C_i is less than or equal to the *minimum* ending time of any object value in C_i, i.e., every pair of objects in C_i is mutually consistent.

In general, the lifetime of arbitrary object values is not known. When site i updates object version X_i at time t, timestamp t is assigned to both X_i^{α} and X_i^{ω}. 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}. Details of protocols that determine if object copy X_i is valid at time $t' > X_i^{\omega}$ can be found in [24, 39].

A local timestamp variable called $\mathbf{Context}_i$ is associated with C_i. Its initial value is 0, and it is updated with the rules:

1. When a copy of object X is brought into C_i (becoming X_i):
   $\mathbf{Context}_i := max(X_i^{\alpha}, \mathbf{Context}_i)$
2. When object copy X_i is updated at time t:
   $\mathbf{Context}_i := X_i^{\alpha} := t$

Context$_i$ keeps the latest start time of any object value that is or has been stored in C_i. When a copy of object X is brought into C_i, its ending time must not be less than **Context**$_i$, if necessary, other servers or client sites are contacted until a version of X that satisfies the condition is found. Furthermore, any object $Y_i \in C_i$ such that $Y_i^{\omega} <$ **Context**$_i$ is invalidated. It is proved in [39] that this protocol induces **SC** on the execution.

5.2 TSC Implementation

Notice that, under the lifetime protocol described in the previous section, site i accesses a state of the objects that *was* consistent at time **Context**$_i$, but the current time can be much later. By controlling that the difference between the current time and **Context**$_i$ is less than or equal to Δ, we are actually inducing **TSC**. Object values whose ending times are older than Δ units of real-time in the past are locally invalidated. This is easily accomplished by adding an extra rule for the updating of **Context**$_i$ (let t_i be the current time at site i):

3. **Context**$_i$:= $max(t_i - \Delta,$ **Context**$_i)$

This works nicely with objects whose ending times are well-known (e.g., write-once objects, leased objects, periodic objects, etc.), but as [39] indicates, it may generate unnecessary invalidations for arbitrary objects whose lifetimes are not known accurately. Several optimizations are possible, for instance, when **Context**$_i$ is updated, those objects whose ending times are less than **Context**$_i$ are not invalidated but just marked as *old*. Any future access to an old object initiates a validation operation through one of the servers, which either advances the ending time of the object value or provides a newer version of the object. Much of this can be done by just comparing timestamps, which avoids the unnecessary sending of large objects. This is similar to the TTL approach used by the current HTTP protocol [16]: if a document whose TTL has expired is accessed, the client site polls the server with an *if-modified-since* request including the URL of the document and its current timestamp. The server checks whether the document has been modified since the timestamp; if this is the case a newer version of the document is returned, otherwise the server just informs the client that its cached copy is still valid [10, 19]. Alternatively, an asynchronous component of the system may update old versions of certain cached objects before they are accessed, i.e., a push update-propagation policy.

In general, the faster a client site communicates a recent update of an object value to a server site, the more efficient the system becomes; however, this does not affect the correctness of the protocol.

5.3 TCC Implementation

The lifetime protocol described in section **5.1** is modified in [39] to induce **CC** on the executions. All the timestamps used in the system (i.e., local clock, **Context**$_i$, and the start and ending times of object values) are now taken from vector clocks [15, 27] or from plausible clocks [37]. The rules of the protocols are slightly modified to consider concurrent timestamps. This variant of the lifetime protocol does not require physical clocks and generates considerably less communications overhead than its **SC** counterpart.

Context$_i$ is updated with versions of rules **1** and **2** of Section **5.1** adapted to logical clocks. Among other things, this requires computing the *maximum* and *minimum* of two logical timestamps that are taken from either vector clocks or plausible clocks [38]. When an object copy is brought from a server into C_i, we verify that its ending time is not *causally before* **Context**$_i$ (however, it may be *concurrent* with **Context**$_i$). If necessary, other servers or client sites are contacted until a version of the object that satisfies this condition is found. Any object $Y_i \in C_i$ whose ending time is causally before **Context**$_i$ (i.e. $Y_i^{\omega} \rightarrow$ **Context**$_i$) is invalidated. Local copies of objects could be invalidated when a new object is brought into the cache, but they are never invalidated as a consequence of the update of a local object value because the logical ending times of local object copies are advanced together with the local logical time of the site [39].

Since **TCC** requires the use of physical clocks in order to guarantee that the effects of a **write** operation are observed before Δ units of real-time, we add a new timestamp taken from a physical clock: the *checking time* of X_i, denoted as X_i^{β}. This timestamp is the latest real-time instant when the value stored in X_i is known to be valid.

Let t_i be the current real-time of site i. When a copy of object X is brought from server s into C_i, we now also require that $X_s^{\beta} \geq t_i - \Delta$. Similarly, any local object $Y_i \in C_i$ such that $Y_i^{\beta} < t_i - \Delta$ is invalidated or marked as old. These rules guarantee that the site always reads on time, because object values suspected to be old are either validated or replaced by newer versions.

Under the same circumstances, this implementation of **TCC** tends to invalidate more objects than the implementation of **CC** presented in [39], but less than the implementation of **TSC** described in Section **5.2**. The optimizations mentioned for **TSC** are also applicable to **TCC**.

5.4 TCC using logical clocks

Implementations that induce **CC** with just the use of logical clocks are described in [3, 23, 24, 33, 39]. **TCC** can be approximated in a distributed system whose sites only share a logical clock. We define a map ξ from the set of logical timestamps to the set of real numbers. The parameter Δ will no longer be expressed in real time units, but as a real number that defines the maximum difference between the values that ξ produces for certain logical timestamps (e.g., the timestamp associated with the event that updates the value **v** of a particular object and the timestamp associated with any operation that reading this same object retrieves value $\mathbf{v}' \neq \mathbf{v}$).

Definition 5. Function ξ maps timestamps taken from a logical clock to the set of real numbers. Let t and u be two logical timestamps, ξ has the following properties:

$$t = u \Rightarrow \xi(t) = \xi(u)$$
$$t \rightarrow u \Rightarrow \xi(t) < \xi(u)$$ ❑

Informally, $\xi(t)$ represents the "amount" of global activity of the system that is known when the event associated with timestamp t is generated. Even if timestamps t and u are concurrent, we want that if at time u the system is aware of more global activity than at time

t, then $\xi(t) > \xi(u)$. For instance, the concurrency measures proposed in [31] can be adapted for the definition of convenient and coherent maps ξ from logical timestamps to real numbers. In any case, the definition of ξ depends on the particular representation of logical time used and there is not a unique or best way to compute it.

Definition **4** is valid with logical clocks if we just redefine the concept of *reading on time* given in Definition **1**. If $\mathbf{a} \in H$, we say that $L(\mathbf{a})$ is the logical time at which **a** is executed.

Definition 6. Let $D \subseteq H$ be a set of operations and S a serialization of D. Let $\mathbf{w}, \mathbf{r} \in D$ be as presented in Definition **1**. We define the set W_r associated with **r**, as:

$$W_r = \{\mathbf{w}' \in D: (\mathbf{w}' \text{ writes a value into object X}) \wedge (\xi(L(\mathbf{w})) < \xi(L(\mathbf{w}')) < (\xi(L(\mathbf{r})) - \Delta))\}$$

We say that operation **r** *occurs or reads on time* in serialization S, if $W_r = \varnothing$. S is *timed* if every **read** operation in S occurs on time. ❑

Therefore, timed consistency requires that if a **write** operation is executed at logical time t, it must be visible at site i before $\xi(t_i) - \xi(t) > \Delta$, where t_i is the logical time of site i.

Consider an example of a mapping ξ for vector clocks: if t is a vector timestamp used in a distributed system with N sites, we can define:

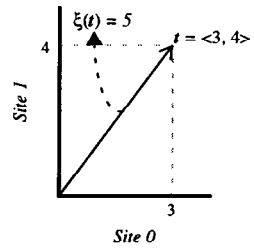
$$\xi(t) = \sum_{i=0}^{N-1} t[i]$$

Thus, $\xi(t)$ is the number of global events that are known to a site when its current logical time is t. This map satisfies Definition **5** and gives us a way to summarize the information conveyed in a vector clock. For instance, if the current logical time of a site is <35, 4, 0, 72>, then this site is aware of 111 global events. Now, if its current copy of object X was written at logical time <2, 1, 0, 18>, then this version of X was created by a site which was aware of 21 global events. For any value of $\Delta < 90$, this object version is either invalidated or marked as old.

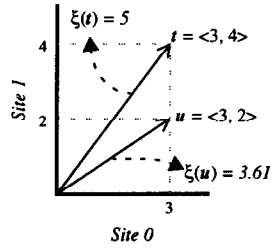
Another interesting version of ξ that allows a geometric interpretation of vector clocks is:

$$\xi(t) = \sqrt{\sum_{i=0}^{N-1} t[i]^2}$$

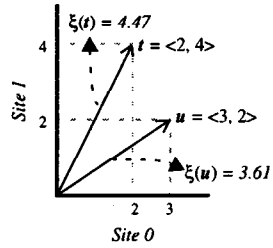
This formula is the length of a vector in a $\Re^N$ vector space. Figure 7 shows some examples for the simplest case with just 2 sites. Figure 7.**a** presents the length for the timestamp <3, 4>. Since timestamp <3, 2> → <3, 4>, then $\xi(<3, 2>) < \xi(<3, 4>)$, i.e. 3.61 < 5, as illustrated by Figure 7.**b**. Furthermore, the area enclosed by the first timestamp is totally covered by the area enclosed by the second timestamp. This is a direct consequence of the definition of "<" for vector clocks [15, 27] and it is valid for larger values of N (but its geometric interpretation is not as evident for N > 3). Conversely, two concurrent timestamps such as <2, 4> and <3, 2> define areas which are not totally contained one into another. However, it is possible to argue that timestamp <2, 4> denotes a global state of the system that is aware of a larger global activity than timestamp <3, 2> and therefore $\xi(<3, 2>) < \xi(<2, 4>)$, i.e. 3.6 < 4.47, as presented in Figure 7.c.



a) Length of a Vector Clock



b) Causally related events



c) Concurrent events

Figure 7. Geometric interpretation of Vector Clocks.

Other examples of the map ξ for vector and plausible clocks are presented in [40].

6 CONCLUSIONS

Timed consistency models examine interesting temporal relationships between objects and sites that form a distributed system, and are able to capture requirements that are not easily expressed by standard consistency models such as **SC** and **CC**. A *timed consistency* model defines a maximum acceptable threshold of time (i.e., parameter Δ) after which the effects of a **write** operation must be available to all the sites of the system.

The value of Δ is the result of a trade-off between the need of perceiving changes to shared objects in a timely fashion and the availability of resources in the system. Small values of Δ require more communications overhead and may decrease the scalability of the system (e.g., in extreme cases, local caches become useless), while large values of Δ require less expensive methods but reduce the timeliness of the information and the actual sharing of information

by the sites. Since such considerations are typical in interactive and collaborative applications, and more recently for dynamic content on the World Wide Web, timed consistency criteria are appropriate to state the requisites of operations ordering and timeliness of these applications.

By combining the requirements of timed consistency and those of consistency criteria such as **SC** and **CC**, we propose **TSC** and **TCC**, which are well suited to meet the needs of many applications. Furthermore, this concept also unifies existing consistency models such as sequential consistency and linearizability. We explore a possible implementation of **TSC** and **TCC** based on the concept of lifetimes of object values. Finally, a possible definition of **TCC** using just logical clocks was presented.

Several issues of this research are to be addressed as part of future work. In order to better understand the relationship between the value of Δ and the cost of accomplishing that particular level of timeliness, we are currently completing detailed simulations (and eventual implementations) of systems based on the consistency criteria described in this paper. Other possible implementations of **TSC** and **TCC** have to be considered. For the case of **TCC** using just logical clocks, it is necessary to explore different mappings from logical timestamps to real numbers, and to provide them with an appropriate semantics for the selection of the parameter Δ.

REFERENCES

[1] S. Adve and M. Hill, "Implementing Sequential Consistency in Cache-based Systems", Proc. of the International Conference on Parallel Processing. Pennsylvania State University, University Park, pp. I-47-I-50.

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

[3] 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.

[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. ICDCS'98, Amsterdam. 1998.

[6] H. Attiya and J. Welch. "Sequential Consistency vs. Linearizability". ACM Transactions on Computer Systems. Vol 12, Number 12. May 1994.

[7] R. Baldoni, A. Mostefaoui and M. Raynal. "Causal delivery of messages with real-time data in unreliable networks". Real-Time Systems, The International Journal of Time-Critical Computing Systems, 10(3), May 1996.

[8] R. Baldoni, R.Prakash, M.Raynal and M. Singhal. "Broadcast with Time and Causality Constraints for Multimedia Applications". Proc. of the 22nd. EUROMICRO Conference, Prague, September 1996.

[9] K. Birman, A. Schiper and P. Stephenson, "Lightweight Causal and Atomic Group Multicast", ACM Transactions on Computer Systems, Vol 9, No. 3, pp. 272-314, Aug. 1991.

[10] P. Cao and C. Liu, "Maintaining Strong Cache Consistency in the World-Wide Web", Proc. of ICDCS'97, pp. 12-21, May 1997.

[11] V. Cate, "Alex - A Global File System", Proceedings of the 1992 USENIX File System Workshop, pp. 1-12, May 1992.

[12] F. Cristian, "Probabilistic Clock Synchronization", Distributed Computing, Vol 3, pp. 146-158. 1989.

[13] R. Drummond and O. Babaoglu, "Low-Cost Clock Synchronization", Distributed Computing, Vol 6, pp. 193-203. 1993.

[14] K.P. Eswaran, J.N. Gray, R. Lorie and I.L. Traiger, "The notion of Consistency and Predicate Locks in a Database System", Communications ACM, Vol 19, No. 11, pp. 624-633, November 1976.

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

[16] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk Nielsen, T. Berners-Lee, "Hypertext Transfer Protocol HTTP/1.1", HTTP Working Group Internet Draft. March 13, 1997.

[17] V.K. Garg and M. Raynal, "Normality: a consisntency criterion for concurrent objects", Parallel Processing Letters, 9(1), March 1999.

[18] K. Gharachorloo and P. Gibbons, "Detecting Violations of Sequential Consistency", Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, pp. 316-326, July 1991.

[19] J. Gwertzman and M. Seltzer, "World-Wide Web Cache Consistency", Proc. of the 1996 USENIX Technical Conference, San Diego, CA. January 1996.

[20] M. Herlihy and J. Wing. "Linearizability: A correctness condition for concurrent objects". ACM Transactions on Program. Lang. Systems. 12, 3. July 1990.

[21] 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.

[22] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems", IEEE Trans. on Computers, vol. C-36, pp. 933-940. August 1987.

[23] 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.

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

[25] L. Lamport, "How to make a Multiprocessor Computer that correctly executes Multiprocess Programs", IEEE Transactions on Computer Systems, C-28(9), 690-691, 1979.

[26] L. Lamport, "Time, clocks and the ordering of events in a Distributed System", Communications of the ACM, vol 21, pp. 558-564, July 1978.

[27] 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.

[28] D. L. Mills, "Internet Time Synchronization: the Network Time Protocol", IEEE Transactions on Communications, Vol. 39, No. 10, pp. 1482-1493, October 1991.

[29] D. L. Mills, "Improved Algorithms for Synchronizing Computer Network Clocks", IEEE Transactions on Networking, Vol. 3, No. 3, pp. 245-254, June 1995.

[30] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates", Journal of ACM, Vol. 26, No. 4, pp. 631-653. October 1979.

[31] M. Raynal, M. Mizuno and M. Nielsen, "Synchronization and Concurrency Measures for Distributed Applications", Proceedings of 12th IEEE International Conference on Distributed Computing Systems, pp. 700-709, Yokohama, Japan, 1992.

[32] 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.

[33] M. Raynal and M. Ahamad, "Exploiting write semantics in implementing partially replicated causal objects", Proceedings of 6th EUROMICRO, Workshop on Parallel and Distributed Processing, pp. 157-163, Madrid, Spain, January 1998.

[34] A. Singla, U. Ramachandran and J. Hodgins, "Temporal Notions of Synchronization and Consistency in Beehive". Proc.of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, June 1997.

[35] S.D. Stoller, "Detecting Global Predicates in Distributed Systems with Clocks". Proc. 11th International Workshop on Distributed Algorithms (WDAG 97). Lecture Notes in Computer Science. Springer-Verlag. 1997

[36] R. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs", Acta Informatica, 19:57-84. 1983.

[37] 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.

[38] F. Torres-Rojas and M. Ahamad. "Computing Minimum and Maximum of Plausible Clocks", Technical Report, College of Computing, Georgia Institute of Technology, 1998.

[39] F. Torres-Rojas, M. Ahamad and M. Raynal, "Lifetime Based Consistency Protocols for Distributed Objects", Proc. 12th International Symposium on Distributed Computing, DISC'98, Andros, Greece, September 1998.

[40] F. Torres-Rojas, M. Ahamad and M. Raynal, "Timed Consistency using Logical Clocks", Technical Report, College of Computing, Georgia Institute of Technology, 1999.

[41] R. West, K. Schwan, I. Tacic and M. Ahamad. "Exploiting Temporal and Spatial Constraints on Distributed Shared Objects". Proc. 17th International Conference on Distributed Computing Systems ICDCS '97. Baltimore, U.S.A. May 1997.