Convergence Through a Weak Consistency Model: Timed Causal Consistency

Francisco J. Torres-Rojas
Centro de Investigación en Computación e Informática Avanzada (*CIenCIA*)
and Instituto Tecnológico de Costa Rica
torres@ic-itcr.ac.cr

and

Esteban Meneses
Centro de Investigaciones en Computación (*CIC*) - Instituto Tecnológico de Costa Rica
and PrediSoft
emeneses@ic-itcr.ac.cr

Costa Rica

Abstract

Given a distributed system with several shared objects and many processes concurrently updating and reading them, it is convenient that the system achieves convergence on the value of these objects. Such property can be guaranteed depending on the consistency model being employed. *Causal Consistency* is a weak consistency model that is easy and cheap to implement. However, due to the lack of real-time considerations, this model cannot offer convergence. A solution for overcoming that problem is to include time aspects within the framework of the model. This is the aim of *Timed Causal Consistency*.

Keywords: Convergence, Weak Consistency, Causal Consistency, Timed Consistency.

1 INTRODUCTION

A distributed application is build up from processes executed at different nodes from possibly distant locations. One important and not trivial problem arises when we want to maintain the consistency of the state shared by such processes. Many consistency protocols have been developed to attack this challenge. Each category is characterized by the type of executions that is permitted or banned by the protocol. However, in some applications it may be not necessary to require all the properties offered by strict consistency models. In its place, in could be better to implement less restrictive (and cheaper) protocols that satisfy the minimum requirements of the particular application.

Causal Consistency [2] is a weak protocol that only accepts executions where the causal order is respected. This model can be implemented at low cost. Besides this, although weak protocols are harder for programmers than strict ones, causal consistency offers a balance where a class of practical programs can be modeled and some useful properties guaranteed.

Convergence is an idea found in multiple areas of science. It is often related to some kind of stability. Although operations made in the past could have created certain disorganization in the arrangement of the analyzed system, there is comfort in knowing that our object of study achieves a stable behavior after a given instant. Consider, for instance, economical indexes after a financial crisis, or descriptive variables in the weather after a hurricane. It is nice to know that, after some time, prices will be under control, and that sunny afternoons may be enjoyed. These conditions will be "well behaved" until some other set of events strikes our system. At least for some range of time we are capable of understanding the state of our system.

In groupware systems (e.g., a collaborative editing system where many users are concurrently working over a document), it is fundamental to offer convergence. In this context, some authors [4, 13] define convergence by looking at the final result of a work session. Operations made by users could arrive at different times to the other sites, executing possibly in different orders. However, it is required that the final result be exactly the same for every user. It is also important offering convergence in mobile computing applications

[6], specially when disconnection periods are considered. In this case, after operations (possibly conflicting) are made over different replicas of the same object, it is required that all replicas converge to the same state after all the processes have been reconnected for sufficiently long.

Nevertheless, convergence is **not** a property inherent to causally consistent executions. *Timed Causal Consistency* is a protocol where time constraints are added to the causal consistency model. Ordering and timeliness are two facets of consistency protocols ([3, 15]). The ordering aspect defines the possible orders in which operations can be executed and perceived by the participant sites, while the timeliness defines how soon the effects of a operation in some process are known by the other processes.

In Section 2, the principal concepts of consistency protocols are revisited. The timed strategy is presented in Section 3, while the convergence model is developed in Section 4. Analysis about convergence and timed causal consistency is provided in Section 5. Conclusions and future work were left for Section 6.

2 CONSISTENCY MODELS REVISITED

A distributed system consists of N user processes and a distributed data storage. Because of caching and replication, several, possibly different, copies of the same data objects might coexist at different sites of the system. Thus, a consistency model, understood as a contract between processes and the data storage, must be provided. There are multiple consistency models [1, 2, 3, 8, 10, 14, 15].

The *global history* $\mathcal{H}$ of this system is the partially ordered set of all operations occurring at all sites. $\mathcal{H}_i$ is the total ordered set or sequence of operations that are executed on site i. If **a** occurs before **b** in $\mathcal{H}_i$ we say that **a** precedes **b** in *program order*, and denote this as $\mathbf{a} <_{\texttt{PROG}} \mathbf{b}$. In order to simplify, we assume that all operations are either **read** or **write**, that each value written is unique, and that all the objects have an initial value of zero. These operations take a finite, non-zero time to execute, so there is a time elapsed from the instant when a **read** or **write** "starts" to the moment when such operation "finishes". Nevertheless, for the purposes of this paper, we associate an instant to each operation, called the *effective time* of the operation. We will say that **a** is *executed* at time t if the effective time of **a** is t. If **a** has an effective time previous to the effective time of **b** we denote this as $\mathbf{a} <_{\texttt{E-T}} \mathbf{b}$. Let $\mathcal{H}_{i+w}$ be the set of all the operations in $\mathcal{H}_i$ plus all the **write** operations in $\mathcal{H}$. The partially ordered *happens-before* relationship "$\rightarrow$" for message passing systems as defined in [9] can be modified to order the operations of $\mathcal{H}$. Let **a**,**b** and $\mathbf{c} \in \mathcal{H}$, we say that $\mathbf{a} \rightarrow \mathbf{b}$, i.e., **a** happens-before (or *causally precedes*) **b**, if one of the following holds:

1. **a** and **b** are executed on the same site and **a** is executed before **b**.
2. **b** reads an object value written by **a**.
3. $\mathbf{a} \rightarrow \mathbf{c}$ and $\mathbf{c} \rightarrow \mathbf{b}$.

Two distinct operations **a** and **b** are *concurrent* if none of these conditions hold between them.

If $\mathcal{D}$ is a set of operations, then a *serialization* of $\mathcal{D}$ is a linear sequence S containing exactly all the operations of $\mathcal{D}$ 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 the same object. If $\prec$ is an arbitrary partially ordered relation over $\mathcal{D}$, we say that serialization S *respects* $\prec$ if $\forall$ **a**, $\mathbf{b} \in \mathcal{D}$ such that $\mathbf{a} \prec \mathbf{b}$ then **a** precedes **b** in S.

Intuitively, one would like that any **read** on a data item `X` returns a value corresponding to the results of the most recent **write** on `X`. In some systems this could mean that after making an update, all other processes may be notified about the change as soon as it is required. Assuming the existence of absolute global time, this behavior can be modeled with *linearizability* [8]:

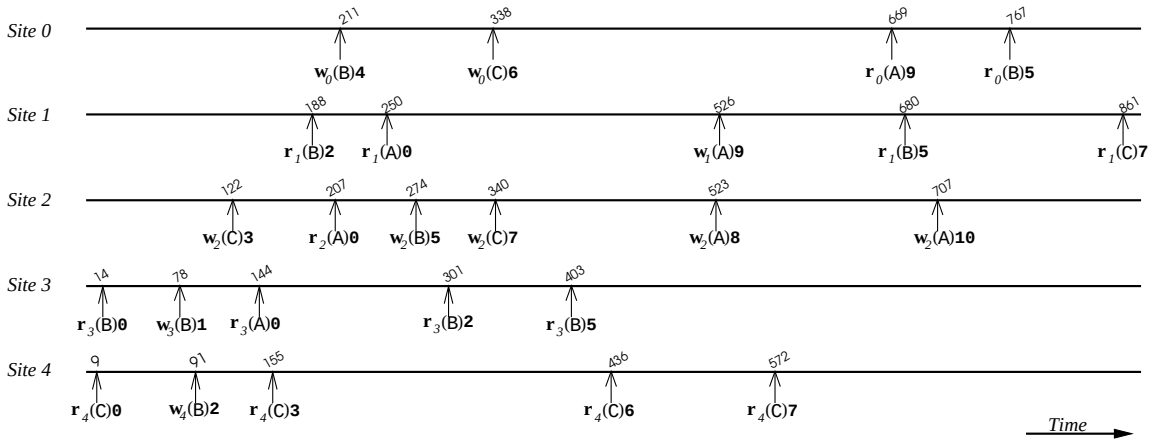
Definition 1 *History* $\mathcal{H}$ *satisfies* Linearizability (**LIN**) *if there is a serialization* S *of* $\mathcal{H}$ *that respects the order* $<_{\texttt{E-T}}$ [8].

A weaker, but more efficient, model of consistency is *sequential consistency* as defined by Lamport in [10]:

Definition 2 *History* $\mathcal{H}$ *satisfies* Sequential Consistency (**SC**) *if there is a serialization* S *of* $\mathcal{H}$ *that respects the order* $<_{\texttt{PROG}}$ *for every site in the system* [10].

SC 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. For instance, History $\mathcal{H}$ presented in 1.**a)** is sequentially consistent, because 1.**b)** shows the required serialization S. Although **SC** can be implemented in a more efficient way than **LIN** and it is a programmer-friendly model, it has been shown that **SC** has performance problems [1, 14].

An even weaker model of consistency is *causal consistency* [2].

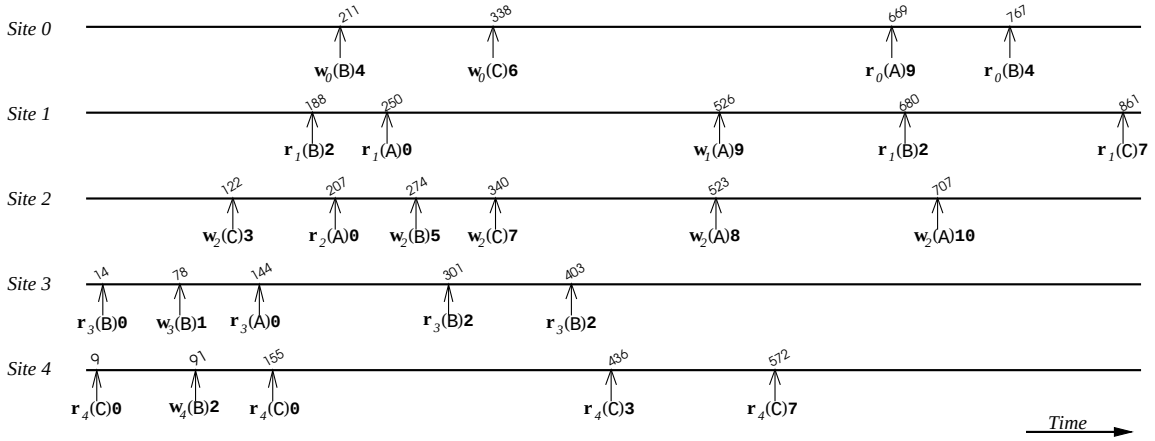


a) Sequentially Consistent Execution

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

b) Serialization that respects program order

Figure 1: Distributed history compliant with sequential consistency



a) Causally Consistent Execution

S_0 $w_4(B)2$ $[w_0(B)4\ w_0(C)6]$ $w_1(A)9$ $[r_0(A)9\ r_0(B)4]$ $w_2(C)3\ w_2(B)5\ w_2(C)7\ w_2(A)8\ w_2(A)10\ w_3(B)1$

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 2: A causally consistent distributed history

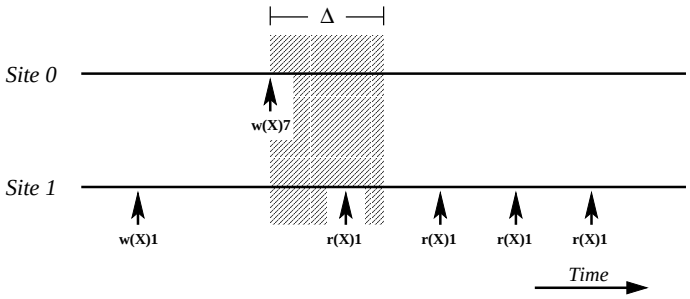


Figure 3: A non-timed sequentially consistent execution

Definition 3 *History $\mathcal{H}$ satisfies* Causal Consistency (**CC**) *if for each site i there is a serialization S_i of the set $\mathcal{H}_{i+w}$ that respects causal order "$\rightarrow$"* [2].

Thus, if **a**,**b** and **c** $\in \mathcal{H}$ are such that **a** writes value **v** in object `X`, **c** reads the same value **v** from object `X`, and **b** writes value **v'** into object `X`, it is never the case that **a** $\rightarrow$ **b** $\rightarrow$ **c**. **CC** requires that all causally related operations be seen in the same order by all sites, while different sites could perceive concurrent operations in different orders. For example, if figure 2 we have a distributed history in 2.**a** which is causally consistent, due to serializations S_i presented in 2.**b**. Note that in 2.**b** there is a serialization S_i for site i and the local events in S_i are distinguished by a surrounding rectangle.

CC is a model of consistency weaker than **SC**, but it can be implemented efficiently [2, 14]. Such implementation requires keeping track of which processes have seen which **write** events. In fact, there is dependency graph for determining which operation is dependent on which other operations. So, this data structure must be built and maintained. For fulfilling this need, vector clocks [11] can be used.

3 TIMED CONSISTENCY MODEL

In neither **SC** nor **CC** real-time is explicitly captured, i.e., in the serializations of $\mathcal{H}$ or $\mathcal{H}_{i+w}$ operations may appear out of order in relation to their effective times. For instance, in Figure 1 the serialization in part **b)** shows event $\mathbf{r}_1(\mathtt{B})\mathbf{5}$ occurring before $\mathbf{w}_2(\mathtt{A})\mathbf{8}$, but the latter event occurred at time 523, while the former occurred at time 680. In **CC**, each site can see concurrent **write** operations in different orders. On the other hand, **LIN** requires that the operations be observed in their real-time ordering. 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 (**TC**) as proposed in [15] 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. It can be seen that when $\Delta = 0$, then **TC** becomes **LIN**. So, **TC** can be considered as a generalization or weakening of **LIN**.

The execution showed in Figure 3 satisfies **SC** and **CC**. Up to the second operation of Site *1*, the execution satisfies **TC** for the value of Δ presented 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** into 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 **writes**. 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. Let $T(\mathbf{a})$ be the real-time instant corresponding to the effective time of operation **a**.

Definition 4 *Let $\mathcal{D} \subseteq \mathcal{H}$ be a set of operations and S a serialization of $\mathcal{D}$. Let* **w**, **r** $\in \mathcal{D}$ *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 $\mathcal{W}_{\mathbf{r}}$, associated with* **r**, *as: $\mathcal{W}_{\mathbf{r}}$=*{**w'** $\in \mathcal{D}$ | (**w'** *writes a value into object* `X`*)* $\wedge$ *(T(**w**) < T(**w'**) < T(**r**) - Δ)}. *We say that operation* **r** ***occurs*** *or* ***reads on time*** *in serialization S, if $\mathcal{W}_{\mathbf{r}} = \varnothing$. S is* **timed** *if every* **read** *operation in S occurs on time.*

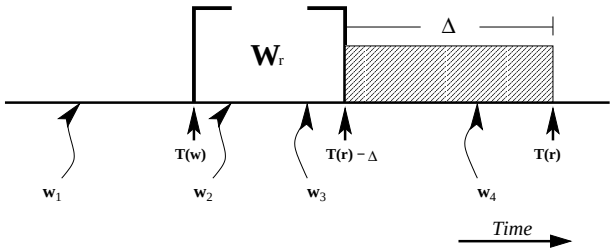


Figure 4: Operation **r** does not read on time

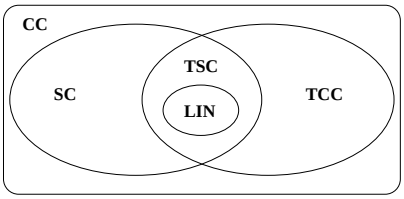


Figure 5: Hierarchy of consistency models

Figure 4 illustrates Definition 4, 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 then 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 $\mathcal{W}_{\boldsymbol{r}}$, and, therefore, operation **r** does not occur on time. The area between $T(\mathbf{w})$ and $T(\mathbf{r})$-Δ represents the interval of time associated with the set $\mathcal{W}_{\boldsymbol{r}}$, which according to definition 4 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).

Definition 5 *Let* $\mathbf{a}, \mathbf{b} \in \mathcal{D} \subseteq \mathcal{H}$ *with effective times* t_1 *and* t_2*, respectively, be two operations over the same object* `X`*. We say that* $\mathbf{a} <_\Delta \mathbf{b}$ *if:*

1. *Both* **a** *and* **b** *are* **write** *operations and* $t_1 < t_2$*, or*
2. **a** *is a* **write** *operation,* **b** *is a* **read** *operation and* $t_1 < (t_2 - \Delta)$*.*

Definition 6 *History* $\mathcal{H}$ *satisfies* Timed Consistency (**TC**) *if there is a serialization* S *of* $\mathcal{H}$ *that respects the partial order* $<_\Delta$ [15]*.*

3.2 Timed Sequential Consistency and Timed Causal Consistency

Now, we combine the requirements of well-known consistency models such as **SC** and **CC** with the requirement of reading on time.

Definition 7 *History* $\mathcal{H}$ *satisfies* Timed Sequential Consistency (**TSC**) *if there is a serialization* S *of* $\mathcal{H}$ *that simultaneously respects the partial order* $<_{PROG}$ *and the partial order* $<_\Delta$ [15]*.*

Definition 8 *History* $\mathcal{H}$ *satisfies* Timed Causal Consistency (**TCC**) *if for each site* i *there is a timed serialization* S_{i} *of* $\mathcal{H}_{\mathrm{i}+\boldsymbol{w}}$ *that simultaneously respects causal order* $\rightarrow$ *and the partial order* $<_\Delta$ [15]*.*

Figure 5 presents a hierarchy of different consistency models. Every linearizable execution is also sequentially consistent, but the opposite is not necessarily true. Similarly, every sequentially consistent execution is causally consistent, while the contrary is not always true. The proofs for these results and some implementation details can be found in [15]. The idea behind these definitions is to show how it is possible to offer flexibility for the consistency protocol in our model; without losing the real-time considerations, consistency aspects can be relaxed, passing from **SC** to **CC**.

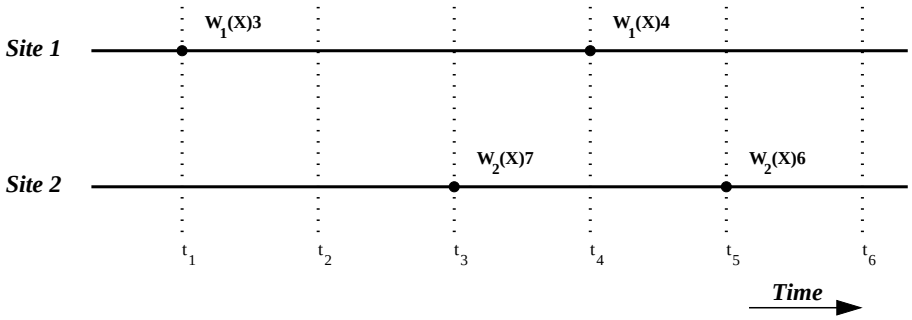


Figure 6: Two sites in a convergent execution

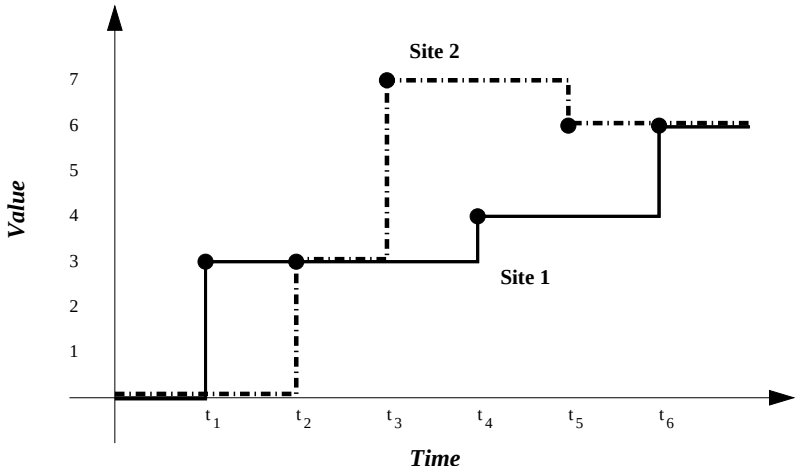


Figure 7: Convergence of two sites

4 CONVERGENCE MODEL

In this section, we present our approach for analyzing convergence in consistency protocols for distributed systems.

Consider the distributed history shown in Figure 6. After `X` is updated by one of the sites, the new value is communicated, with some delay, to the other site. Site *1* updates `X` at time t_1 giving it the value `3` (which was `0` initially). By that time, Site *2* has no knowledge of this change, so it surely believes that `X` still has `0` value. It is not until time t_2 that Site *2* discovers that `X` has been updated. However, at time t_3 Site *2* makes a new change to `X`, giving it the value of `7`. Let's say that news of this change arrive too late to Site *1*, and by time t_4, Site *1* has updated again `X` to value `4`. Similarly, Site *2* does not perceive this last change and updates `X` at time t_5 to value `6`. At time t_6 Site *1* realizes that `X` has a new value and from here on, both sites agree on the value of `X`. Thus, finally, convergence has been reached.

Figure 7 plots the values that `X` takes at every instant, as perceived by each site, and it shows that there is convergence after time t_6. Nevertheless, it could be claimed that the time intervals $[0,t_1]$,$[t_2,t_3]$ and $[t_6,+\infty]$ form a set of ranges where convergence was achieved, we refer to these time ranges as *convergence frames* (see Section 4.3). Our formal definition for convergence will capture these two kinds of stability ranges: the one obtained after the *last* update to some distributed object, and the time frames where two or more sites agree on the same value for a particular object.

4.1 Trivial Convergence

It is possible that a distributed system achieves convergence over some object `X` if all sites have previously agreed in setting a particular value for such object. This is the less interesting case for convergence, since a situation like this hardly represents the general case, and even though the system is reaching convergence in the value of `X`, this does not imply correctness in the execution.

Definition 9 *An execution in a distributed system is* **trivially convergent over object** `X` *if all sites have agreed to assign a particular value for* `X` *after time* t*. If the execution is trivially convergent over every*

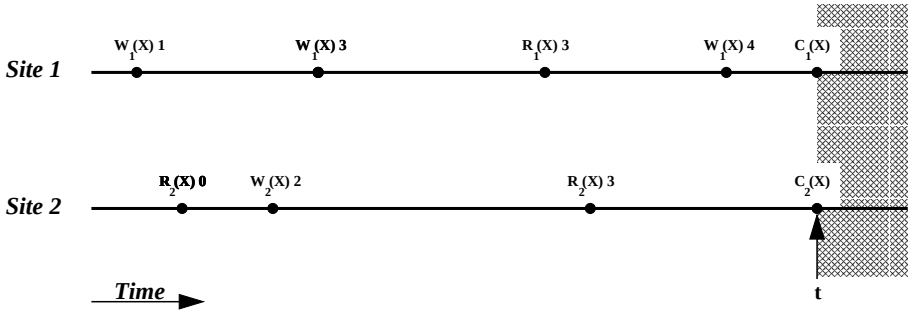


Figure 8: Convergent cut

possible object, we say that the execution is **trivially convergent**.

4.2 Absolute Convergence

Let's analyze convergence after the "last" **write** operation to a particular object. Figure 8 shows a simple distributed computation, with 2 sites and one shared object `X`. There are a series of **writes** and **reads** executed by both sites, and, at several times during execution, sites see different values for object `X`. But, at some time t after event $\mathbf{w}_1(\mathtt{X})4$, which happens to be the last actualization to `X` in the whole execution, all **reads** to object `X` executed by any site *should* return the value `4`. Thus, after time t, this system has converged regarding the value of object `X`. The intuition behind *absolute convergence* is that, at the end of the day, after the **writes** stop, every site involved in a distributed computation will agree on the same values for the same objects.

Following the lines of *consistent cuts* [11], we define a *convergent cut* this way:

Definition 10 *We say that a* **convergent cut over object** `X` *is a set of phantom events* $\mathcal{C}=\{C_1,C_2,...,C_N\}$, *where every* C_i *is inserted in local history* $\mathcal{H}_i$, *all at the same time* t. *All the* C_i *are* **read** *operations over* `X` *that would return exactly the value written by the latest* **write** *into object* `X` *that occurred before* t.

Definition 11 *An execution in a distributed system is* **absolutely convergent over object** `X` *if at any arbitrary time after* t, *which itself occurs after the last* **write** *to object* `X`, *a* convergent cut over object `X` *can be inserted. If the execution is absolutely convergent over every possible distributed object, we say that the execution is* **absolutely convergent**.

If an execution is absolutely convergent over object `X` at time t, i.e., we were able to insert a convergent cut $\mathcal{C}$ at time t, it must be true that the same cut $\mathcal{C}$ can be inserted, with identical results, at any time $u > t$. Now, if there are M shared objects $\mathtt{X}_j$, $1 \leq j \leq M$, and the execution is absolutely convergent for all the M objects, then for every object we associate a minimum time t_j where its corresponding convergence cut can be inserted. Therefore, the distributed system is absolutely convergent at any time after $t = max(t_1, t_2, ..., t_M)$.

4.3 δ-Convergence

It is typically desirable that the lapse before an update is communicated to everybody else in a distributed system be as short as possible. However, in a very active system with frequent **writes** to the same shared objects, it is normal that the values of these objects diverge during execution. Even under these circumstances, we could expect that after the "last" **write**, as mentioned in the previous section, the system reaches *absolute convergence*. Besides, if the system, after considering factors such as overhead, communication delays, and consistency protocols, can guarantee that an update is known to the complete system (either by updating or by invalidations) in at most δ units of time, the execution might manifest intervals where the system is evidently convergent in relation to some objects.

Now, we claim that if the lapses between multiple consecutive **writes** to the same object are shorter than the parameter δ, there was not enough time for propagating the values set by all the **writes**, but the system can still be classified as convergent. Conversely, if two consecutive **writes** to the same object `X` occur more than δ units apart and we are **not** able to insert a convergent cut for this object at least δ units of times after the first **write**, the system is not convergent. This is the intuition of what we called δ-convergence:

Definition 12 *An execution satisfies* δ**-convergence** *if it can be guaranteed that, at any time when the lapse between two consecutive* **writes** *to the same object* `X` *is greater than* δ *units of time, a convergent cut over* `X` *can be inserted* δ *units of time after the first* ***write***.

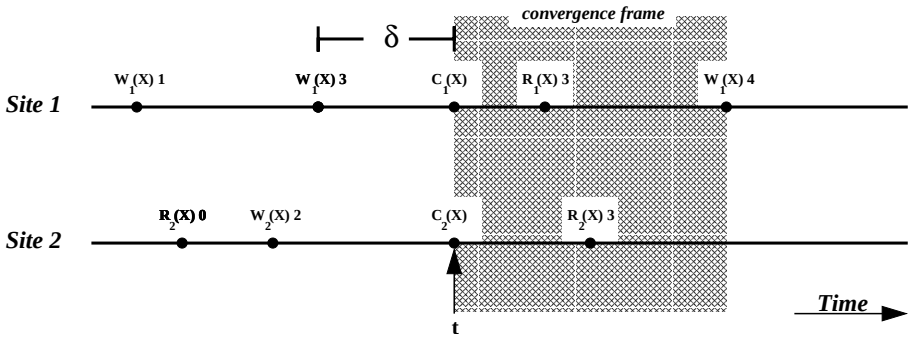


Figure 9: Convergence frame

Thus, in a δ-convergent execution, if X is updated at time t and the next update to this object, anywhere in the system, occurs at time u, with $t + \delta < u$, there is an interval $[t + \delta, u]$ where all sites in the system would perceive, if they read it, the very same value for object X. We call this interval a *convergence frame* for object X. On the other hand, if $t + \delta > u$, we might not define such a convergence frame, but still claim that the system is δ-convergent. In other words, the system is allowed to be "unstable" for at most δ units of time after a **write**, without being considered non-convergent.

Figure 9 shows some of the previous concepts. If δ units of time after operation $\mathbf{w}_1(\mathtt{X})3$ occurred, we are able to insert a convergent cut associated to object X (which means that if every site in the system would read X all they would find the same value), this establishes a convergence frame for object X. Of course, another update to object X can be made thereafter, but until that new update the system has converged on the value of X. Extending this concept to several objects is straightforward.

5 CAUSAL CONSISTENCY AND CONVERGENCE

As it was mentioned in definition 3, **CC** does not require that each **read** event over object X returns the latest value written. It merely needs to construct serialization S_i, for each site i, that respects causal order "$\rightarrow$". Thus, **CC** does not offer convergence *per se*. Due to the lack of real-time restrictions in its definition, sites are not obliged to update or invalidate their local objects unless that it is required for building the serializations S_i.

Figure 10 presents a simple example of a distributed execution that satisfies **CC**, but whose shared objects never converge. Notice that this history is not compliant with **SC** either. Site *2* writes value 1 into object X, and, some time later, Site *1* writes value 2 into object X. The following **read** on Site *1* returns 1, while the next **read** operation on Site *2* returns value 2. Finally, Site *1* writes value 3 into object X.

Considering only these five events, we prove that the history, so far, satisfies **CC** by taking the following serializations: $S_1 = \mathbf{w}_1(\mathtt{X})2, \mathbf{w}_2(\mathtt{X})1, \mathbf{r}_1(\mathtt{X})1, \mathbf{w}_1(\mathtt{X})3$ and $S_2 = \mathbf{w}_2(\mathtt{X})1, \mathbf{w}_1(\mathtt{X})2, \mathbf{r}_2(\mathtt{X})2, \mathbf{w}_1(\mathtt{X})3$. These serializations do not respect real-time, but fulfill the requirements for **CC** [2]. Now, nothing forces Site *1* nor Site *2* to agree, at any point in the future, on the value of X. Consider operation sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, both containing just **reads** on object X, one executing on Site *1* and the other executing on Site *2*, respectively. The value retrieved by operations in $\mathcal{Q}_1$ is 3, while the old value retrieved by operations in $\mathcal{Q}_2$ is 2. It can be proved by induction over $|\mathcal{Q}_1|$ and $|\mathcal{Q}_2|$ that this distributed history is causally consistent. A possible serialization set would be: $S_1 = \mathbf{w}_1(\mathtt{X})2, \mathbf{w}_2(\mathtt{X})1, \mathbf{r}_1(\mathtt{X})1, \mathbf{w}_1(\mathtt{X})3, \{\mathcal{Q}_1\}$ and $S_2 = \mathbf{w}_2(\mathtt{X})1, \mathbf{w}_1(\mathtt{X})2, \mathbf{r}_2(\mathtt{X})2, \{\mathcal{Q}_2\}, \mathbf{w}_1(\mathtt{X})3$. Thus, Sites *1* and *2* can execute an arbitrarily large number of **reads** of object X, satisfying **CC**, and never converging to the same value. In this example, *absolute convergence* is not guaranteed by **CC**, and, if we choose $\mathbf{w}_1(\mathtt{X})2$ and $\mathbf{w}_2(\mathtt{X})1$ as occurring more than δ units of time apart, neither δ-convergence is satisfied. It is easy to build an example more complex than Figure 10, involving multiple sites, shared objects and values written, where **CC** is respected, and where absolute convergence and δ-convergence are never met.

On the other hand, we claim that **TCC** guarantees convergence. Remember that **TCC** 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.

Figure 11 shows the same execution of figure 10 but includes the requirements of **TCC**. Then, it can be observed that Δ time units after the event $\mathbf{w}_1(\mathtt{X})3$ Site *2* must be aware of the changes done over object X. Then, a convergent cut can be placed at this moment and, after that, the sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$ will report the same read value 3. We can generalize this concept into the next theorem.

Theorem 1 ***TCC*** *satisfies* absolute convergence *and* δ-convergence.

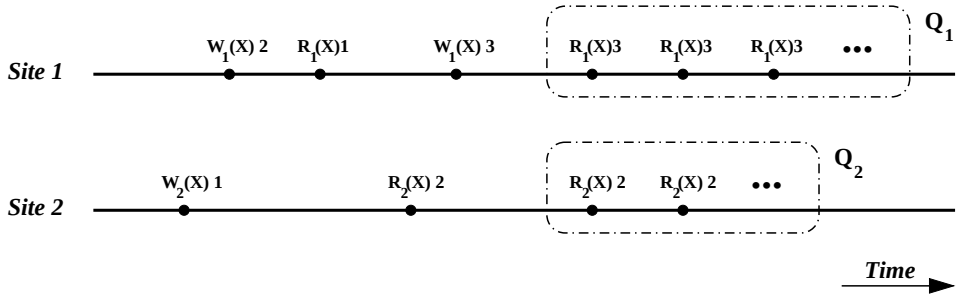


Figure 10: A causally consistent history that doesn't converge

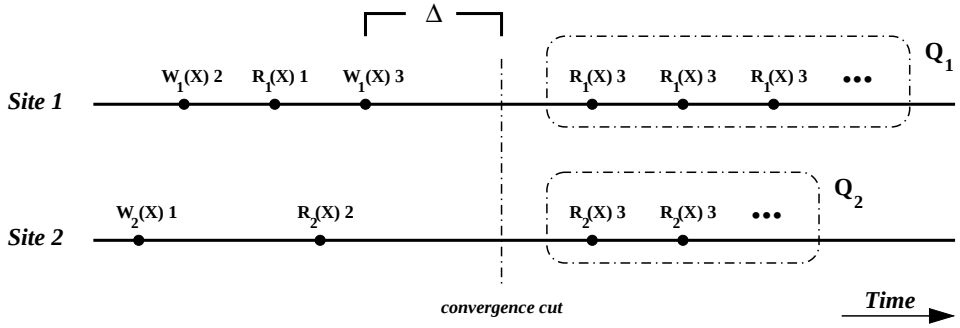


Figure 11: A timed causally consistent history

Proof At most Δ units of time after the last **write** for every shared object, **TCC** guarantees that the updated value is known to every site in the distributed system, therefore, we can insert a convergent cut over each shared object Δ units of time after the corresponding last **write** operation, which according to Definition 11 proves that the execution satisfies *absolute convergence*. Now, it should be easy to see that δ-convergence is guaranteed for the value $\delta = \Delta$.

Both, absolute convergence and δ-convergence, are desirable properties among distributed systems applications, such as collaborative software and mobile computing. Recent research in these areas ([7],[12]) show that **CC** is a good alternative for maintaining consistency. Then, **TCC** which is a strengthening of **CC**, is a good candidate to be applied in those contexts.

6 CONCLUSIONS AND FUTURE WORK

The *causal consistency* model gives a weakening of *sequential consistency*, making a difference between events that are potentially causally related and those that are not. Its implementation is easier and cheaper than more strict models (like **LIN** or **SC**). Although **CC** does not guarantee convergence property for the values of the shared objects in the system, it is possible to enrich this model with time considerations. Besides this, a timed consistency model provides a mix of two facets of consistency: order and time. **TCC** can be conceived as a promising candidate to provide convergence at low implementation costs.

Many distributed system applications require convergence as one of their most important properties. In those areas, **TCC** can be introduced to provide such objective, assuring at the same time that causally related updates will be respected by all processes.

In the short term, we are building a distributed shared memory system where diverse consistency protocols, timed and not timed, are implemented, tested and compared.

References

[1] Adve, S. and Gharachorloo, K. *Shared Memory Consistency Models: A Tutorial*. Western Research Laboratory, Research Report 95/7, 1995.

[2] Ahamad, M. et al. *Causal memory: definitions, implementation and programming*. Distributed Computing. September, 1995.

[3] Ahamad, M. and Raynal, M. *Ordering and Timeliness: Two Facets of Consistency?*, Future Directions in Distributed Computing, 2003.

[4] Ellis, C.A. and Gibbs, S.J. *Concurrency Control in Groupware Systems*. In ACM SIGMOD'89 proceedings,pages 399-407, 1989.

[5] Ellis, C.A., Gibbs, S.J. and Rein, G.L. *Groupware: some issues and experiences*. Communications of the ACM, Vol 34(1), January, 1991.

[6] Guerraoui, R. and Hari, C. *On the Consistency Problem in Mobile Distributed Computing*. ACM POMC, 2002.

[7] Galli, R. and Luo Y. *Mu3D: A Causal Consistency Protocol for a Collaborative VRML Editor*. VRML, Monterrey, California, USA, 2000.

[8] Herlihy, M. and Wing, J. *Linearizability: A Correctness Condition for Concurrent Objects*. ACM Transactions on Programming Languages and Systems. Vol 12(3), July 1990.

[9] Lamport, L. *Time, Clocks and the Ordering of Events is a Distributed System*. Communications of the ACM, vol 21, July, 1978.

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

[11] Mattern, F. *Virtual Time and Global States of Distributed Systems*. Proceedings of the International Workshop on Parallel and Distributed Algorithms, 215-226, 1989.

[12] Ram, D. Janaki et al. *Causal Consistency in Mobile Environment*. URL: http://lotus.iitm.ac.in.

[13] Sun, C. et al. *Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems*. ACM Transactions in Computer-Human Interaction, 5(1):63-108, 1998.

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

[15] Torres-Rojas, F. J., Ahamad, M. and Raynal, M. *Timed Consistency for Shared Distributed Objects*. Annual ACM Symposium on Principles of Distributed Computing PODC'99, Atlanta, Georgia, 1999.