

# $\frac{1}{k}$ Phase Timestamping for Replica Consistency in Interactive Collaborative Applications

Sumeer Bhola\*      Mustaque Ahmad  
{sumberb, mustaq}@cc.gatech.edu

## Abstract

Interactive Collaborative applications, such as distributed virtual environments, collaborative CAD and multi-player games, are enabled by the data being shared across distributed users. The need to support such applications in a wide-area environment, and at the same time provide fast response to users, motivates replication of this shared data. The key characteristics of this shared data include short, incremental atomic operations for modifying this data, and the application's ability to predeclare access information for these operations. This and certain other differences from conventional replicated databases, motivate a different architecture for algorithms maintaining replica consistency. The main architectural differences include a complete separation of timestamping (for consistently ordering the operations) and execution of operations, propagation of the operation itself (instead of its effect) and local commit. We describe a specific consistency algorithm which does 2 phase timestamping (analogous to 2 phase locking) by sequentially acquiring stamps for objects accessed by an operation. Then we show how to eliminate the shrink phase and reduce the grow phase by a factor of  $k$  (when an operation accesses  $k$  objects), by concurrently getting all the stamps. This new algorithm is deadlock free and does not abort any operations.

## 1 Introduction

Interactive collaborative applications or groupware<sup>1</sup> refers to any application that allows a potentially distributed group of users to collaborate or compete on a task at the same time. Therefore, it encompasses traditional groupware like shared whiteboards and text editors, and more complex applications like multi-player games, distributed virtual environments, and collaborative CAD tools.

The interaction between users is enabled by the shared data. This shared data is highly dynamic in content and size. For example, in a distributed virtual environment, the entities in the environment are the shared data. Entity state is updated, for example when they move, entities can be added and deleted, for example the firing of a missile causes a missile entity to be added and when it hits the target it is deleted, and entities can interact, for example collision between entities, which results in atomic updates across multiple entities. Similar characteristics of the shared data are observed in other applications too, for example in the collaborative design of an articulated model or the key-frame for an animation [4].

In this paper we represent the shared data as *sets of objects* (o-sets). Each o-set is a collection of related shared data objects. Users/sites in a collaboration can dynamically *connect* or *disconnect* from an o-set and objects can be *added* to and *deleted* from the o-set. The sites that are currently connected to the o-set are referred to as *participants* in the o-set. Objects are modified using *updates* that can read and modify a subset of objects in an o-set. The operations on the o-set, i.e. add, delete, update and read-only, are the units of *atomicity*. The key characteristics that make the management of this shared data different from most conventional databases are:

- *Short, incremental operations* : The operations (add, delete, update, read-only) on an o-set have very short execution times. One reason for this is that users usually want to quickly see the impact of another user's

---

\*Contact author: address – College of Computing, 801 Atlantic Drive, Georgia Institute of Technology, Atlanta, GA 30332-0280, USA. email – [sumberb@cc.gatech.edu](mailto:sumberb@cc.gatech.edu), phone – (404) 894-6169, fax – (404) 894-9442

<sup>1</sup>We prefer to use the term Interactive, instead of Synchronous or Real-time which are commonly used in the research literature. Also groupware refers to Interactive groupware in this paper, unless explicitly qualified.

operations (as they are interacting at the same time). This is in sharp contrast to asynchronous groupware, where the model is of long-lived transactions with some cooperation between such transactions [2, 1]. A second reason is the common usage of direct manipulation interfaces. Such interfaces utilize input devices such as mice, joysticks and body-position sensors, which result in fine-grained and incremental actions by the users. Each such action corresponds to an operation on the o-set.

- *No Explicit Aborts* : One consequence of the incrementality of operations, and the usual way that users interact with such applications, is that operations are not explicitly aborted by the users. However, a user may want to undo a previous operation, which is typically done by executing an undo operation. We will treat such undo operations as regular operations.
- *Predeclared Access Information* : The objects that an operation will access, and the nature of that access can usually be predicted very accurately at the time when an operation is issued. This implies that an operation does not need to be executed in order to get its access information, which will have a strong influence on our algorithm design.
- *Commutativity of Operations* : In many collaboration scenarios, concurrently issued operations by multiple users are commutative. For example, many geometric transformations on the same object commute. Therefore, we should be able to utilize more than just read, write access information per object. We will represent such information as an access compatibility matrix (chapter 2 of [3]).

One of the critical requirements of a system managing the o-set is to provide appropriate *responsiveness*. By responsiveness we mean both the time taken to see the response of an operation on the o-set by the user issuing that operation, response time or RT , and the impact of that operation (only for non read-only operations) seen by a different user, user to user time or UUT . Providing appropriate RT and UUT is difficult when groupware is deployed in a wide-area distributed environment like the Internet, where high communication latencies are common. Direct manipulation user interfaces can require response times of about 100ms, however due to the fundamental limitation of the speed of light, the round-trip delay to the far side of the planet is at least 200ms. Some of the other causes of higher message latencies are the use of mobile wireless computers, and when connecting from home through a modem.

Though UUT by its very nature has to depend on the communication latency between a pair of users, it is possible to make RT less dependent on message latency by replicating the o-set at all participant sites. In addition to reducing RT, replication can also provide resilience to failures, which can be common in a distributed environment. Finally, multiple factors, including smaller size of messages, and batching of operations, can allow replication to consume fewer network resources than a centralized approach. We only consider *full replication* of an o-set at all participant sites. Read-only operations are immediately executed locally. The other operations are scheduled for local and remote execution by the consistency algorithm. The focus of this paper is on such consistency algorithms.

Figure 1 shows the general architecture of an application and the underlying data consistency algorithm. We assume for now that the participant sites in the o-set are fixed. User actions are translated into operations on the o-set by the application, which are then issued to the consistency algorithm. Each operation is encapsulated as an *executable object* (not the same as the objects in the o-set). The operation is *timestamped* by the algorithm, and these timestamps define an ordering on all the operations issued by all participants. It is then *disseminated* to all participants (including the issuing site), where it is *scheduled* for execution, and then *executed*. A new *view*, on an output device such as a computer screen, is then rendered. The actual execution time of each operation is short, and most of the time is spent in timestamping, dissemination and scheduling. As dissemination needs to be done by all algorithms, the main point of difference between such algorithms is the timestamping and scheduling of operations.

## 1.1 Timestamping and Scheduling

As mentioned earlier, timestamps define an order on all operations, which we call the *timestamp order*. This order is a partial order relation i.e. it is transitive and asymmetric. We only consider partial orders that are *convergent*,

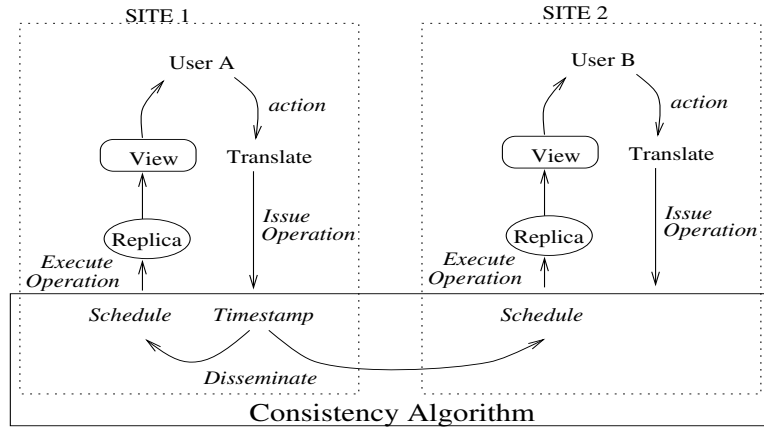


Figure 1: General Architecture with Replicated Set

i.e. execution of operations in any total order that respects this partial order will result in the same state of the o-set.

Before discussing the scheduling it is important to define our notion of *commit*. An operation  $e$  commits at a site  $p$ , when  $p$  knows that it has committed all operations preceding  $e$  in the timestamp order. If a site executes operations only after they commit at that site, it is guaranteed to respect the timestamp order. Such a scheduling algorithm is *pessimistic*. *Optimistic* algorithms execute an operation immediately after receiving it, however, they have to undo and reexecute the operation if the order of execution does not respect the timestamp order. The main difference in the general structure of the algorithms considered here from that of most replicated data algorithms are:

- *Decoupling of timestamping and execution* : The timestamping and execution of an operation are totally decoupled at the issuing site. This is possible because we use predeclared access information so even timestamping that needs such access information does not need to execute the operation. This is critical for the  $1/k$  phase algorithm discussed later.
- *Dissemination of Operation Itself* : The operation, encapsulated as an executable object, is itself disseminated. This decouples execution of the operation at any site from the execution at the issuing site. This would not be possible if we disseminated the effect of the operation i.e. the state changes made by the operation.
- *Local and Independent Commit* : The commit notion we just defined is local to a site. This is because we are not interested in the usual notion of failure atomicity, and the o-set replica at a site is not persistent across failures at that site. Replication at participant sites gives us enough fault-tolerance. Also, as we disseminate the operation itself, the commit of an operation at a site is independent of commit at the issuing site.
- *Atomicity*: By atomicity we imply that, the execution of an operation is not interleaved with the execution of another operation at the same site.

One important point to note here is that decoupling of timestamping and execution, and dissemination of the operation itself are not always good ideas. If the operation execution time is large, it is beneficial to overlap the timestamping and execution at the issuing site, and to disseminate the changes to the data made by the operation. However, in groupware, the dominant cost is timestamping, scheduling, and the rendering of the view after the execution is completed.

A previous evaluation study [5] has shown that there is no one consistency algorithm suitable for all groupware applications and network connectivity scenarios. Some algorithms assign timestamps without using the access information of an operation, for example using a Lamport clock [11] at each site to timestamp operations. In this case a total order can be defined on all operations by using the (*site identifier, Lamport timestamp*) pair as the operation timestamp. With this timestamping methodology, a site does not need to communicate with other participant

sites when assigning the timestamp. Such timestamps are classified under the general category of *independent* and *imprecise* timestamps [4]. Independent, because the timestamp is independent of the access information, and imprecise because after receiving a timestamped operation, a site cannot always decide immediately whether there are any operations preceding this operation in the global order that it has not received. To make the notion of timestamp *precision* clearer, consider vector clocks [13] that are used for defining a causal order. Vector timestamps are precise because, on receiving an operation timestamped with a vector clock, a site can immediately know whether it has received all operations preceding this operation in the causal order<sup>2</sup>. Independent and imprecise timestamps are well suited for optimistic execution as an operation can be quickly timestamped and disseminated. However, the ordering can be unnecessarily strong, as they do not use access information, and commit can be slow because of the strong ordering and imprecision of the timestamps. Also, commit requires all participant sites to be communicating. In this paper we will discuss two algorithms that use *dependent* and *precise* timestamps. Such algorithms have the potential for fast commit despite some sites being slow to communication or being disconnected. However, timestamping may require communication with some other sites which can increase the time taken to timestamp, and also the amount of communication. Therefore the key is to keep timestamping costs low while getting dependent and precise timestamps. The main contributions of this paper are

1. Identifying the key requirements of interactive groupware and defining a general algorithmic structure for consistency of the replicated data. This definition is made precise in section 2.
2. Two algorithms which generate *dependent* and *precise* timestamps. The 2 phase timestamping algorithm (section 3) gets a stamp for each of the objects accessed by an operation in a 2 phase manner, i.e. sequentially acquires the stamps and releases at the end. As acquiring and releasing the stamping for an object can involve communication with other sites, this algorithm has a high timestamping cost. The  $1/k$  phase (section 4) algorithm eliminates the release phase and concurrently gets all the stamps needed for an operation. This reduces the grow phase by a factor of  $k$ , when  $k$  objects are accessed by the operation being timestamped. This can cause cycles in the the ordering defined by these timestamps, which we resolve by traversing the cycles in the ordering graph in an identical manner at all sites. This algorithm does not deadlock and never needs to abort the timestamping of an operation.

For simplicity, the two algorithms are described assuming only incompatible write accesses on objects, and a fixed group of participants with no failures. Section 4.1 briefly discusses how to generalize the algorithms, and section 5 describes related work. Proofs are omitted from this extended abstract.

## 2 Operation Semantics and Algorithm Properties

We first give a precise specification of the operations on an o-set, assuming a single copy of the o-set. Let  $S = \{(n_1, o_1), (n_2, o_2), \dots, (n_C, o_C)\}$  be an o-set, where  $n_i$  is the name of the object with state  $o_i$ , and all the  $n_i$ 's are distinct. Also, let a *n-set* be any set of tuples, where the first element of each tuple is an object name. Therefore, an o-set is also an n-set. We define  $\mathcal{N}$  as the *name* operator, which gets the name from every tuple in a n-set. Therefore,  $\mathcal{N}(S) = \{n_1, n_2, \dots, n_C\}$ . For a set  $N$  of names, and a n-set  $D$ , the *extraction* operator  $\sqcap$  is defined as

$$D' = N \sqcap D \Leftrightarrow D' \subseteq D \wedge \mathcal{N}(D') = N \cap \mathcal{N}(D)$$

Intuitively,  $\sqcap$  extracts a subset from the n-set corresponding to the set of names.

Let  $S_i$  be the o-set before the execution of an operation  $e$  and  $S_{i+1}$  be the o-set immediately after. We consider the semantics of the add, delete and update operations by showing how  $S_{i+1}$  is a function of  $S_i$  and  $e$ .

1.  $e = \text{add}(n, o)$  : Add object  $n$  with state  $o$ . Formally,  $S_{i+1} = S_i \cup \{(n, o)\}$ .
2.  $e = \text{delete}(n)$  : Delete object with name  $n$ . Formally,  $S_{i+1} = (\mathcal{N}(S_i) - \{n\}) \sqcap S_i$ .

---

<sup>2</sup>Of course, a causal order does not necessarily satisfy the convergence property.

3.  $e = \text{update}$  : The predeclared access information, allows us to infer the read and modification sets.  $e$  reads objects with names  $R(e)$  and modifies objects with names  $W(e)$ . Also, let  $A(e) = R(e) \cup W(e)$  be the access set. The properties satisfied by the update are

**E1**  $A(e) \subseteq \mathcal{N}(S_i)$

**E2**  $\mathcal{N}(S_{i+1} - S_i) = \mathcal{N}(S_i - S_{i+1}) \subseteq W(e)$  i.e. only the state of the objects in  $W(e)$  is modified, and no objects are added to or deleted from the o-set.

**E3** The execution of  $e$  is *deterministic* i.e. given another set  $S_j$ , in which the objects that  $e$  accesses have the same state,  $e$  will behave exactly the same. Formally,

$$\forall S_j : A(e) \cap S_j = A(e) \cap S_i \Rightarrow S_{j+1} - S_j = S_{i+1} - S_i$$

For the add operation  $W(e) = A(e) = \{n\}$ , and this is the first write on the object with name  $n$ . The delete operation requires synchronization between all participants to ensure that they do not issue an update which accesses the deleted object. It can be easily implemented by multiple sub-operations, each with an empty access set, which together implement the synchronization. We limit our discussion to the interesting operations i.e. the add and update.

## 2.1 Predeclared Access Information

As mentioned before, we want to consider access semantics in the predeclared access information. This is represented as an access compatibility matrix for each object. The access compatibility matrix for an object with name  $n$  is a predicate  $\mathcal{C}_n : \mathcal{T}_n \times \mathcal{T}_n \mapsto \mathcal{B}$ , where  $\mathcal{T}_n$  is the set containing the accesses, and  $\mathcal{B} = \{\text{true}, \text{false}\}$ .  $\mathcal{C}_n(a, b)$  is true if the two accesses are commutative, else false. Also,  $\mathcal{W}_n \subseteq \mathcal{T}_n$  are the accesses which modify  $n$ . For any operation  $e$ , the application provides  $A(e)$ , as defined earlier, and for each  $n \in A(e)$ , it defines the function  $a(n, e)$  as the access on  $n$  by  $e$ . Using this we can detect if accesses on  $n$  by two operations are commutative. It also allows us to infer the modification set,  $W(e)$ , using

$$n \in A(e) \wedge a(n, e) \in \mathcal{W}_n \Leftrightarrow n \in W(e)$$

We consider one special type of write access  $w \in \mathcal{W}_n$ , which is incompatible with all other accesses, i.e.  $\forall x \in \mathcal{T}_n : \mathcal{C}(x, w) = \text{false}$ . This is used for the add operation, i.e.  $a(n, \text{add}(n, o)) = w$ .

## 2.2 General Algorithm Properties

Assume that there are  $K$  processes<sup>3</sup>  $p_1, \dots, p_K$ , all connected to an o-set. We consider an execution history of the consistency algorithm. The initial o-set replica at all the processes is the same. Let  $R$  be the set of all operations issued in an execution history and  $R_i$  be the set of operations issued by process  $p_i$ . An operation  $e$ , such that  $e \in R_i$ , is issued at  $p_i$ . This operation is then assigned a *timestamp* by  $p_i$ . The timestamps create a *partial order*  $(R, \rightarrow)$ . A *partial order* relation is asymmetric, and transitive. The timestamped operation is then broadcast to every process (including  $p_i$ ), which commit it at some time in their local history. The *commit* of  $e$  at a process  $p_j$  is represented by the local *event*  $e^j$ .

Let  $E_i$  be the set of events occurring at  $p_i$ . For convenience, we assume that the history is complete i.e. every issued operation has committed at every process. Then,

$$E_i = \{e^i \mid e \in R\}$$

The events in  $E_i$  are totally ordered using the relation  $\rightarrow_i$ , which is asymmetric and transitive, and  $(E_i, \rightarrow_i)$  is the *process history* of  $p_i$ . Therefore, the *global history*  $H$  is

$$H = [(R, \rightarrow), (E_1, \rightarrow_1), (E_2, \rightarrow_2), \dots, (E_K, \rightarrow_K)]$$

<sup>3</sup>Each process represents a different site with a replica of the o-set.

**Definition 1**  $H$  is well-formed if and only if the commit order at each process respects the timestamp order i.e.

$$\forall e_1, e_2 \in R, p_i : e_1 \rightarrow e_2 \Rightarrow e_1^i \rightarrow_i e_2^i$$

**Definition 2** The conflict predicate on any two operations  $e_1, e_2$  is defined as

$$\text{conflict}(e_1, e_2) = \exists n \in A(e_1) \cap A(e_2) : \neg \mathcal{C}_n(a(n, e_1), a(n, e_2))$$

We assume that each process somehow (optimistically or pessimistically) executes operations in its commit order. Therefore, using the operation semantics and the above definitions we can show the following.

**Lemma 1** Given a well-formed history and  $(R, \rightarrow)$ , such that  $\forall e_1, e_2 \in R : \text{conflict}(e_1, e_2) \Rightarrow e_1 \rightarrow e_2 \vee e_2 \rightarrow e_1$ , all replicas will converge to the same state.

We refer to such a timestamp order,  $(R, \rightarrow)$ , as a *convergent* order. The algorithms we consider generate a convergent order, and a well-formed history.

## 3 2 Phase Timestamping

We assume for simplicity that for any object  $n$  accessed by an operation  $e$ ,  $a(n, e) = w$ , i.e. no accesses to an object commute. Section 4.1 discusses the generalization for other accesses.

### 3.1 Tokens

Each object, say with name  $n$ , has an abstract *token*  $l_n$  associated with it. Conceptually, a token has an integer valued counter which is initialized to 0. This counter is incremented by 1 whenever a process requests a *stamp* from the token. The value prior to the increment is returned as the *stamp value*. A token supports three methods.

1. `getStamp` : Return the current counter value and increment the counter by 1.
2. `getStampAndLock` : Similar to `getStamp` , but the token is locked after the increment. Until the lock is released, other requests for a stamp will be queued.
3. `release` : Release the lock on the token so that it can satisfy other stamp requests.

There are many possible implementations of such an abstract token. For example, there could be a static site managing a token, and all methods on the token involve communicating with that site. Another option is to dynamically move the site managing the token closer to the process invoking the methods. If there is high locality and low contention for the token across processes, this can decrease the cost of the above methods. The cost of a method invocation includes the cost of communication and the time taken for the invocation to complete. Regardless of the implementation, during periods of contention, for example when there is potential for collision of entities in a virtual environment, the cost of these methods will be high. We will not concern ourselves with the exact token implementation, but assume that in general the above methods can be costly. Also, using `getStampAndLock` k can increase the time for subsequent stamp requests as they may have to queue up. Therefore we will want to minimize the total number of method invocations on tokens, and the number of `getStampAndLock` invocations.

An operation  $e$  needs to get a stamp from all  $l_n$ , where  $n \in A(e)$ . The add operation for object  $n$  is the first to get a stamp from  $l_n$  and is therefore timestamped  $\{(n, 0)\}$ . In general, timestamps are of the form  $\{(n_a, t_a), (n_b, t_b), \dots\}$ , where  $n_a, n_b, \dots$  are object names. An operation is *fully* timestamped when it has got a stamp from all the tokens it is supposed to, as decided by its access set.

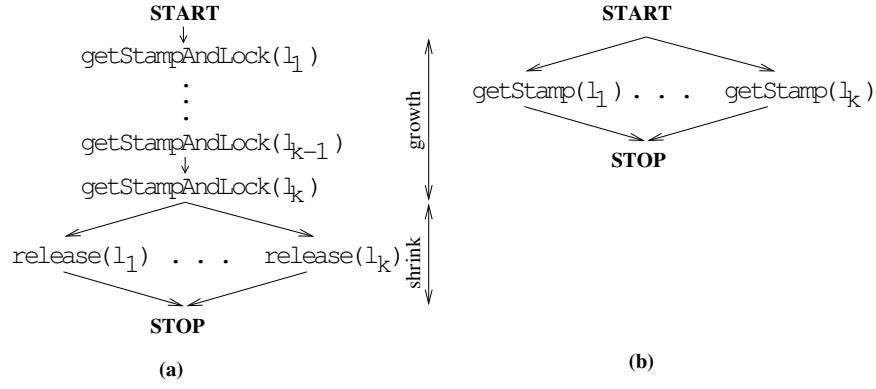


Figure 2: 2 Phase and  $1/k$  Phase Timestamping

### 3.2 The Algorithm

Let an operation  $e$  access objects  $\{1, 2, \dots, k\}$ . Figure 2(a) shows the 2 phase timestamping methodology for such an operation. Each token is locked, and stamps are requested sequentially i.e. the stamp from  $l_{i+1}$  is requested after getting the stamp from  $l_i$ . When  $e$  is fully timestamped, all tokens that were locked can be concurrently released. Deadlocks are avoided in the 2-phase timestamping scheme by getting the stamps from the tokens in a certain predefined order.

**Operation Ordering** Let  $e.ts$  be the timestamp of an operation  $e$ . For an  $n$ -set  $A = \{\dots, (n_b, t_b), \dots\}$ , define  $A[n_b] = t_b$ . Then, we define the relation  $\rightarrow^t$  as

$$e \rightarrow^t e' \Leftrightarrow \exists n \in \mathcal{N}(e.ts) \cap \mathcal{N}(e'.ts) : e.ts[n] + 1 = e'.ts[n]$$

Let the transitive closure of the above relation be the relation  $\rightsquigarrow$ . Due to 2 phase timestamping, the  $(R, \rightsquigarrow)$  relation is asymmetric, and hence is a partial order. Also, this relation is *convergent* i.e. if processes respect this order, their replicas will converge.

**Commit Rule** Process  $p_i$  maintains a current time  $T_i$ . An add operation that adds object  $n$  results in a new current time of  $T_i \cup \{(n, 0)\}$ . For an update operation  $e$ , it first checks if  $\mathcal{N}(e.ts) \subseteq \mathcal{N}(T_i)$ . If not, this operation accesses some objects that are yet to be added, so  $e$  has to wait. If yes, it then checks if

$$\forall n \in \mathcal{N}(e.ts) : T_i[n] + 1 = e.ts[n]$$

If yes,  $e$  commits, and for all  $n \in \mathcal{N}(e.ts)$ , we increment  $T_i[n]$  by 1. It is easy to see that the commit order at each process respects  $(R, \rightsquigarrow)$ , and therefore the state of the replicas converge.

The above algorithm can be slightly optimized by changing the last stamp request, `getStampAndLock( $l_k$ )`, in figure 2(a) to `getStamp( $l_k$ )`. This makes it a  $(2 - 1/k)$  phase algorithm. In the simple case when  $k = 1$ , this reduces to a 1 phase algorithm. However, in general, this algorithm has some major drawbacks. Stamps are requested sequentially, which slows down timestamping, and tokens are locked, which can increase the time to get each stamp.

## 4 $\frac{1}{k}$ Phase Timestamping

Figure 2(b) shows the timestamping procedure for the same operation using  $1/k$  phase timestamping. All the stamps are requested concurrently and no tokens are locked.

**Operation Ordering** We again consider the  $\rightarrow^t, \rightsquigarrow$  relations. The  $(R, \rightsquigarrow)$  relation can have cycles, i.e. there can be some  $e, e'$  such that  $e \rightsquigarrow e'$  and  $e' \rightsquigarrow e$ . To resolve such cycles, each operation  $e$  is also stamped with a random number  $e.r$  at the issuing process. We assume that such random numbers are unique for each operation. In reality, even with a 64 bit random number, there is always a non-zero probability that two processes with different

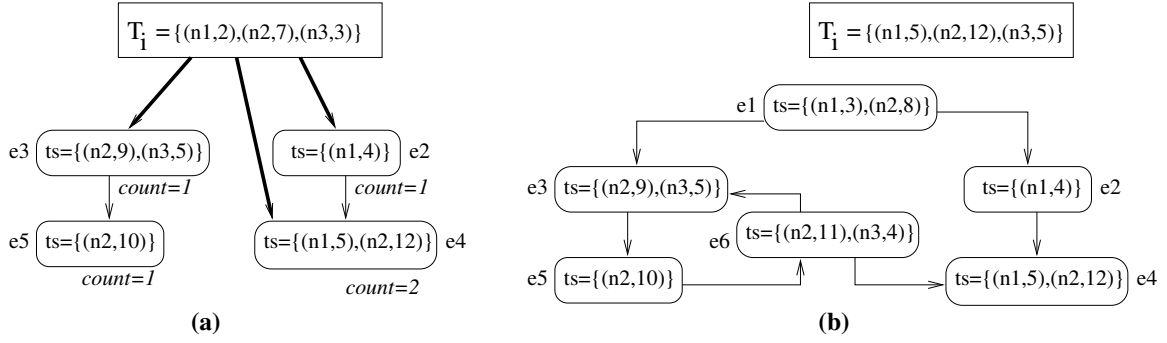


Figure 3: Detecting Ready to Commit Operations

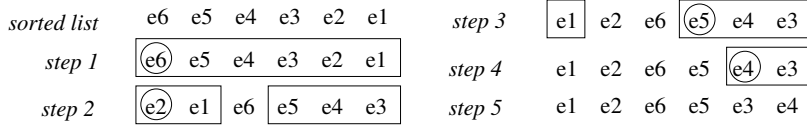


Figure 4: Commit Ordering

seeds will produce the same random number. In such cases, the unique process identifier can be appended to the random number, making it unique enough for our purposes. We can now generate a partial order  $(R, \rightsquigarrow')$  that is convergent, as follows.

$$e \rightsquigarrow e' \wedge e' \not\rightsquigarrow e \Rightarrow e \rightsquigarrow' e' \quad , \quad e \rightsquigarrow e' \wedge e' \rightsquigarrow e \wedge e.r < e'.r \Rightarrow e \rightsquigarrow' e'$$

**Ready to Commit** To ensure at each process that operations commit in an order that respects  $\rightsquigarrow'$ , we need the notion of *ready to commit*. If an operation  $e$  is ready to commit at a process, all operations  $e' \in R$  such that  $e' \rightsquigarrow e$  are also ready to commit, or have committed at that process. This implies that if  $e \rightsquigarrow e'$  and  $e' \rightsquigarrow e$ , then  $e, e'$  will be ready to commit at the same time.

We sketch the algorithm for detecting readiness to commit at a process  $p_i$ . At any instant there is a directed graph with vertices being the operations waiting to be ready, and  $T_i$ , the current time. Consider any two operations  $e, e'$  in this graph. There is an edge from vertex  $e$  to vertex  $e'$  if and only if  $e \rightarrow^t e'$ . Also, there is a heavy-edge from  $T_i$  to a vertex  $e'$  if and only if there exists a  $e$  such that  $e \rightarrow^t e'$ , but  $e$  has not yet been received at this process. The existence of such an  $e$  can be inferred at all instants at a process because the timestamps are precise. An example of such a graph is shown in figure 3(a), with the timestamps of each operation. The heavy-edge from  $T_i$  to  $e3$  is because no operation with timestamp  $(n2, 8)$  or  $(n3, 4)$  has been received. Similarly, the  $T_i$  to  $e2$ , and  $T_i$  to  $e4$  heavy-edges are due to missing the operation(s) with timestamps  $(n1, 3)$  and  $(n2, 11)$  respectively. Each vertex maintains a count of the number of different heavy-edges that can be used to reach it. Whenever a new timestamped operation is received, it is inserted into the graph, and the count values are updated. This updating can be easily done in  $O(|V| + |E|)$ , where  $V$  are the vertices and  $E$  are the edges in the graph. This is not costly, especially since we expect this graph to be small at any instant. When the count at a vertex becomes zero, it is ready to commit. In the example shown in figure 3, assume that  $e6$  is received and then  $e1$ . At this instant, all operations in the graph are ready to commit, as shown in figure 3(b), and the value of  $T_i$  is updated as in the 2-phase scheme.

**Commit** Once a subgraph of operations are ready to commit, we need to traverse this ready graph in an order that respects  $\rightsquigarrow'$ . We sketch a simple algorithm to do this in  $O(|V|^2 + |V||E|)$ . We start with the ready operations sorted in increasing order of their random stamp  $r$ . Assume that this sorted list for the example, is as shown in figure 4. The first operation in this list,  $e6$ , is picked and a graph traversal is done to find the vertices reachable from it.  $e6$  is then positioned before the vertices it can reach and after the vertices it cannot reach. This position of

$e_6$  is its final position and can potentially divide the list into two sublists. Intuitively, all operations before  $e_6$  are not in a cycle with  $e_6$ , so  $e_6$  is positioned to commit before any operations it could be in a cycle with. This step is repeated for each sublist. Each step takes  $O(|V| + |E|)$  and there are  $O(|V|)$  steps.

## 4.1 Discussion and Extensions

The  $1/k$  phase algorithm is enabled by the predeclared access information. This allows us to timestamp an operation without executing it. If we had needed to execute the operation, a timestamping cycle would have resulted in a deadlock, which would require aborting the operation.

**Token Granularity** We have assumed a token per object. When objects are very fine-grained, or when operations access many objects, the algorithm can be extended to handle multiple objects per token. However, this can make the timestamp ordering stronger.

**Using Access Compatibility in Tokens** The algorithm discussed above assumes that all accesses are non-commutative. To extend it for general access compatibility, the stamps generated by the tokens need to be changed slightly. The counter is replaced by a  $(\text{phaseCount}, \text{qCount}, \text{prevQCount})$  triple. Each phase consists of accesses that were all commutative with each other. A new phase is begun when a stamp request is received for an operation whose access is not commutative with at least one of the operations stamped in the current phase.  $\text{qCount}$ ,  $\text{prevQCount}$  keep a count of the number of operations stamped in the current phase and previous phase respectively. All stamps include the  $\text{phaseCount}$  and the  $\text{prevQCount}$ , so that each process knows when all the operations in the previous phase have been received. Using this approach the algorithms discussed earlier can be extended to handle general accesses.

**Membership Change and Failures** Dealing with changes in participants and failure of participants is very important for data management algorithms deployed in a wide-area environment. In this discussion we assume that tokens are somehow fault-tolerant, and concern ourselves with building a fault-tolerant consistency algorithm on top of such tokens. Other than method invocations on tokens, all messages in the system are for dissemination of timestamped operations to participants. Therefore, it is natural to utilize fault-tolerant multicast protocols based on virtual synchrony [7]. Such protocols initiate a flush whenever multicast group membership changes, before installing a new membership view. When such a flush is initiated, a process can have some operations that are not fully timestamped. For such operations, the process will need to abort the timestamping i.e. ignore the stamps that have already been received, and start all over again after the flush is completed. Also, all counters associated with tokens are initialized to zero after the flush completes. This can create holes in the timestamping order. For example, an operation which got the stamp  $(n, j)$  may abort timestamping, while an operation that got stamp  $(n, j + 1)$  in the same membership view may be disseminated to everyone. But we do not want this second operation to wait forever to commit. The problem is easy to solve. At the end of a flush, each process has got all the operations that were going to be delivered in a certain membership view and nothing more. Therefore, they can commit even if there are holes in the timestamping order.

## 5 Related Work

Many algorithms for interactive groupware use independent and imprecise timestamps, for example the ORESTE algorithm [10]. An important exception is the DECAF algorithm [14], which has properties of both independent and dependent timestamps. It uses a Lamport clock to timestamp operations, but the timestamp is validated by the primary copies of the objects that the operation accesses. However, as the timestamps are not precise, a process receiving a timestamped operation has to go and check with the primary copies to ensure that it can be committed. Timestamps can be rejected by the primary copies, and then timestamping has to be retried. There is also no bound on the number of times a timestamp for an operation can be rejected.

Other approaches include explicit on-demand locking, which is basically a 2-phase algorithm, like in [12], or using totally ordered multicast, like in [6]. Totally ordered multicast is effectively an o-set with one token, and each operation has to get a stamp from this token.

The five-color concurrency control protocol [8] uses predeclared access sets to do non 2-phase locking. However, like all database protocols we are aware of, it keeps the dependency graph between transactions acyclic. By decoupling the timestamping from the execution, we can handle cycles in the operation dependency graph, and at the same time never need to abort.

## 6 Conclusions

We have described the key characteristics of shared data in interactive groupware which make it different from most replicated databases. These characteristics include short operations, no explicit aborts and the ability to predeclare access information. This results in a different architecture for algorithms that manage this shared data. Due to the availability of predeclared access information, we can totally decouple timestamping and execution of an operation. Our definition of commit is local to a site, as users cannot abort operations, and replicas are not persistent across failures. By disseminating the operation itself, as an executable object, we can make the commit at each site independent of commit at the issuing site. These design choices allow us to develop an efficient alternative to 2 phase timestamping, which is  $1/k$  phase timestamping, by concurrently getting all stamps. This algorithm is deadlock and abort free.

## References

- [1] D. Agrawal, J. L. Bruno, A. El Abbadi, and V. Krishnaswamy. Relative serializability: An approach for relaxing the atomicity of transactions. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems*, 1994.
- [2] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.
- [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
- [4] S. Bhola and M. Ahamad. The design space for data replication algorithms in interactive groupware. Technical Report GIT-CC-98-15, College of Computing, Georgia Institute of Technology, 1998.
- [5] S. Bhola, G. Banavar, and M. Ahamad. Responsiveness and consistency tradeoffs in interactive groupware. In *Proceedings of 7th ACM Conference on Computer Supported Cooperative Work*, November 1998. To appear.
- [6] S. Bhola, B. Mukherjee, S. Doddapaneni, and M. Ahamad. Flexible batching and consistency mechanisms for building interactive groupware applications. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS)*, 1998.
- [7] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [8] P. Dasgupta and Z. M. Kedem. The five color concurrency control protocol: Non-two-phase locking in general databases. *ACM Transactions on Database Systems*, 15(2), June 1990.
- [9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89*, pages 399–407, 1989.
- [10] A. Karsenty and M. Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, pages 195–202, 1993.
- [11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [12] A. Prakash and H. S. Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the 5th ACM Conference on Computer Supported Cooperative Work*, 1994.
- [13] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7:149–174, 1994.
- [14] R. Strom, G. Banavar, K. Miller, A. Prakash, and M. Ward. Concurrency control and view notification algorithms for collaborative replicated objects. *IEEE Transactions on Computers*, 47(4):458 – 471, April 1998.