Shared State Consistency for Time-sensitive Distributed Applications *

Vijaykumar Krishnaswamy †
Mustaque Ahamad
{kv,mustaq}@cc.gatech.edu
Georgia Institute of Technology
USA

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

David Bakken
bakken@eecs.wsu.edu
Washington State U.
Pullman, WA
USA

Abstract

Distributed applications that share dynamically changing state are increasingly being deployed in wide-area environments. Such applications must access the state in a consistent manner, but the consistency requirements vary significantly from other systems. For example, shared memory models such as sequential consistency focus on the ordering of operations and the same level of consistency is provided to each process. In interactive distributed applications, the timeliness of updates becoming effective could be an extremely important consistency requirement and it could be different across different users. We propose a system that provides both non-timed and time sensitive read and write operations for manipulating dynamic shared state. For example, a timed read can be used by a process to read a recently written value whereas a timed write can make a new value available to all readers within a certain amount of time. We develop a consistency model that precisely defines the semantics of timed and non-timed read and write operations. A protocol that implements this model is also presented. We also describe an implementation and some performance measurements.

Keywords: Consistency model, Consistency protocol, Timeliness, Ordering, Caching.

*This work was supported in part by NSF grant CCR-9619371 and the NSF-INRIA grant INT-9724774.
†Contact author: Email – kv@cc.gatech.edu, Phone – (404) 894-6169, Fax – (404) 894-9442.

1 Introduction

As computers become pervasive and homes become better connected, several new applications will emerge. Such applications will allow remotely located users and services to share information that is created and updated dynamically. Examples of such applications include distributed games, auctions, planning for emergencies and a range of other activities that involve same time interactions across distributed users. Such applications present a number of challenges for the systems that will support them. Because of the interactive nature of the applications, user actions that access the application state must be completed in a timely fashion. If users are not to perceive unacceptable delays, response time for such actions must be low (in milliseconds). It is a real challenge to provide such response time in wide area systems where communication latencies can be very high and can vary across different users.

Many of the user actions in the applications will manipulate the application's shared state. For example, in a distributed game, the positions of various entities, which change with time, could be represented as shared objects. Thus, the system can support such applications by providing shared objects across distributed users. To meet the response time requirements, the state can be replicated or cached at or close to user nodes. The users action can be completed with such a near by copy. If a user action results in an update to shared state, the new value can be propagated to users according to their timeliness needs. Other users may be willing to work with a recent (but not the most current) state of the objects and can choose to "refresh" it when it is necessary or convenient. Such flexible and adaptable coupling between the object copies at different users is desirable because of a number of reasons. First, users may play different roles in an application and their timeliness requirements may be different. Second, the achievable timeliness could depend on communication resources available to a user. If the amount of available resources drops too low to support the requested level of timeliness, many users will prefer to lower their requested timeliness and have it met, rather than having the timeliness be violated, perhaps in unpredictable ways.

In a wide area system, to combat latencies, a shared object system will necessarily employ techniques such as replication and caching to provide fast response time for user requests. In such systems, consistency must be maintained across multiple copies. We claim that a shared object system that addresses timeliness for updating the replicated copies is crucial for supporting interactive applications. Because the applications may both read and write the state, the system must support both of these operations. Furthermore, the system must support *timed reads* and *timed writes*. A timed read can be used by a user to inform the system that it desires to access a recent value of the object. A timed write allows the producer of a new value to inform its potential readers that the new value has become available.

Consider a stock quote update application. If the stock price changes drastically, the write that reflects the change can be made timed and all consumers can be informed of the change within a certain time interval. On the other hand, small changes need not be disseminated quickly and users, based on their requirements and available resources, can request them using timed reads. In a distributed game, when one entity moves to a new region, a timed write can be used to update its state. On the other hand, incremental movements within a region can be disseminated lazily.

Such application requirements motivate four operations: read(), write(), timed-read() and timed-write(). The timed operations address the timeliness needs explicitly while others may be used to access values when there are no timeliness constraints. By including both timed read and timed write, we provide the mechanisms for both push and pull style dissemination of updates. Applications can use them based on a policy suited to their needs and the resources available to them.

Our shared state system supports both timed and non-timed read and writes. Since these operations may be executed with replicated or cached copies, it is necessary to develop a precise consistency model for the system when users can use all of these operations. The focus of existing consistency models is primarily on ordering of reads and writes [13, 1]. Other models only address timeliness [7]. Our goal is to develop a precise model that characterizes consistency when processes use both timed and non-timed operations. For the timed operations, a timeliness interval Δ is defined and a clock system must be assumed. Informally, any read of object x that follows a timed write after Δ time will return its value or a more recent value for the object. Our model characterizes both time independent and time dependent consistency requirements and shows how they can be combined to define a model that is well suited for time sensitive applications.

In addition to the consistency model, we also present a protocol that can maintain the consistency of object copies as required by the model. The protocol allows different timeliness for different users. The protocol has been implemented in a distributed object framework called Quality Objects (QuO) [19], which provides a quality of service layer on top of CORBA. We use this system to experimentally evaluate the protocol, and study the impact of timeliness threshold on the communication resources required as well as the response time experienced by user actions. For example, we show that when timeliness is set to 5 seconds, response time almost as efficient as a local invocation can be achieved even for workloads with very high proportion of timed operations.

We briefly review existing consistency models in Sect. 2. We present a new *time sensitive (TS)* consistency model in Sect. 3. Section 4 describes the protocol that has been developed to support timed and non-timed operations. Its implementation and performance results are presented in Sect. 5, and the paper is concluded in Sect. 6.

2 Consistency Models

A variety of systems such as distributed shared memory, distributed file systems and the world-wide web, allow distributed users to access shared data that is cached or replicated at multiple sites. A number of consistency models have been developed to define the behavior of concurrent read and write operations that are executed with the copies of such shared data. If strong guarantees are provided by the model, programming with the system is easier because the semantics of read and write operations is closer to a shared memory system. On the other hand, weaker consistency models offer efficient implementations.

Many of the existing consistency models can be characterized by the type of orderings they require for read and write operations executed by different processes. Sequential consistency (SC) [13] requires that all operations appear to execute in a serial order that respects the order of operations at each processor. Linearizability [8] requires a stronger serial order that must also respect the time induced order between operations of different processes. Weaker ordering are explored in models such as causal consistency [1] and PRAM [6]. Distributed shared memory systems have exploited synchronization operations to weaken the ordering constraints for data operations. Example of such models include release consistency [4], lazy release consistency [9] and entry consistency [2]. Distributed file systems such as xFS [3] provide strong ordering by ensuring that a single writer or multiple readers are able to access a file at a given time. Other systems permit weaker orderings to deal with disconnections or to improve performance or availability (e.g., Bayou [18], Coda [10]).

Although orderings of writes is important to determine the value of a shared data item, in many interactive applications, how quickly write operations become effective is also important.

For example, timeliness has been explored for dynamic content in the web [14]. Ordering and timeliness are orthogonal requirements that can be used to define a consistency model. For example, SC requires strong ordering but places no constraints on timeliness. The time sensitive (TS) consistency model we propose here generalizes the timed consistency model previously developed by us [5]. In particular, we allow timeliness constraints to be defined for some read and write operations while other operations may not have any timeliness requirements. It is also possible to combine various ordering and timeliness requirements to create several different consistency models. Thus, the TS consistency model not only provides enhanced flexibility but also allows different sites to observe different levels of consistency based on their application needs or the resources available to them for maintaining consistency.

3 TS Consistency Model

3.1 Notations

We assume a distributed system that is composed of a finite set of sequential processes $P_1, \ldots, P_n$ that interact via a finite set X of shared objects. Each object $x \in X$ can be accessed by read and write operations. These operations can either be timed or non-timed. Although the timeliness threshold could be different for different operations, for simplicity, we assume it to be Δ for all timed operations. A non-timed write (w) and a timed-write (tw) into an object defines a new value for the object. Similarly the non-timed read (r) and timed-read (tr) operations obtain a value of the object. In the following discussion, the term timed will be explicitly used to denote timed operations, if not it is to be assumed that the operations are non-timed. A write of value v into object x by process P_i is denoted $w_i(x)v$ and the notation for a timed-write is $tw_i(x)v$. Similarly a read of x by process P_j is denoted $r_j(x)v$ and the timed-read as $tr_j(x)v$ where v is the value returned by the read or the timed-read operation.

In situations where a timed-read is not distinguishable from a read operation, R can be used to represent either of them. Similarly W represents either a write or a timed-write. *op* will denote either R (read) or W (write). The function *T(op)* returns the real-time at which the operation *op* got executed. We assume synchronized global clocks. Hence the function *T()* will be unique across all processes. Also, the timeline for the execution sequence at all processes start at T_{Start}, though the operations may not immediately commence at that time. All the objects values are initialized to zero at the start by an initial fictitious write operation. For simplicity, as in [16, 17], we assume all values written into an object x are distinct[1]. Moreover, the parameters of an operation are omitted when they are not important.

3.2 Semantics of Operations

A timed-write (tw) into an object defines a new value for the object and also ensures that this value is perceived at all the processes no later than Δ time units after the completion of the write. A timed-read (tr) obtains a value of the object that is no older than the the "most recent" value at time t - Δ where t is the execution time of tr. The operations write (w) and read (r) make no such time based guarantees. Figure 1 shows an example timeline for a timed-write operation for three processes $\mathtt{P}_1$, $\mathtt{P}_2$ and $\mathtt{P}_3$. The operation $\mathtt{tw}_1\mathtt{(x)1}$ occurring at time t_1, will be definitely made visible Δ time (i.e, at $t_1 + \Delta$) after its completion. Hence the read operation $\mathtt{r}_2\mathtt{(x)1}$, occurring after Δ time, returns the value written by $\mathtt{tw}_1\mathtt{(x)1}$, while $\mathtt{r}_3\mathtt{(x)0}$ which completes before $t_1 + \Delta$ does not return 1. More generally, this means that at any time

[1]Intuitively, it can be seen as an implicit tagging of each value by a pair composed of the identity of the process that issued the write plus a sequence number.

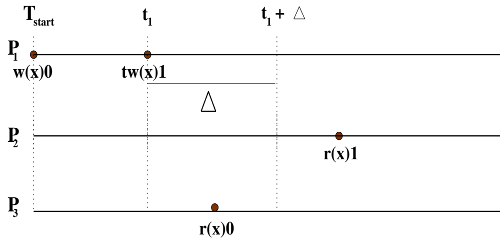

Figure 1: Example timeline for *timed-write* operation.

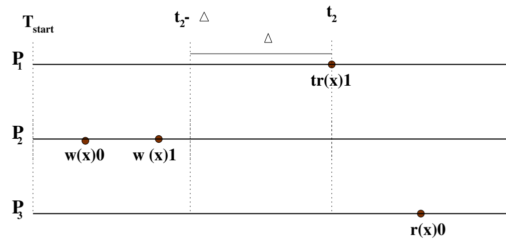

Figure 2: Example timeline for *timed-read* operation.

after $t_1 + \Delta$, no value of x older than the one at t_1 is accessible. Similarly, Fig. 2 is an example timeline for a timed-read. The timed-read $\texttt{tr}_1\texttt{(x)1}$ at time t_2 reads the value corresponding to the most recent write before t_2 -Δ (i.e., $\texttt{w}_2\texttt{(x)1}$), while this value is necessarily not available to the read operation $\texttt{r}_3\texttt{(x)0}$ which is non-timed.

3.3 Histories

Although operations occur at certain times, an order based on their completion time may be too strong and costly to implement. Because of these reasons, other ordering constraints have been considered in various consistency models. They are obtained by imposing some ordering constraints on the set of operations issued by the program. These constraints can either be independent of time or based on it and are discussed below.

3.3.1 Time Independent History

The *local history* (or local computation) $\widehat{h}_i$ of P_i is the sequence of operations issued by P_i. If op_1 and op_2 are issued by P_i and op_1 is issued first, then we say op_1 precedes op_2 in P_i's process-order, which is denoted as $op_1 \rightarrow_i op_2$. Let h_i denote the set of operations executed by P_i; then the local history $\widehat{h}_i$ is the total order $(h_i, \rightarrow_i)$.

A *time independent execution history* (or simply a history, or a computation) $\widehat{H}$ of a shared object system is a partial order $\widehat{H} = (H, \rightarrow_H)$ such that :

- $H = \bigcup_i h_i, i = 1 \dots n$
- $op_1 \rightarrow_H op_2$ if :

  i) $\exists\ P_i\ : op_1 \rightarrow_i op_2$ (in that case, $\rightarrow_H$ is called *process-order* relation),

  or *ii)* $op_1 = W_i(x)v$ and $op_2 = r_j(x)v$ (in that case $\rightarrow_H$ is called *read-from* relation),

  or *iii)* $\exists op_3\ : op_1 \rightarrow_H op_3$ and $op_3 \rightarrow_H op_2$.

Two operations op_1 and op_2 are *concurrent* in $\widehat{H}$ if neither $op_1 \rightarrow_H op_2$ nor $op_2 \rightarrow_H op_1$ is true (denoted as $op_1 || op_2$).

3.3.2 Time Dependent History

A *time dependent execution history* $\widehat{T}$ of a shared object system is a partial order $\widehat{T} = (H, \rightarrow_T)$ such that :

- $H = \bigcup_i h_i, i = 1 \dots n$

- $op_1 \rightarrow_T op_2$ if :

  i) $op_1 \rightarrow_H op_2$

  or *ii)* $T(op_2) - T(op_1) \geq \Delta$ and $op_1 = tw$ and $op_2 \in \{R, W\}$.

  or *iii)* $\exists op_3 : op_1 \rightarrow_T op_3$ and $op_3 \rightarrow_T op_2$.

Time dependent execution history forces any *op* occurring Δ units after a timed-write (*tw*) to be ordered after *tw*. Hence $\rightarrow_T$ enforces a time (based) order among the set of operations.

3.4 Legality Constraints

The legality concept is a key notion on which our definition of TS consistency is based. Legality can also be separated out into time-dependent and time-independent components.

3.4.1 Time Independent Legality Constraints

A read operation $R(x)v$ is *legal* in history $\widehat{S} = (H, \rightarrow_S)$,if : (i) $\exists\ W(x)v\ : W(x)v \rightarrow_S R(x)v$ and (ii) $\nexists\ op(x)u\ : (u \neq v) \wedge (W(x)v \rightarrow_S op(x)u \rightarrow_S R(x)v)$. A history $\widehat{H}$ is legal if all its read operations are legal.

3.4.2 Time Dependent Legality Constraints

A timed-read operation $tr(x)v$, in history $\widehat{S} = (H, \rightarrow_S)$ is time-legal if : (i) $\exists\ W(x)v\ : W(x)v \rightarrow_S tr(x)v$ and (ii) $\nexists\ op(x)u\ : (u \neq v) \wedge (T(tr) - T(op) \geq \Delta) \wedge (T(W) < T(op))$. A history $\widehat{H}$ is time-legal if all its timed-read operations are time-legal.

In a legal history no read operation can get an overwritten value and in a time-legal history, no timed-read (*tr*) operation will return a value older than the "*most recent*" value at *T(tr)* - Δ.

3.5 Definition of TS Consistency Model

Although timed write operations must become effective after Δ time, the value of a timed write can become available to processes at different times in the Δ interval. Similarly, non-timed writes can become effective at different times. Thus, it is not possible to create a single serial history comprising of all operations of the processes that is legal. Instead, we define a history for each process that includes its reads and all writes (this approach has been used to define several models [1]). Let $\widehat{T_i}$ be the sub-history of $\widehat{T}$ from which all read operations not issued by P_i have been removed[2]. The history $\widehat{T} = (H, \rightarrow_T)$ is *TS consistent*, if *for each process* P_i, $\widehat{T_i}$ *is both legal and time-legal.*

In Fig. 3, it is possible to obtain a history $\widehat{T_i} = (T, \rightarrow_T)$ for a process P_i consisting of the local operations and all the write operations as follows:

$\widehat{T_1} = \{tw_3(y)15 || w_2(y)10\}$,

$\widehat{T_2} = \{w_2(y)10, r_2(y)10 || tw_3(y)15, r_2(y)15\}$

$\widehat{T_3} = \{tw_3(y)15 || w_2(y)10\}$

These histories respect the time order($\rightarrow_T$) as well as the time legality conditions. Therefore the given execution sequence is *TS* consistent. The execution in Fig. 4 is not *TS* consistent for the following reasons. Consider the following partial order

[2] More formally, $\widehat{T_i}$ is the sub-relation of $\widehat{H}$ *induced* by the set of all the writes of H and all the reads issued by P_i.

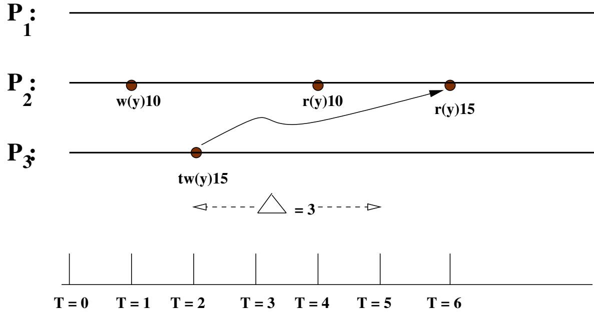

Figure 3: An example operation sequence that respects *time sensitive* consistency model. Assume a timeliness value (Δ) of 3 units and $T_{start} = 0$. The timed-write `P3:tw(y)15` gets ordered before operations happening after T = 5 (i.e, `P2:r(y)15`).

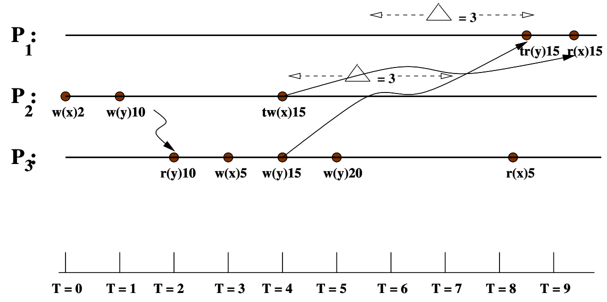

Figure 4: An execution trace that violates *TS* consistency (Δ=3, $T_{start} = 0$). `P1:tr(y)15` does not respect the time-legality constraint.

$\widehat{T_1} = \{w_2(x)2, w_2(y)10, (w_3(x)5, w_3(y)15)||tw_2(x)15, (tr_1(y)15, r_1(x)15)||w_3(y)20\}$, for process P_1 comprising of its local operations and all the writes. $\widehat{T_1}$ is not time legal because $\texttt{tr}_1\texttt{(y)15}$ does not return a value that is latest at the time 5 (corresponding to `w(y)20`). Thus timed legal history for P_1 cannot be constructed. A similar problem exists for $\widehat{T_3}$. Hence the execution is not *TS* consistent.

Other consistency models which specify different orderings can be obtained by setting different values for the timeliness threshold Δ, and restricting the type of operations allowed (say only timed-writes and any reads etc.). For a timeliness value of infinity, i.e, $\Delta = \infty$, there is no upper bound on the time-guarantees for timed-read and timed-write operations. Hence a timed-read can return a value that is arbitrarily stale, while a timed-write does not have to be perceived in bounded time. In other words, the semantics of timed-reads and timed-writes is not different from their non-timed counterparts. The system execution effectively reduces to a set of non-timed operations that are bounded by the time independent legality constraint (read legality). The time independent partial history $\widehat{T} = (H, \rightarrow_H)$, thus obtained will respect the read legal constraints, which is same as the definition of *causal consistency*. Therefore for a timeliness value of infinity, a *TS consistent* system reduces to a *causally consistent* system.

For a timeliness value of 0 ($\Delta = 0$), a timed-read is guaranteed to perceive the most-recent write that was completed before the read operation, while a timed-write with zero timeliness is visible immediately after its completion at all the processes. In a system in which all writes are timed, it is possible to obtain an ordering for the operations that is similar to real time ordering. Thus for a timeliness value of zero and for a system which permits only timed-writes, *TS* consistency reduces to *linearizability*.

4 Δ-Time Protocol for TS Consistency

We have developed a protocol that ensures *TS* consistency for multiple replicas of a shared object cached at different clients. Clients locally create replicas of an object instantiated at a server and can perform read and write operations with their copies. The protocol provides consistency by enforcing the required ordering and timeliness constraints on a group of related objects. Before describing the protocol, we discuss some of the assumptions made by us.

- The protocol assumes a client-server architecture. The server plays an important role in maintaining the consistency of the copies cached at the clients. It allows clients to acquire

object and control state. A client need not be aware of the other clients present in the system. We assume that a group of related objects is managed by a single server.

- We assume the presence of a synchronized global clock. Node clocks are assumed to run in synchrony with the global clock and are used to timestamp local events. A protocol like NTP [15] can be used to synchronize node clocks. Both the model and the protocol can be adapted to clocks that are approximately synchronized, but in the following discussion, we assume that the clocks are perfectly synchronized.
- A read or a write operation executes in the finite time period $[T_{begin}, T_{end}]$. T_{op}, the value returned by the function $T()$ for an operation *op* (the time of occurrence of the operation) will be a time in this interval. This is called the *effective time* of the operation.
- Although no assumptions are made about network latency in the model, the protocol is designed to perform well in situations where Δ is much greater than the network latency.

The protocol must ensure the following:

- A read operation at any process should never return a value that was overwritten in its history (causal guarantee).
- A timed-write on a variable, say `x`, should be made visible to all the client processes sharing `x` in Δ time (timed-write guarantee).
- A timed-read operation that completes at time T should not return a value that was overwritten at time $T - \Delta$.

4.1 Protocol Variables

The protocol assumes a system of m clients, $\mathcal{P} = P_1, P_2, \ldots, P_m$ that perform read and write operations on a related set of n objects, $\mathcal{X} = X_1, X_2, \ldots, X_n$. Server S coordinates the client accesses to shared objects in a manner consistent with the *TS* consistency model. The objects are represented as records. Each record has two fields, storing the value of the object (*val*) and the time at which that value was produced (T_w). Thus $x.val$ corresponds to the value of the object x, while $x.T_w$ is the effective time of the write operation *W(x)val*. This record is exchanged between the clients and the server as a part of consistency messages. An object cached at a client can either be in an `invalid` state or as a `read-only` or `writable` copy. The vector `state[]` at a client stores the current state of all the objects in its local cache. Each client stores the time of the most recent write to an object as known to it in the timestamp vector `TS[]`. A client also keeps the time at which $\mathtt{x}_i$ was locally made valid in $\mathtt{T}_{refresh}[x_i]$.

`S` maintains a record of the clients sharing `x` (i.e., readers) in `readerSet(x)`, while the identity of the clients that can write a new value to `x` (i.e., writers) is stored in `writerSet(x)`. The time stamp vector `TS[]` at `S` has the recent write times for the objects as known to the server. We use a remote procedure call (RPC) notation for the communication between different processes. These functions are invoked by processes interested in propagating consistency actions. For example, server `S` can initiate consistency actions at a client, say $\mathtt{P}_k$, by invoking the appropriate remote function `f()` at $\mathtt{P}_k$ and this is shown as $\mathtt{P}_k$`.f()` in the protocol.

ACTIONS AT CLIENT P_k :

```
init()
 //initializing client meta data
  ∀j ∈ [1..n], set
     state[j] := invalid
     T_refresh[j] := 0 //refresh time
     TS[j] := 0 //timestamp

r(x_i)
 //read
  if(state[i] = invalid) then
     readmiss(i,nonTimed)
  return(x_i.val)

tr(x_i)
 //timed-read
  if((state[i] = invalid )OR
   (T_current − T_refresh[i] > Δ)) then
     readmiss(i, timed)
  return(x_i.val)

w(x_i,val)
 //write
  if(state[i] ≠ writable) then
     writemiss(i)
  x_i.val := val
  x_i.T_w := T_current
  TS[i] := T_current

tw(x_i,val)
 //timed-write
  if(state[i] ≠ writable) then
     writemiss(i)
  x_i.val := val
  x_i.T_w := T_current
  TS[i] := T_current
  S.update(x_i, TS)

readmiss(i, isTimed)
 //x_i invalid or staler than Δ
  <x_i,TS_S,T_svr> :=
    S.readableCopy(i,isTimed,P_k)
  T_refresh[i] := T_svr
  invalObjs(TS_S)
  TS[i] := max(TS[i],x_i.T_w)
  state[i] := read-only

writemiss(i)
 //x_i not writable
  S.writableCopy(i, P_k)
  state[i] = writable

update(i,TS_S[1..n])
 //timed-write intimation
  invalObjs(TS_S)
  state[i] := invalid

invalObjs(TS_S[1..n])
  ∀ ∈ [1..n], if(TS_S[j] > TS[j]) then
     state[j] := invalid
     TS[j] := TS_S[j]

getCopy(x_S_i)
 //S requests a new copy
  if (x_S_i.T_w < x_i.T_w) then
     return <true, x_i, TS>
  else return <false,null ,null>
```

ACTIONS AT SERVER S :

```
init()
 //initialize server meta data
  ∀j ∈ [1..n], set
     writerSet[j] := empty //writer list
     readerSet[j] := empty //reader list
     x_j.val := 0
      //recent timestamps as known to S
     TS[j] := 0

readableCopy(i, isTimed, P_k)
 //request for shared copy
  readerSet[i].add(P_k)
  if(isTimed OR(x_i.T_w < TS[i])) then
     ∀ P_owner ∈ writerSet[i]
        <isNew, x_temp,TS_temp> :=
           P_owner.getCopy(x_i)
        if(isNew = true) then
          x_i := x_temp
          TS_c := TS_temp
          ∀j ∈ [1..n],
           TS[j] := max(TS_c[j], TS[j])
  return <x_i, TS, T_current>

writableCopy(i, P_k)
 //request for a writable copy
  readerSet[i].add(P_k)
  writerSet[i].add(P_k)

update(x_c_i, TS_c[1..n])
 //timed-write update
  x_i = x_c_i
  ∀j ∈ [1..n],
     TS[j] := max(TS[j],TS_c[j])
  ∀ P_reader ∈ readerSet[i]
     P_reader.update(i, TS)
     readerSet[i].remove(P_reader)
     writerSet[i].remove(P_reader)
```

Figure 5: Δ-Time Protocol for m clients sharing n related objects. It shows consistency actions happening at a client P_k and the Server S for the ith object x_i in a group $\mathcal{X}$ = $\{x_1, x_2, \ldots, x_n\}$ of n related objects.

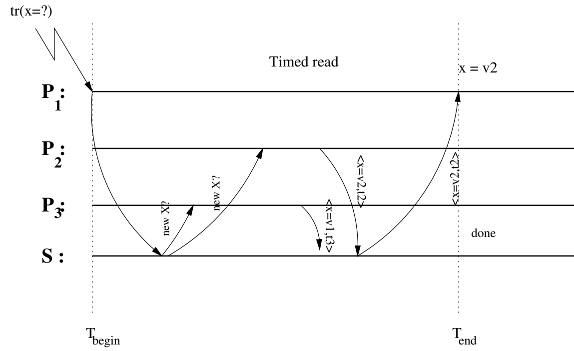

Figure 6: An example time line showing a timed-read in action. The distributed system has three clients P_1, P_2, P_3 and a server S. Let `owners(x)=` {`P2, P3`} and the timestamp t_2 from P_2 is greater than the timestamp t_3 from P_3. The timed read completes at a time $T()$ such that $T_{start} < T() < T_{end}$.

4.2 The Protocol

We first present a simple conservative protocol and then address many of the optimizations. This protocol is shown in Fig. 5. It shows the implementation of `r,w,tr` and `tw` operations at the client process $\texttt{P}_k$ and the consistency calls made between $\texttt{P}_k$ and the server `S`, for object $\texttt{x}_i$. For ease of understanding, we will describe the operations for a single object, say `x`. The object is initially in an `invalid` state at all the clients. `S` initializes `x` and its write timestamp to 0 at the start. It also sets `readerSet` and `writerSet` to be empty.

The messages exchanged during the execution of a timed read operation are shown in Fig. 6. During a *timed-read* operation on object `x`, a client process $\texttt{P}_k$ verifies if the current cached copy is in a `valid` state and if the current time ($\texttt{T}_{current}$) does not exceed the time when the copy was locally refreshed by more than Δ units. If so, the cached copy is good and the timed-read completes locally. Otherwise, the client contacts `S` for a more recent copy, if one exists (`S.readableCopy`). `S` polls all the writers of `x` ($\texttt{P}_{writer}$`.getCopy`) and chooses the copy with the most recent timestamp. It then updates its local value of `x` and sends it to $\texttt{P}_k$. $\texttt{P}_k$ marks the local copy to be in valid state and updates the refresh time. Similarly a `read` operation checks if the local copy is valid, if so, then the value of the copy is returned, otherwise the `readmiss` function contacts `S`. If the local copy at `S` is valid, then it will be returned, even though the copy may be old. The server contacts the writers only when its copy is in an `invalid` state.

When a process $\texttt{P}_k$ executes a timed-write or a write operation on `x`, it checks to see if it has a writable copy of `x`. If not, the `writemiss` function is called, which contacts `S` for a writable copy of `x` (`S.writableCopy`). `S` adds $\texttt{P}_k$ to the `readerSet` as well as the `writerSet` of `x` and allows $\texttt{P}_k$ to make modifications to `x`. On completing the timed-write, $\texttt{P}_k$ contacts `S` propagting the new value of `x` (`S.update`). `S` updates its local copy and sends invalidation messages to all the readers ($\texttt{P}_{reader}$`.update`). The readers invalidate their local copy. On receiving acknowledgements for the invalidation messages from the readers, S removes them from the `writerSet` and the `readerSet` of x. This pruning of readers and writers will eliminate any unwanted messages between the server and clients for a consistency action in the future. `S` then responds back to $\texttt{P}_k$, allowing the timed-write operation to proceed to completion. Figure 7 shows a timed-write in action. A write operation is similar to a timed-write. However, the operation only updates the local state of the object and the writer does not propagate this new value to `S`.

The protocol manages a group of objects instead of a single object as explained above. Whenever a consistency call is made between a client and the server, `TS` vector, having the most recent write time as known to that client, is also exchanged. This vector is compared

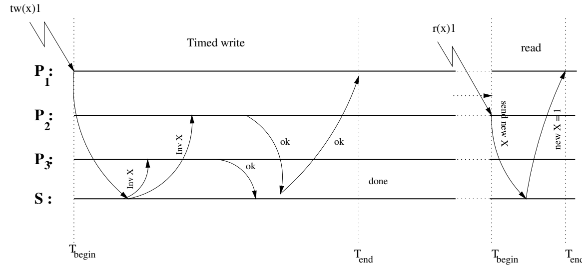

Figure 7: An example time line showing a timed-write in action. The distributed system has three clients P_1, P_2, P_3 and a server S. Also `readers(x)=` $\{$`P`$_2$`, P`$_3\}$. The S server sends messages to the readers requesting them to invalidate their copies.

with the local timestamp vector and all the objects with local write timestamps less than the timestamps received in the consistency message are invalidated. This ensures that a client will not read causally overwritten value even when there are no timeliness requirements.

4.3 Optimizations

The protocol for timed-read and timed-write operations as discussed above would be expensive to implement for the following reasons. The server asks readers to invalidate their copies when it receives a request for a timed-write operation. However, according to the definition of a timed-write operation, readers can continue to read the older values until after Δ time has passed since the completion of the write. This results in potentially unnecessary consistency messages exchanged between the server and the readers. Also, as is seen from Fig. 7, the server sends invalidation messages to the readers for every timed-write. For timed-writes occurring in a Δ time interval, multiple invalidation messages can be exchanged between the server and the readers, if `readerSet` gets updated between two timed-writes. Another shortcoming of the protocol is that whenever a writer requests a writable copy, it gets added to the writer list for that object and is contacted for every subsequent timed-read, even when it has stopped making updates to the object. Similarly, for a timed-read operation, the server does not need to poll the writers when its copy of the object is no older than Δ time before the read. These observations motivate the following optimizations to the protocol.

- For a timed-write operation, the server will send invalidation messages to the readers to request them to invalidate their copies after Δ time. This allows them to continue to read the cached value until it is no longer consistent.

- The writers are leased a copy of an object when a writable copy is requested, rather than a indefinite ownership. This way the server can periodically prune the writer list for an object, effectively reducing the number of consistency messages generated during a timed-read operation. Also during a timed-write, the server can reset the lease for a writer to be valid for the next Δ time units. Assuming Δ to be smaller than the lease time, this can effectively reduce the number of messages generated for a consistency action due to a timed read.

- For a timed-read, the server will only poll the writers if the timestamp of its local copy indicates that it is older than the timeliness threshold. If not, it will return its copy. For a normal read operation that has faulted, server can return its own copy as long as the write time ($\mathtt{T}_w$) is not less than time stamp entry in `TS` for that object (causal ordering).

A complete protocol that incorporates these optimizations is presented in Appendix A. The performance results discussed in the next section are based on an implementation of this optimized protocol.

5 Performance Measurements

5.1 Implementation of Δ-Time protocol

The Δ-Time protocol for TS consistency has been implemented in a distributed object caching framework developed by us. The framework can transparently cache objects at clients that invoke them. It supports different notions of quality of service (QoS) for shared state. For example, it is possible to set the timeliness threshold to different values in the Δ-time protocol for different clients. The *Quality Objects*(QuO)[19] framework provides an interface which can be used by applications to specify and adapt to different QoS guarantees offered by the underlying system. In our earlier work, we have used QuO to define shared state QoS interfaces to our caching framework [11]. Our current caching prototype has been developed in Java. It provides the same interface to the applications as *Java RMI* (Remote Method Invocation).

The caching framework has provisions for caching objects individually or as a group. Consistency protocols used to govern the cached objects can be specified at runtime via a high level shared state QoS property. The protocols are implemented as client side and server side consistency objects. For example, the actions at client $\mathtt{P}_k$ for the protocol described in Appendix A will be implemented as a client side consistency object and the actions at `S` will be implemented as the server side consistency object. The framework also allows caching and non-caching clients to co-exist. The other components of the framework include smart delegates which forward invocations to the locally cached objects or remote clients based on QoS requirements, meta-state regarding the read and write access information for member functions of objects, infrastructure to ship the byte code corresponding to the object definitions from the server in case the definitions are not locally found at the clients, and a transport object that provides communication channels between server and clients for the dissemination of consistency actions. Additional details of the framework can be found in [11] and [12].

5.2 Evaluation

The goal of this section is to experimentally evaluate the performance of the Δ-time protocol in our caching system to quantify the benefits of a *TS* consistent object system. We evaluated the optimized version of the protocol which is described in Appendix A. We measure the response time for invoking a shared object. We first measure the response times of a local invocation and a remote invocation executed at the server. To determine the cost of invocation for the Δ-time consistency model, we use a synthetic workload derived from attributes of an interactive distributed application.

The experiments were conducted on a cluster of 248 MHz Sparc Ultra-30's connected by a 100 Mb Ethernet. The machines were all equipped with 128 MBytes of memory. The Java virtual machine used was Java2 from Javasoft and we used it with the just-in-time (JIT) compilation option enabled. There were no other applications running on the machines when the experiments were conducted and hence the numbers generated were repetitive. We ran each of the experiments three times and the numbers presented here are averaged across multiple runs and over multiple clients. It was difficult to generate numbers that were repetitive in a wide-area environment. This was primarily due to our lack of control on the network. Because of this reason, we are only presenting the measurements for the local-area environment in this

paper. In the future, we plan to repeat these with widely distributed sites connected by the Internet, possibly using an Internet emulation test bed.

In our experiments clients invoked an object X implemented by the server. The definition of X has four member functions: *read()*, *write()*, *timed-read()*, and *timed-write()*. The read and the timed-read method have a null body while the write and the timed-write methods increment the state of a shared counter. Since little time is spent in the execution of the methods, the average invocation time obtained in the experiments is a direct measure of the communication and computation costs associated with the protocol.

We used synthetically generated workloads based on important parameters of interactive applications to evaluate our system. We briefly describe some of the parameters that were considered in generating the workloads.

Number of Objects: There are n related objects $X_1, X_2, \ldots, X_n$. They are all instantiated at the same server. In our experiment we used a value of 64 for n. Also the size of all the objects was 64 bytes.

Number of Clients: There are m clients $\mathcal{P} = P_1, P_2, \ldots, P_m$ that can make invocations on the objects. We assigned a value of 16 to m.

Number of Invocations per Client: Each client makes k invocations. k was chosen to be 50,000.

Ownership: The owners can make modifications to the objects. An object can be owned by multiple clients at the same time. The owner set for an object ($\mathcal{O}$) is a randomly generated and is a subset of the total client set ($\mathcal{O} \subset \mathcal{P}$).

Read Frequency: Assuming that interactive applications are visual and require frequent screen updates, we generated read requests to a random set of objects once in every 30 milliseconds.

Write Frequency: The writes in these applications may be because of user actions or because of movement of autonomous entities (e.g., movement of an entity in a predetermined trajectory). We also assumed that a user does not recognize events happening in a time period less than 100ms. So the lower limit for the time between writes is 100ms (for autonomous entity movement) and the higher limit was fixed at 3 seconds (for user actions). The writes were generated at random in the [.1, 3] second range.

Timed/Non-timed operation ratio : The percentage of timed-operations was varied between 0 (none), 10, and 100 (all), to study the impact of timed operations on the invocation response time.

We synthetically generated five different workloads (*tr0-tw0, tr-10, tr-100, tw10* and *tw-100*) based on the above parameters. Each workload has 8 traces of 50,000 invocations that were used to drive the clients (8 of them). *tr0-tw0* was generated for non timed read and write operations. It was used as the base and all the other workloads were generated from it. For example, *tr-10* was generated by randomly choosing read operations from *tr0-tw0* and changing them to timed-reads such that the timed to non-timed read ratio was 1/9. Similarly, *tr-100* has 100% timed-reads, while *tw-10* and *tw-100* have 10% and 100% timed-writes respectively.

Table 1 shows the cost of making an invocation locally on a cached copy with no consistency actions, and at a remote server with 8 active clients. It takes 25 microseconds for an invocation on an object stored locally by the caching framework, while it takes about 11.01 milliseconds[3] for a remote call to the server to complete in the presence of 8 active clients. These numbers do not depend on the type of workload used. We want to treat these as the boundary cases and

[3] An RMI invocation at a remote server by a single client can be executed in 1.24 milliseconds. However, when the server is concurrently invoked by 8 clients, this time increases to 11.2 milliseconds

Invocation Execution	Invocation Time in milliseconds
At locally cached copy	0.025
At remote server	11.013

Table 1: Comparison of the time per invocation in milliseconds averaged over 50,000 invocations. The invocation times for locally cached copies and invocations at the server with 8 active clients. The size of the object was 64 bytes and a group of 64 related objects were used.

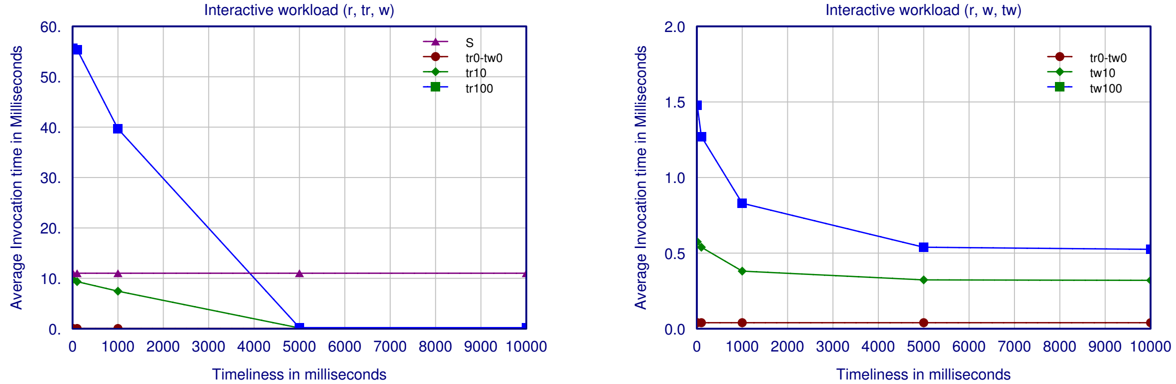

Figure 8: The invocation time averaged over 50000 invocations with 8 clients and 64 objects for Δ-time protocol. The first graph was generated from workloads with no timed operations, 10% timed-reads and 100% timed-reads respectively, while the second one was for non timed, 10% and 100% timed-writes.

would like to determine the values of timeliness threshold for which the invocation latencies for Δ-time protocol will be between the local invocation and invocation at the server.

The average invocation time for Δ-time protocol for different values of timeliness threshold is shown in in Fig. 8. The first graph shows different plots corresponding to invocation at server (S), workloads with no timed operations (*tr0-tw0*), with 10 (*tr-10*) and 100 (*tr-100*) percent timed reads respectively. The plot for *tr0-tw0* is a straight line as the invocation latency for non-timed operations is independent of timeliness. The invocation time (40 microseconds) is higher than the local invocation time (25 microseconds) because of the additional time spent in executing consistency actions that maintain causality. The plot corresponding to *tr-10* shows that the average invocation latency decreases for higher timeliness thresholds. This is because for lower values of Δ, the locally cached copies expire faster, hence the client has to contact the server and if need arises, the writer to fetch a new copy for faulted invocations. This leads to higher values of average invocation times. As the timeliness threshold is increased, more and more invocations are executed locally, resulting in fewer consistency messages, and hence average invocation time reduces considerably.

A similar trend can be observed for *tr-100*, but since every read in this trace is timed, lot more consistency messages are generated in comparison with *tr-10*, accounting for high invocation times. For *tr-10*, even for a low timeliness value of 10 milliseconds, the average invocation time when caching is enabled is better than the remote invocation at S. *tr-100* performs better than S for Δ values greater than 4 seconds. Similar shaped curves are obtained for *tw-10* and *tw-100* in the second graph. However, the average invocation times are smaller than for *tr-10* and *tr-100*. This is because the write frequencies in *tr0-tw0* are much smaller than the reads (only about 15% of operations are writes) and as a result far fewer consistency actions are required, improving the invocation latencies.

6 Conclusions

In this paper we have presented the *TS* consistency model which provides both order and time based guarantees for copies of shared objects. In addition to the normal read and write, it defines two new operations, namely *timed-read* and *timed-write* , which can be used by distributed applications for time-based access to dynamically changing shared state. Distributed interactive applications benefit from the *TS* consistency model by smartly exploiting its timeliness property. We also presented the Δ-time protocol for *TS* consistency. A conservative and an improved version of the protocol were discussed. The optimized protocol has been implemented in a QoS enabled caching framework that is being developed by us. The Δ-time protocol provides better response time for invocations, almost as small as an invocation on a local object, for a timeliness threshold value greater than 5 seconds, even for a workload comprising of very high percentage of timed operations. A more realistic scenario would have fewer number of timed operations and hence it is possible to obtain even better invocation times for much smaller Δvalues.

In the future, we would like to develop a more comprehensive synthetic workload for interactive applications. Actual experiments and simulations driven by the workload will allow us to perform a detailed evaluation of the protocol implementations. We are also interested in developing applications with which we can drive the framework to determine the effectiveness of the system.

References

[1] M. Ahamad, G. Neiger, J. E. Burns, P. W. Hutto, and P. Kohli. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9:37–49, 1995.

[2] B. N. Bershad and M. J. Zekauskas. Midway - shared memory parallel programming with entry consistency for distributed memory multiprocessors. In *Technical Report CMU-CS-91-170, Carnegie Mellon University*, September 1991.

[3] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proc. of ACM SIGMETRICS*, 1994.

[4] M. Dubois and C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[5] M. Ahamad, F. Torres-Rojas, and M. Raynal. Timed consistency for shared distributed objects. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 163-172, 1999.

[6] R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[7] J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *Proc. of the 1996 USENIX Technical Conference, Jan 1996*, 1996.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, July 1990.

[9] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium of Computer Architecture (ISCA)*, pages 13–21, May 1992.

[10] J. Kistler and M. Satyanarayanan. Disconnected operation in Coda file system. In *ACM Symposium on Operating systems and Principles*, 1992.

[11] V. Krishnaswamy, I. Ganev, J. M. Dharap, and M. Ahamad. Distributed object implementation for interactive applications. In *Proceedings of Middleware 2000*, April 2000.

[12] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementation of Java remote method invocation (RMI). In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, April 1998.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer Systems*, c-28(9):690–691, 1979.

[14] C. Liu and P. Cao. Maintaining strong consistency in the world-wide web. In *Proc. of the International Conference on Distributed Computing Systems*, 1997.

[15] D. L. Mills. *RFC1305: Network time protocol (version 3)*, 1992. http://www.cis.ohio-state.edu/htbin/rfc/rfc1305.html.

[16] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 1(8):142–153, Jan 1986.

[17] M. Raynal and A. Schiper. From causal consistency to sequential consistency in shared memory systems. In *Proceedings of 15th Int. Conf. FST&TCS*, Springer Verlag LNCS 1026, pp. 180-194. 1995.

[18] D. B. Terry, M. M. Theimer, K. Peterson, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM SOSP*, 1995.

[19] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. In *Theory and Practice of Object Systems*, 1997.

A Optimized Protocol

We discussed some of the optimizations permitted by *TS* consistency model which could be used to improve the performance of a protocol implementing the model in Sect. 4.3. An improved version of the Δ-time protocol is presented in Fig. 9. We briefly describe the new variables that have been introduced in the new protocol to implement these optimizations.

A.1 Protocol Extensions

$\mathtt{T}_{valid}$ is a timestamp vector on the client side which is used to track the validity of an object until a time in the future. Thus, during a timed-write, the server can set $\mathtt{T}_{valid}$ such that all objects that have operations on them causally preceding the timed-write will expire in Δ time units, rather than invalidating them immediately as in the earlier protocol. Also, the server leases the objects to the clients for a certain period of time. The lease period of each object is stored in $\mathtt{T}_{lease}$ vector and is used to determine if the client has a valid lease for all write and timed-write operations. $\mathtt{T}_{refresh}$ at `S` stores the time an object was locally refreshed. Server can use this to determine whether its local copy could serve a timed-read request. `writerSet` data structure was used in the original protocol to store the identities of the writers for an object. In the improved protocol, it stores a tuple containing the identity of the writer client as well as its lease time for a particular object (<`writer-id, leasetime`>). *S* examines this record before issuing a consistency message to the writer. If a writer's lease for an object expires then `S` removes it from the `writerSet` it maintains for that object. This pruning of the writers will reduce the number of messages needed to perform consistency actions for a timed-read request in the future.

Figure 10 shows a sequence of operations on objects `x` and `y` at three clients P_1, P_2, P_3 and *S* as permitted by the new Δ-time protocol. The clients perform both timed and non-timed operations on `x` and `y`. The square bracket on top of an operation marks the start and the end times of that operation. The lease time for the writers is assumed to be 12 units while

ACTIONS AT CLIENT P_k :

```
init()
  ∀j ∈ [1..n], set
    T_valid[j] := 0
    T_refresh[j] := 0
    T_lease[j] := 0
    TS[j] := 0

r(x_i)
  if(T_valid[i] < T_current) then
    readmiss(i,nonTimed)
  return(x_i.val)

tr(x_i)
  if((T_valid[i] < T_current)OR
    (T_current − T_refresh[i] > Δ)) then
    readmiss(i, timed)
  return(x_i.val)

w(x_i,val)
  if((T_valid[i] < T_current)OR
    (T_lease[i] < T_current)) then
    writemiss(i)
  x_i.val := val
  x_i.T_w := T_current
  TS[i] := T_current

tw(x_i,val)
  if((T_valid[i] < T_current)OR
    (T_lease[i] < T_current)) then
    writemiss(i)
  x_i.val := val
  x_i.T_w := T_current
  TS[i] := T_current
  S.update(x_i, TS, P_k)

readmiss(i, isTimed)
  <x_i,TS_S,T_svr> :=
    S.readableCopy(i,isTimed,P_k)
  T_refresh[i] := T_svr
  invalObjs(TS_S, 0)
  TS[i] := x_i.T_w
  T_valid[i] := ∞

writemiss(i)
  T_lease[i] := S.writableCopy(i, P_k)
  T_valid[i] := ∞

update(i,TS_S[1..n])
  invalObjs(TS_S, Δ)
  T_valid[i] := T_current + Δ

invalObjs(TS_S[1..n], invTime)
  ∀j ∈ [1..n], if(TS_S[j] > TS[j]) then
    T_valid[j] := T_current + invTime
    TS[j] := TS_S[j]

getCopy(x_Si)
  if (x_Si.T_w < x_i.T_w) then
    return <x_i, TS>
```

ACTIONS AT SERVER S :

```
init()
  ∀j ∈ [1..n], set
    TS[j] := 0
    T_refresh[j] := 0
    x_j.val := 0
    readerSet[j] := empty
    writerSet[j] := empty
    ∀k ∈ [1..m]
      leaseTime[j][k] := leaseTime_jk

readableCopy(i, isTimed, P_k)
  readerSet[i].add(P_k)
  if((isTimed AND T_current − T_refresh[i] > Δ)
   OR (x_i.T_w < TS[i])) then
    ∀ <P_owner,T_lease> in writerSet[i])
      <x_i,TS_c> := P_owner.getCopy(x_i)
      T_refresh[i] := T_current
      ∀j ∈ [1..n],
        TS[j] := max(TS_c[j], TS[j])
      if(T_lease < T_current) then
        writerSet[i].remove(P_owner)
  return <x_i, TS, T_refresh[i]>

writableCopy(i, P_k)
  readerSet[i].add(P_k)
  T_lease := leaseTime[i][k] + T_current
  writerSet[i].add(P_k, T_lease)
  return T_lease

update(x_ci, TS_c[1..n], P_k)
  x_i := x_ci
  T_refresh[i] := T_current
  ∀j ∈ [1..n],
    TS[j] := max(TS_c[j], TS[j])
  ∀ P_reader ∈ readerSet[i]
    P_reader.update(i, TS)
    readerSet[i].remove(P_reader)
    T_lease := T_current + Δ
    writerSet[i].set(P_reader, T_lease)
```

Figure 9: Optimized Δ-Time Protocol for m clients sharing n objects. It shows consistency actions happening at a client P_k and the Server S for the ith object x_i in a group $\mathcal{X}$ = {x_1, x_2, ..., x_n} of n related objects.

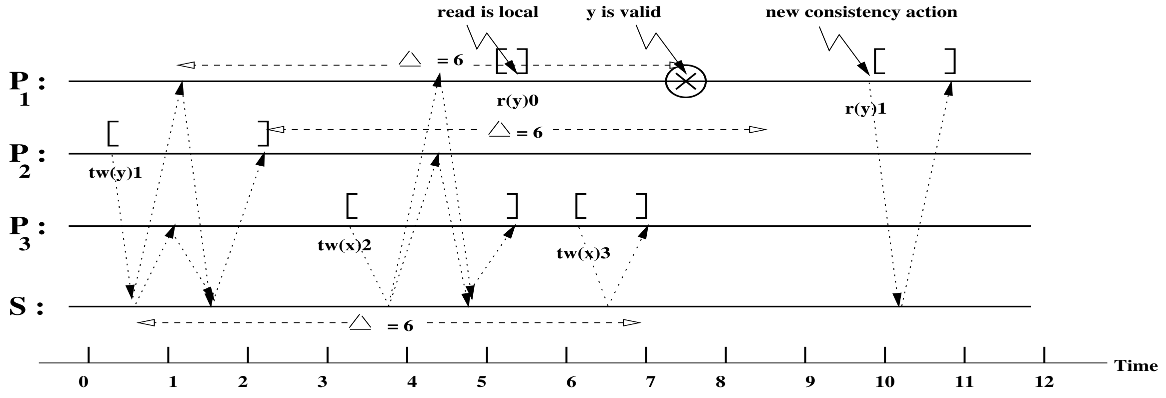

Figure 10: A set of operations that obey *TS* consistency. The dotted lines represent the consistency messages exchanged across the processes. The square brackets ([ ]) represent the duration of the operation. A network latency of .5 time units and a Δ of 6 units is assumed. As seen, the consistency message for the timed-write $\texttt{tw}_2\texttt{(x)1}$ marks the copies at P_1 and P_3 to be invalidated at a future time. Also S does not contact the readers for the timed-write $\texttt{tw}_3\texttt{(x)3}$ as the previous timed-write $\texttt{tw}_3\texttt{(x)2}$ is from the same process (P_3) and the causal history has not changed in between these two operations.

the timeliness threshold is 6 units. Also for the ease of explanation of the protocol, we assume a higher bound on the network latency to be 0.5 time units and the computation time for a consistency message at a node to be negligible. This example demonstrates some of the protocol improvements for timed-write operations. The processes P_2 and P_3 have leased `x` and `y` and hence can make modifications to them, while P_1 can only read the state of the objects. During the execution of the timed-write $\texttt{tw}_2\texttt{(y)1}$, P_2 contacts S, which in turn contacts the readers (P_1, P_3) asking them to invalidate their copies after $\Delta = 6$ time units, rather then requesting for immediate invalidation. So operations on `y` at P_1 before this expiration period can be executed with the local copy. For example, as seen in the figure, P_1 invalidates its local copy of `y` after time 7 and hence the operation $\texttt{r}_1\texttt{(y)0}$ occurring before it executes with the local copy returning 0, while $\texttt{r}_1\texttt{(y)1}$ executing after the invalidation results in consistency a message to S. Similarly, when P_3 contacts S for the timed-write $\texttt{tw}_3\texttt{(x)3}$, S does not communicate with the readers because this timed-write is from the same client (P_3) as the previous one ($\texttt{tw}_3\texttt{(x)2}$) and also there is no change to the causal history between the two writes.

Figure 11 shows a snapshot of operations executing on object `x` at five clients P_1, P_2, P_3, P_4, P_5 and S. The lease time for the writers is assumed to be 12 units while the timeliness is set at 3 time units. This example demonstrates the improvements for timed-read operations. The processes P_4 and P_5 have leased `x` and hence can make modifications to it, while P_1, P_2 and P_3 can only read the state of `x`. P_4's lease expires at around 14 time units as shown in the figure, while P_5 gets a new lease on `x` when $\texttt{w}_5\texttt{(x)1}$ is executed. During the execution of $\texttt{tr}_2\texttt{(x)1}$, P_2 contacts the server for a recent copy. S checks the refresh timestamp of its local copy of `x`. Since it does not exceed the current time by more than the timeliness threshold, it returns this copy to P_1 rather than polling the writers for a newer version of `x`. This effectively reduces the number of consistency messages exchanged between the clients and the server for a timed-read operation. During the timed-read $\texttt{tr}_1\texttt{(x)3}$, S polls the writer P_4, even though its lease has already expired. S has to do this because it has to verify if P_4 has the most recent copy of `x` before it can remove it from the `writerSet`. Once it verifies that, P_4 is removed from the `writerSet` and S will not contact it during subsequent read faults. S cannot send its local copy of `x` to P_2 for the timed-read $\texttt{tr}_2\texttt{(x)4}$ because the copy is more than Δ time units old and hence may not be suitable for the timed-read. But since S has pruned its `writerSet` earlier, it only has to communicate with P_5 for a recent copy of `x`.

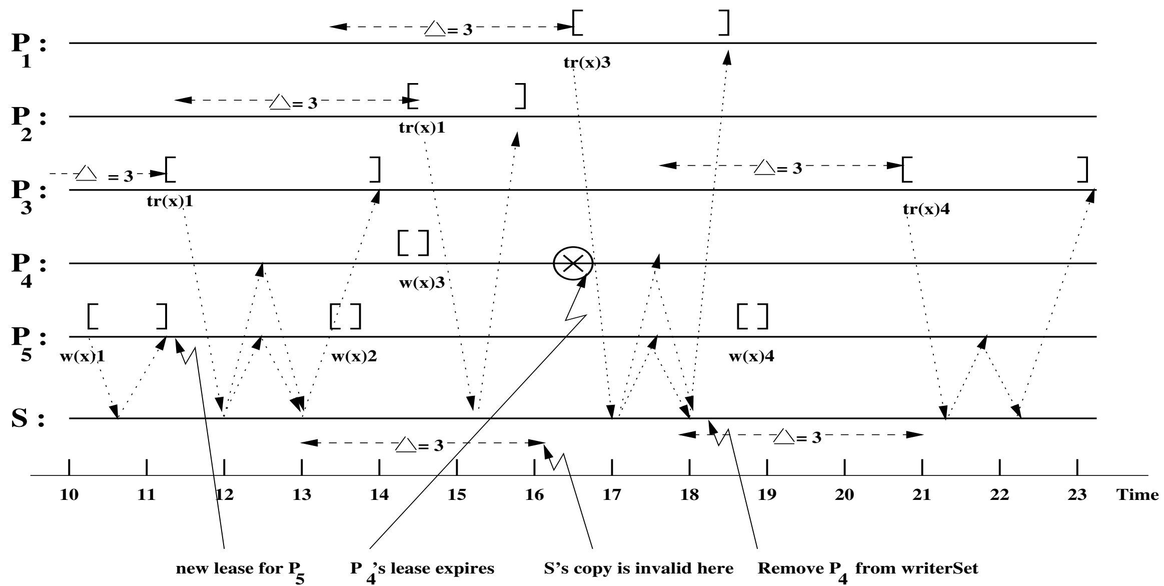

Figure 11: P_4 and P_5 are the writers. P_4's lease expires at the marker, while P_5 renews its lease during the execution of `w`$_4$`(x)1` and is good for the entire time. S returns its copy of `x` to P_2 during `tr`$_2$`(x)1` as the current time does not exceed the local refresh time by more than Δ time units. Also during the operation `tr`$_3$`(x)4`, S does not contact P_4 as its lease on `x` has already expired.