Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects

Robert Strom, Guruduth Banavar, Kevan Miller,
Atul Prakash, *Member*, *IEEE Computer Society*, and Michael Ward

Abstract—This paper describes algorithms for implementing a high-level programming model for synchronous distributed groupware applications. In this model, several application data objects may be atomically updated, and these objects automatically maintain consistency with their replicas using an optimistic algorithm. Changes to these objects may be optimistically or pessimistically observed by view objects by taking consistent snapshots. The algorithms for both update propagation and view notification are based upon optimistic guess propagation principles adapted for fast commit by using primary copy replication techniques. The main contribution of the paper is the synthesis of these two algorithmic techniques—guess propagation and primary copy replication—for implementing a framework that is easy to program to and is well suited for the needs of groupware applications.

Index Terms—Groupware, model-view-controller programming paradigm, replicated objects, optimistic concurrency control, optimistic views, pessimistic views.

1 Introduction

Synchronous distributed groupware applications are finding larger audiences and increased interest with the popularity of the World Wide Web. Major browsers include loosely integrated groupware applications like chat and whiteboards. With browser functionality extensible through programmability (Java applets, plug-ins, ActiveX), additional groupware applications can be easily introduced to a large community of potential users. These applications may vary from simple collaborative form filling to collaborative visualization applications to group navigation tools.

Synchronous collaborative applications can be built using either a nonreplicated application architecture or a replicated application architecture. In a nonreplicated architecture, only one instance of the application executes and GUI events are multicast to all the clients, via systems such as shared X servers [1]. In a replicated architecture, each user runs an application; the applications are usually identical, and the state or the GUI is "shared" by synchronously mirroring changes to the state of one copy to each of the others [7], [14].

In this paper, we assume that replicated architectures are used because they generally have the potential to provide better interactive responsiveness and fault tolerance, as users join and leave collaborative sessions. However, the domain of synchronous collaborative applications is broader than those supported by a *fully* replicated application architecture. For example,

- the applications may have different GUIs and even different functionality, sharing only the replicated state,
- the shared state may not be the entire application state, and
- an application may engage in several independent collaborations, e.g., one with a financial planner, another with an accountant, and each collaboration may involve replication of a different subset of the application state.

In order to support the development of such a large variety of applications, it is clearly beneficial to build a general application development framework. We have identified the following requirements for such a framework:

- From the end-user's perspective, collaborative applications built using the framework must be highly responsive. That is, the GUI must be as responsive as a single user GUI at sites that initiate updates, and the response latency at remote sites must be minimal. Second, collaborative applications must provide sufficient awareness of ongoing collaborations.
- From the perspective of the developer of collaborative applications, the framework must be application-independent and high-level. That is, it must be capable of expressing a wide variety of collaborative applications. Second, the developer should not be required to be proficient in distributed communication protocols, thread synchronization, contention, and other complexities of concurrent distributed programming.

- *R. Strom, G. Banavar, K. Miller, and M. Ward are with the IBM T.J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532. E-mail: {strom, banavar, klm, mjw}@watson.ibm.com.*
- *A. Prakash is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: aprakash@umich.edu.*

Manuscript received 1 April 1997.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 106309.

0018-9340/98/$10.00 © 1998 IEEE

We have implemented a framework called DECAF (Distributed, Extensible Collaborative Application Framework) that meets the above requirements. Our framework extends the well-known Model-View-Controller paradigm of object-based application development [11]. In the MVC paradigm, used in GUI-oriented systems such as Smalltalk and InterViews [13], view objects can be *attached* to model objects in order to track changes to model objects. Views are typically GUI components (e.g., a graph or a window) that display the state of their attached model objects; model objects contain the actual application data. Controllers, which are typically event handlers, receive input events as a result of user gestures and, in response, invoke operations to read and write the state of model objects. Updated model objects then *notify* their attached views of the change, so that each attached view may recompute itself based on the new values. The MVC paradigm has several beneficial properties, such as

1) modular separation of application state components from presentation components, and
2) the ability to incrementally track dependencies between such components.

To support groupware applications, DECAF extends the MVC paradigm as indicated in Fig. 1. First, the framework supplies generic collaborative model objects, such as Integers, Strings, Vectors, etc., to application developers. These model objects can have *replica relations* with model objects across applications, so that replicated groupware applications can be easily built. Second, it provides *atomicity* guarantees to updates on model objects, even if multiple objects are modified as part of an update. The framework automatically and atomically propagates all updates to replicas of model objects and their attached views. Third, writers can choose whether views see updates to model objects as they occur (*optimistic*) or only after commit (*pessimistic*). Fourth, applications can dynamically establish collaborations between selected model objects at the local site and model objects in remote applications. Finally, users may also code *authorization monitors* to restrict access to sensitive objects.

In this paper, we first introduce the basic concepts of the DECAF framework (Section 2). Next, we describe the distributed algorithms that implement consistent update propagation (Section 3) and view notification (Section 4). Then, we discuss performance of DECAF and experience with using DECAF (Section 5). Next, we discuss related work (Section 6). Finally, we present some concluding remarks (Section 7).

2 THE DECAF FRAMEWORK

As mentioned earlier, DECAF extends the Model-View-Controller paradigm [11]. DECAF model object classes are supplied by the framework; the application programmer simply instantiates them. In Fig. 1, A and its replica A', B and its replica B', and C are model objects. The application programmer writes views and controllers, which initiate transactions. In Fig. 1, a controller initiates transaction T that reads or updates objects A' and B'. A view V is notified

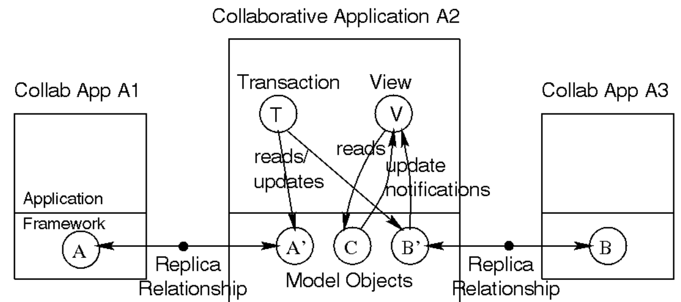

Fig. 1. Typical structure of DECAF applications.

when either B' or C' changes and it can choose to read them when it is notified. In the following subsections, we describe the key concepts in the framework and the atomicity guarantees on access to model objects provided by the DECAF infrastructure.

2.1 Model Objects

Model objects hold application state. All model objects allow reading, writing, and attaching views.

There are three kinds of model objects:

1) *Scalar* model objects, which currently are of types integer, real, and string;
2) *Composite* model objects, which support operations to **`embed`** and to **`remove`** other model objects, called *children*; currently supported composite model objects include *lists* (linearly indexed sequences of children) and *tuples* (collections of children indexed by a key); and
3) *Association* model objects, which are used to track membership in collaborations.

Model objects can join and leave *replica relationships* with other model objects. The value of an association object is a set of replica relationships that are bundled together for some application purpose. Each replica relationship in the association object contains the set of model objects that have joined, together with their sites and object descriptions.

The operations on association objects relevant to this paper are **`join`** and **`leave`**, by which a model object joins or leaves a particular replica relationship, as described in Section 2.6.

2.2 Replica Relationships

A replica relationship is a collection of model objects, usually spanning multiple applications, which are required to mirror one another's value. Replica relationships are symmetric and transitive. Model objects in a replica relationship are said to be joined in *collaboration* with each other. They are also referred to as being *replicas* of each other.

2.3 Controllers

A controller is an object that responds to end-user initiated actions, such as typing on the keyboard, clicking or dragging the mouse, etc. A controller may initiate *transactions* to update collaborative model objects. A controller may also perform other external interactions with the end user.

2.4 Transactions

Transactions on model objects are executed by invoking an **execute** method on a *transaction object*. Application programmers may define transaction objects, with their associated **execute** method, for actions that need to execute atomically with respect to updates from other users. The **execute** method may contain arbitrary code to read and write model objects within the application. Any changes to model objects will be automatically propagated to their replicas.

An example based on Fig. 1 illustrates a simple transaction object. In the figure, consider objects *A′* and *B′* in collaborative application *A*2 to hold the balances of two accounts. A transaction object **XferTrans** that transfers an amount **xferAmt** from the model object *A′* (coded as **Ap**) to *B′* (coded as **Bp**) can be written in Java, as shown in Fig. 2.

```
class XferTrans implements Transaction {
 XferTrans    (DecafFloat Ap,
              DecafFloat Bp,
              float xferAmt)
 {
   /* initialize object's private data */
 }
 public void execute () {
   if (Ap - xferAmt >= 0) {
     Ap.setValueTo(Ap.floatValue() - xferAmt);
     Bp.setValueTo(Bp.floatValue() + xferAmt);
   } else {
     throw new RuntimeException
       ("Can't transfer more than balance");
   }
 }
 public void handleAbort (Exception e)
 {/* take appropriate action */}
 /* private data */
}
```

Fig. 2. Code for a transaction object to transfer balances.

The execution of a transaction is an atomic action. That is, it behaves as if all its operations—those of the **execute** method and those that propagate changed values to replicas—take place at a single instant of time with respect to operations of other transactions, i.e., the transaction is totally ordered with respect to the times of all the other transactions in the system.

Atomicity is implemented optimistically in DECAF. Transactions may abort due to a concurrency control conflict, e.g., if two transactions originated at different sites and each transaction guessed that it read and updated a certain value of a replicated object before the other transaction did, then one of the transactions will abort.

Transactions aborted due to concurrency control conflicts are automatically reexecuted at the originating site. For most groupware applications, implicit reexecution is the desired behavior by users because transactions usually result from some user action; otherwise, users will have to explicitly redo events that generated the transaction.

A transaction may also be explicitly programmed to be aborted without retry by throwing an exception within the transaction. Any exceptions that arise during the execution of a transaction are guaranteed to be caught, since DECAF's transaction thread will catch them if the application does not. Any uncaught exceptions are turned into transaction aborts, so faulty applications will not be able to create inconsistent states or crash the entire application. In case of an abort due to uncaught exception, the transaction is not retried and a standard method, called **handleAbort()**, is called on the transaction object so that user can be notified if desired by the application.

2.5 View Objects

A view object is a user-defined object that can be dynamically attached to one or more model objects. When a view is attached to a model object, that view object will be able to track changes to the model object by receiving *update notifications* as calls to its **update** method. If a view object is attached to a composite model object, it will receive notifications for changes to the composite as well as to any of its children. The purpose of a view object is to compute some function, e.g., a graphical rendering, of some or all of the model objects it is attached to.

When the view object receives an update notification, its **update** method may take a *state snapshot* by reading any of the model objects that it is attached to. State snapshots are guaranteed by the infrastructure to be atomic actions—behaving as if they are instantaneous with respect to update transactions from controllers. Besides taking a state snapshot, the **update** method may initiate new transactions and perform arbitrary external interactions, such as rendering on the display, printing, and playing audio data.

Each update notification contains a list of all objects, and only such objects that have changed value since the last notification. Objects not on this list may be assumed not to have changed value. This information allows a view object to recompute its function more efficiently, for example, when only a part of a composite object has changed.

2.5.1 Optimistic and Pessimistic Views

View objects can be either *optimistic* or *pessimistic*. Optimistic and pessimistic views differ in the protocols for delivery of update notifications. Optimistic views are designed for responsiveness to updates, whereas pessimistic views are designed for observing only committed updates.

Pessimistic views receive update notifications only when a transaction updating an attached model object commits. The system makes two guarantees to a pessimistic view:

1) to never show any uncommitted or inconsistent values and
2) to show *all* committed values in monotonic order of applied updates.

An optimistic view will receive an update notification as soon as possible after a transaction that changes any of its attached model objects executes locally, but perhaps before its changes have propagated globally and before it commits. The state snapshot taken by an optimistic view after an update notification, therefore, may read uncommitted state, which could eventually prove to be inconsistent if the transaction aborts. If an optimistic view ever takes an inconsistent snapshot, the infrastructure will eventually execute a superseding update notification. Therefore, so long as the system eventually reaches a quiescent state, the final snapshot taken before the system quiesces will be correct.

An optimistic view will receive a *commit notification* (as a call to its `commit` method) whenever its most recent update notification is known to have been from a committed state. Committed state snapshots are always consistent and always occur in monotonic order. An optimistic view therefore trades off accuracy and the risk of wasted work in exchange for responsiveness.

To illustrate, an optimistic view object `BalanceView` that displays the value of the account balance model object `Bp` of the previous example can be written in Java, as shown in Fig. 3. The update method displays the value of the model object in a different color than the commit method, so that the end-user can be aware of the optimistic nature of the update notification.

```
class BalanceView extends TextField
                 implements OptView
{
 BalanceView (DecafFloat Bp, /*...*/) {
   /* initialize view's private data */
   Bp.attach(this);
 }
 public void update (/*... */) {
   textField.setForeground(Color.red);
   textField.setText(acctBal.toString());
 }
 public void commit () {
   textField.setForeground(Color.black);
 }

 /* private data */
}
```

Fig. 3. Code for an optimistic view object to show balance.

2.6 Collaboration Establishment

Consider a user that is running a DECAF application *A* and wishes to have another application *B*, usually belonging to another user, establish replica relationships with objects in *A*. For establishment of such relationships, the following steps must occur:

- Application *A* must create replica relationships, if they do not already exist, that are joined by objects that it wishes to share with *B*.
- Application *A* must create an *association object* `Aassoc` containing those replica relationships.
- Application *A* must then publicize the right to make replicas of its objects by creating an external token, called an *invitation*, containing a reference to `Aassoc` somewhere where application *B* can access it (e.g., on a bulletin board).
- Application *B*, usually at the request of its owner, must then *import* this invitation and use it to instantiate its own association object `Bassoc`. Object `Bassoc` must then be authorized to reveal `Aassoc`'s replica relationships.
- *B* can then read the value of `Bassoc`, discover the existence of replica relationships, and issue `join` commands to establish relationships with its objects.

Since association objects are also model objects, and can have views attached to them, changes in membership in associations are signaled as update notifications in exactly the same way as changes in values of data objects.

3 Concurrency Control

This section describes the optimistic concurrency control algorithms for propagating updates among model objects in replica relationships.

Each transaction is started at some *originating site*, where it is assigned a unique *virtual time* (*VT*) prior to execution. The *VT* is computed as a Lamport time [12], including a site identifier to guarantee uniqueness.

When a transaction is initiated, a *transaction implementation object* is created at the originating site. When updates are propagated to remote replicas, transaction implementation objects are created at those sites. Each transaction implementation object at a site contains: the *VT* of the transaction, references to all model objects updated by the transaction at that site, and state update information to carry out the update.

Each model object holds:

- *Value History:* The value history is a set of pairs of values and *VT*'s, sorted by *VT*. The value with the latest *VT* is called the *current value.*
- *Replication Graph History:* This is a similarly indexed set of *replication graphs*. A replication graph is a connected multigraph whose nodes are references to model objects, and whose multi-edges are the replication relations built by the users. It includes the current model object and all other model objects that are directly or indirectly required to be replicas of the current model object as a result of replication relations. In practice, replication graphs change infrequently.

Histories are garbage-collected as transactions commit. Committal makes old values no longer needed for view snapshots or for rollback after abort, thus, they are discarded. Since replication graphs in practice change infrequently, usually replication graph history will contain only a single graph after garbage collection.

There is a function which maps replication graphs to a selected node in that graph. The node is called the *primary copy* and the site of that node is called the *primary site*, adapting a replication technique used by Chu and Hellerstein [5] and others. The correctness of the concurrency control algorithm is based upon the fact that the order of updates of replicas is guaranteed to match the order of the corresponding updates at the primary copy, and the *VT* of each value read at a replica matches the *VT* of the corresponding value read at the primary copy. Whenever the originating site of a transaction is not the primary site of some of the objects read or written by a transaction, the transaction executes *optimistically*, guessing that the reads and updates performed at the originating site will execute the same way when reexecuted at the primary sites. If these guesses are eventually confirmed, the transaction is said to *commit*. If not, the effects of the transaction are undone at all sites, and the transaction is retried at the originating site. This strategy is derived from the optimistic guess propagation principles defined in Strom and Yemini [16], and applied to a number of distributed systems (e.g., optimistic call streaming [2] and Hope [6]). However, our algorithm makes certain specializations to reduce message traffic.

3.1 Concurrency Control for Scalar Model Objects

When a transaction T is first executed, it is assigned a virtual time, which we call t_T. As it executes, the transaction reads and/or modifies one or more model objects at the originating site. Each model object records each operation in the transaction implementation object. For each model object M read, the transaction implementation object records the *read time* t_R^M, where t_R^M is defined as the *VT* when the current value was written. For "blind writes" (writes into objects not read by the transaction), t_R^M is defined as equal to t_T. The transaction object additionally records the *graph time* t_G^M, defined as the *VT* that *M*'s replication graph was last changed.

Consider a transaction T given in Fig. 4 that originated at some site. T is assigned a *VT* $t_T = 100$. The *current values* of W, X, Y, and Z at t_T are:

- $W = 4$, last written at *VT* 80;
- $X = 2$, written at *VT* 60;
- $Y = 3$, written at *VT* 70; and
- $Z = 6$, written at *VT* 40.

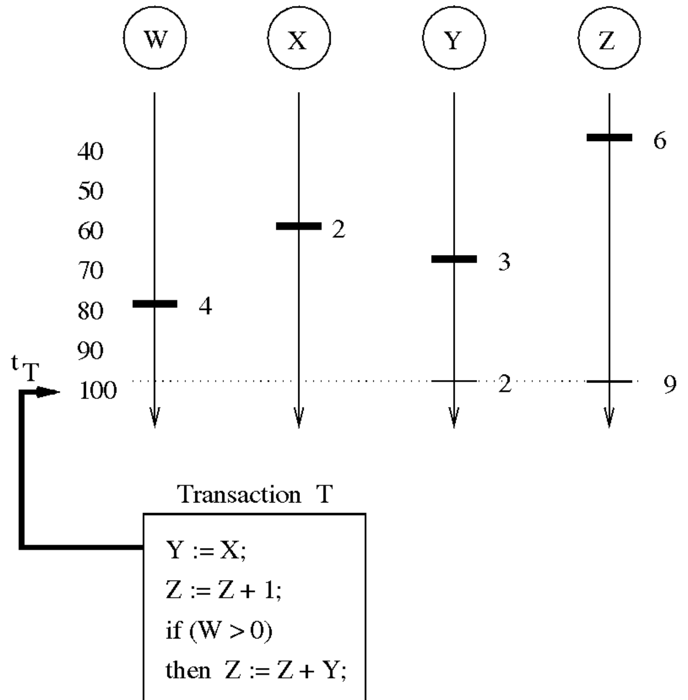

Fig. 4. Example of transaction execution.

Assuming all replication graphs were initialized at *VT* 10 (not shown in the figure), after transaction execution, the transaction implementation object will record the following:

- Read object W, $t_R^W = 80$, $t_G^W = 10$;
- Read object X, $t_R^X = 60$, $t_G^X = 10$;
- Update object Y, $t_R^Y = 100$, value 2, $t_G^Y = 10$;
- Update object Z, $t_R^Z = 40$, value 9, $t_G^Z = 10$.

Observe that the update to Y is a "blind write," since Y was not read in this transaction; hence, $t_R^Y = t_T = 100$.

The transaction implementation object next distributes the modifications to all replicas of the above model object.[1] The transaction requests each primary copy to "reserve" a region of time between t_R^M and t_T as write-free. Since replica graphs can also change (albeit slowly), the transaction must also reserve a region of time between t_G^M and t_T as free of graph updates.

As mentioned earlier, the originating site of the transaction executes optimistically, guessing that its reads and updates will be conflict-free at each primary copy. Specifically, the validity of the transaction depends upon the following types of guesses:

- **"Read committed" (RC) guesses:** That each model object value (or graph) read by the transaction was written by a committed transaction.
- **"Read latest" (RL) guesses:** That, for each value (or graph) of model object M read by transaction T, no write of M by another transaction occurred (or will occur) at the primary copy between t_R^M (or t_G^M) and the transaction's t_T. This guess implies that the primary site would have read the same version of the object had the transaction executed pessimistically. For blind writes, the RL guess check is trivially satisfied.
- **"No conflict" (NC) guesses:** That, for each model object value (or graph) written by the transaction, no other transaction reserved at the primary copy a write-free region of time containing the transaction's *VT*. This guess implies that the primary site would not invalidate previously confirmed reads by confirming this write.

For RC guesses, the originating site simply records the *VT* of the transaction that wrote the uncommitted value that was read. The originating site will not commit its transaction until the transaction at the recorded *VT* commits. For each uncommitted transaction T at a site, a list of other transactions at the site which have guessed that T will commit is maintained.

The RL and NC guesses are all checked at the site of the primary copy of an object M. The RL guess checks that no value (or graph) update has occurred between t_R^M (or t_G^M) and t_T, and if this check succeeds, creates a *write-free reservation* for this interval so that no conflicting write will be made in the future; the NC guess checks that no write-free reservation has been made for an interval including t_T.

For each object M read but not written, a message is sent to the primary copy (if it is at a remote site). This message contains t_R^M, t_G^M, and t_T. Each primary copy object then verifies the RL guesses for values and graphs. A confirmation message is then issued, confirming or denying the guess. In the general Strom-Yemini approach [16], this confirmation message would be broadcast to all sites. But, in the DECAF implementation, this confirmation is sent only to the originating site. It is a property of our DECAF implementation

1. For scalar objects, such as integers, it suffices to distribute the final value; for composite objects, it is usually efficient to distribute the change as an increment change. These differences are important for reducing the communications bandwidth, but they do not affect the concurrency control algorithm.

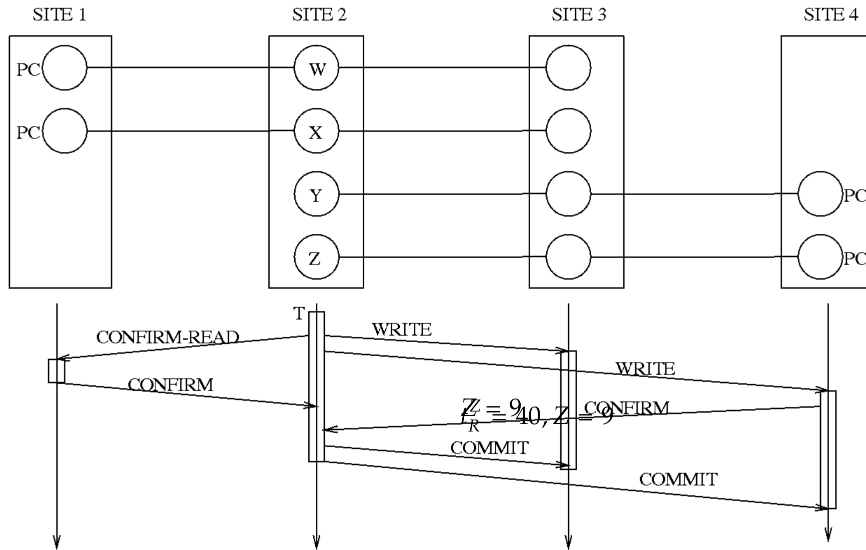

Fig. 5. Example of update propagation.

that the originating site always knows the totality of sites affected by its transaction by commit/abort time. Therefore, the originating site is in a position to wait for all confirmations to arrive and then to forward a summary commit or abort of the transaction as a whole to all other sites. This avoids the need for each primary copy to communicate with all nonprimary sites, and it avoids the needs for nonprimary remote sites to be aware of guesses other than the summary guess that "the transaction at virtual time t_T commits".

For each object M modified by T we send a message to all relevant sites, containing t_R^M, t_G^M, t_T, and the new value. However, while all sites, other than the primary site, simply apply the update at the appropriate VT, the primary site additionally performs the RL and NC guess checks and then sends a confirmation message to the originating site.

The originating site waits for confirmations of guesses from remote primary sites. If all guesses for a transaction are confirmed, the originating site commits the transaction and sends a commit message to all remote sites that received update messages. If any guess is denied, the originating site aborts the transaction and sends an abort message to all remote sites. The originating site then reexecutes the transaction.

If a site detects that a transaction at VT has committed, the modified model objects at that site are informed. This notification can be used to schedule view notifications and eventually to garbage-collect histories. The site retains the fact that the transaction has committed so that if any future update messages arrive, the updates are considered committed.

If a transaction is aborted, the modified model objects are informed so that the value at VT can be purged from the history. The site retains the fact that the transaction has aborted so that if any future update messages arrive, the updates are ignored.

A further optimization is performed in the case where a transaction has only one remote primary site, a common case in two-party and even multiparty collaborations. Assume there is only one remote primary site and that there are no RC guesses (all the objects read by the transaction were already committed). In that case, rather than waiting for the single primary site to send a confirmation back to the originating site (which would then send a summary commit), the originating site "delegates" the responsibility for committing the whole transaction to the single remote primary site. In that case, the message would contain the site identifiers of all the remote sites affected by the transaction.

Besides two-party collaborations, the above optimization is applicable to many other DECAF applications. In many applications, all the objects affected by a transaction are replicated at all the sites, resulting in isomorphic replica graphs for all the objects. Our primary site selection algorithm, which simply computes a function of the replication graph, thus selects the same primary site for all objects in such a case. Therefore, transactions in such applications either have no remote primary site or have a single remote primary site.

Let us examine how these algorithms would apply in our example, shown in Fig. 5. Suppose there are four sites, and that W and X are replicated at sites 1, 2, and 3, while Y and Z are replicated at sites 2, 3, and 4. Suppose that T is initiated at site 2. Suppose further that the primary site of W and X is 1, and of Y and Z is 4. Ignoring graph times and graph updates for now, and assuming that the three current values read by the transaction were committed (hence, there are no RC guesses), the following messages are sent from site 2 after the transaction is applied locally (we perform the obvious optimization of sending messages only to relevant sites):

1) To site 1: CONFIRM-READ $t_T = 100, t_R^W = 80, t_R^X = 60$
2) To sites 3 and 4: WRITE $t_T = 100, t_R^Y = 100, Y = 2$, $t_R^Z = 40$, $Z = 9$

Site 1 checks that W is write-free for the VT range from 80 to 100 (RL guess check) and that X is write-free for the VT range from 60 to 100 (RL guess check). If so, it reserves those times as write-free and sends a CONFIRM to site 2.

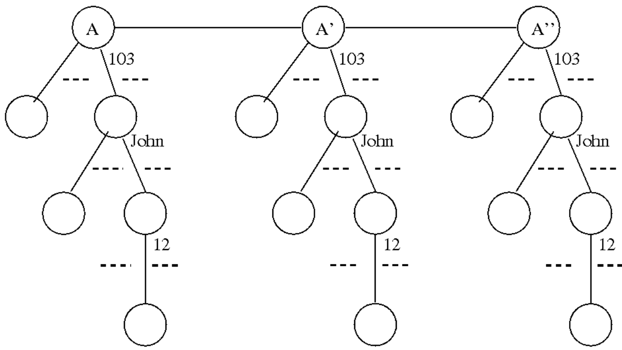

Fig. 6. Replicas of composite model objects.

Site 3 simply applies the updates to its Y and Z replica objects.

Site 4 checks that Z is write-free for the VT range from 40 to 100 (RL guess check). If so, it reserves those times as write-free. It also checks that writing Y or Z at VT 100 does not conflict with any previously made read reservations (NC guess checks). If all the above checks succeed, it applies the updates and sends a CONFIRM to site 2.

Site 2 awaits both responses. If both are confirmations, it sends COMMIT 100 to all other sites involved.

If, instead of sites 1 and 4, site 3 were the primary site for all the model objects involved, then the delegate commit optimization would apply. Site 2 would send site 3 the parameter "delegate commit" and the list of replica sites on the message containing the WRITE and CONFIRM-READ requests. Site 3 would then directly send COMMIT 100 to all other sites rather than returning a response to site 2.

3.2 Concurrency Control for Composite Model Objects

Although the concurrency control algorithm is the same for composite objects as for scalar objects, it is desirable to save space by not keeping a separate replication graph for each object inside a composite. That is, if composite A is a replica of composite A' and A'' (see Fig. 6), we wish to avoid encoding inside object A[103] the information that it is a replica of objects A'[103] and A''[103].

Our approach is that, by default, an object embedded within a composite inherits the replication graph of its root; e.g., A[103]'s replicas would be at the same sites as A's replicas, at the corresponding index (103). Similarly, if A[103] is itself a composite object, its embedded objects, e.g., A[103][John][12] would be replicated at the same sites.

The set of indices between the root and a given object is called its *path*; when an object such as A[103][John][12] is modified, the change and the path to A are then sent by A to its replicas A' and A'', which then use the same path name, [103][John][12], to propagate the update to their corresponding components A'[103][John][12] and A''[103][John][12]. We call this technique *indirect propagation* of updates, in contrast to the *direct propagation* technique discussed earlier, in which each object holds its own replication graph and communicates directly to its replicas.

In addition to saving space, indirect replication avoids the problem in direct replication that small changes to the embedding structure could end up changing a large number of objects. For example, if indirect replication were not used, adding a new replica A''' to the set $\{A, A', A''\}$ would entail updating the replication graph for every object embedded within A and its replicas. Similarly, removing A[103] from A would entail updating the replication graph for every object embedded within A[103].

3.2.1 Adjustments to Support Indirect Propagation

There are two adjustments that have to be made to ensure the correctness of the concurrency control algorithm in the presence of indirect propagation.

The first has to do with the relative order of list items. A transaction at VT 100 may modify A[103][John][12] without having seen that an earlier transaction at VT 80 that deleted A[5] so that what the originating site thinks of as A[103] may appear to some other sites to be A[102]. This is not a concurrency control conflict, because the two transactions read/update different objects. It is simply a consequence of the fact that path names like [103][John][12] are fragile. To overcome this, in addition to using the actual list index in a path name, the propagation algorithm includes the VT at which the object was updated as a tag to the index—e.g., if A[103] was embedded in A by a transaction at VT 40, then 40 is used as a tag to the index 103.

A composite object receiving such an indirect propagation message can always propagate it down the tree regardless of the order in which it has received other structure-changing operations. During such a propagation, if it is determined (using the VT tag) that an earlier path changing update has not yet been received, the propagation will block until the earlier update is received.

The second adjustment has to do with guesses associated with the paths for indirect object propagation. The updated model objects must make RC guesses to ensure that transactions that created their paths have committed and RL guesses that no straggling transactions have removed any component of their paths.

3.2.2 When Indirect Propagation Is Not Possible

Indirect propagation is the default mode of propagating value updates to objects within composites. However, indirect propagation is not always possible. Consider the configuration in Fig. 7. In this case, node C can indirectly propagate changes to C', but node B cannot because it has a different set of replicas than the rest of the tree. We therefore use direct replication for objects B, B', and B''.

If indirect propagation were used, then a transaction that originates at Site 1 and updates B could propagate the update to B' at Site 2, but the originating site would not know about Site 3 and object B''. We could require Site 2 to propagate the update from B' to B'' independently (*transitive propagation*), but that introduces problems in committing the transaction. So far, it has always been the case that the originating site (or a site delegated by the originating site) is in a position to know all the guesses on which a transaction depends, and is able to send summary commit messages to all sites when those guesses are all confirmed. If Site 2 were to propagate the update to B'', it would require the use of a nested transaction protocol, complicating the concurrency control algorithm.

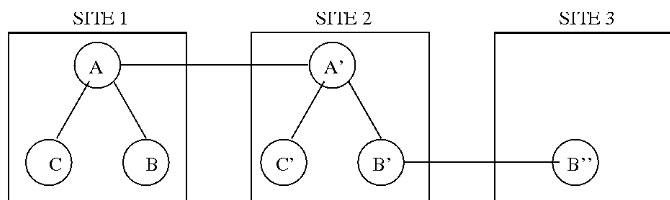

Fig. 7. Indirect propagation not possible for this case.

Rather than introducing a nested transaction protocol, we avoid the problem by using direct rather than indirect propagation for node B. That is, the replication graph of object B at all sites, including Site 1, includes both B' and B''. Note that the case shown in Fig. 7 occurs only when there is replication among both roots and nonroots of composites or when there is aliasing and, thus, non-tree-structured composites result.

In our implementation, indirect propagation is the default. Once a collaborating node is embedded within another collaborating node (either because it was previously collaborating and is now embedded, or because it was previously embedded and is now collaborating, or because the parent node was previously noncollaborating and is now collaborating), that node switches to direct propagation, and a propagation graph is sent to all replicas. The parent node notifies the collaborating embedded node of all changes to its replica graph.

3.3 Dynamic Collaboration Establishment

The set of replica relations between objects remains relatively static. Most transactions change the values of objects rather than the replication graphs. But, replication graphs do change, as users join and leave collaborations. Direct propagation graphs for embedded objects inside composites can also change as a result of deleting objects from composites and embedding them elsewhere. Dynamic collaboration establishment transactions need not be especially fast, but they must work correctly in conjunction with all the other transactions.

We have already seen some of the effects of dynamic collaboration establishment in the algorithms described above. Replication multigraphs are timestamped with the transaction's *VT* that changed them. There is no "negotiation" for primary copy; recall that each node is able to map a given multigraph to the identity of the primary site for that configuration. A primary copy always confirms the RL guess that the graph hasn't changed, as well as confirming whatever else it is being asked to check; this guards against the possibility that the originating site is propagating to the wrong set of sites or that it is asking the wrong primary copy because of a graph change that it hasn't seen yet. A primary copy always reserves the graph against changes during a region of time that a previously confirmed transaction has assumed to be change-free.

Additionally, our algorithm does not include a distributed "election" procedure for primary copies. The election procedure is implicit in the constant function mapping graphs to primary copies. As a result, there is no phase during which updates are not possible because a primary site is being chosen.

It remains to show the protocol for joining (or leaving) collaborations. Let us assume that a model object A is already collaborating with one or more model objects. It is going to join a collaboration with a number of other model objects including B.

When A's owner initiates a transaction including the operation of A joining B's replica relationship via *Aassoc* (see Section 2.6), the following happens:

- The value of *Aassoc* is read. It contains a reference to one of the objects in the replica relationship A is trying to join, in this case, B. The value is then optimistically updated to reflect that A has joined the relationship. This is treated for concurrency control purposes like the read and update of any other value—that is, the current value is used optimistically, and the new value is broadcast, but a request for confirmation is sent to *Aassoc*'s primary copy.
- A remote call is made to B, sending it A's replication graph g_A.
- B computes as the return value: B's replication graph g_B, B's value, and, if objects embedded in B are using direct propagation, their replication graphs as well. The return value is sent back to A.
- In parallel, B merges g_A and g_B, passing this merged value to all B's replicas. This update must be confirmed by g_B's primary site, and the confirmation returned to A via a separate message (unless g_B's primary site is itself the site of B).
- If the current value of g_B is uncommitted, then this fact is remembered at B, and A is additionally notified in the reply that it must wait for the transaction that wrote this value to commit.
- When A receives the value of B and the graph(s) g_B, A merges g_B with g_A and passes this merged graph (together with the value of B) to all A's replicas. This update must be confirmed by g_A's primary site, and the confirmation returned to A via a separate message (unless g_A's primary site is itself the site of A).
- A's site can commit when it has heard:
  a) that g_A's primary site confirmed the update of the replication graph,
  b) that g_B's primary site confirmed the update of the replication graph,
  c) if necessary, that the transactions that wrote the value of g_A and g_B have committed, and
  d) that *Aassoc*'s primary copy has confirmed.

Although this protocol is more complex because of the need for A to make a remote call to B, it remains the case that the originating site always knows the totality of sites to which it must send a commit or abort. Therefore, it remains true that sites other than the originating site need only remember their dependency on a transaction identified by a particular virtual time.

3.4 Dealing with Client Failures

A client in a collaborative environment may abruptly quit or kill their session, e.g., by quitting the Web browser that launched the collaborative applet. Connectivity to a client may also be lost, for instance, if the client is connected over a modem and the modem drops the connection. Since such situations can occur during a collaboration, we describe our

solution to making the concurrency control protocol resilient to such client failures.

For the above failures, we assume that the underlying communication infrastructure provides notification of such failures and, as common in systems such as ISIS [4], presents them to the application as fail-stop failures—further communication with failed or disconnected clients is prevented by the communication layer until these clients rejoin the collaboration by going through a join protocol as new members.

For transactions that are in progress during a site failure, the protocol needs to be resilient to the following types of failures:

1) failure of an originating site of a transaction and
2) failure of a primary site for a transaction.

If the originating site fails, the remaining sites, upon failure notification, simply determine if any of them received a commit message, which are logged and garbage collected, regarding the transaction. If so, the transaction is committed at all the sites; else, it is aborted. If the primary site fails before the transaction commits, the transaction is aborted; it is retried later after the graph update has committed and a new primary site is identified to coordinate the transaction.

When the sites are notified of a site failure, the replication graphs of various objects that were at the failed site need to be updated using timestamped transactions. This is not a problem as long as the primary site for a replication graph is not the failed site; the primary site can coordinate the transaction for that replication graph. If the primary site for coordinating updates to a replication graph is the one that failed, then there is a circularity problem—a primary to coordinate the update to the replication graph cannot be determined until the graph update has committed because the primary site is a function of the replication graph, but a primary site is needed to commit the transaction. To break the circularity, we simply use an alternative protocol to commit the graph update as an atomic action. The remaining sites use a distributed consensus protocol to first commit any conflicting transactions that are known by any of them to have committed, abort any other transactions that conflict with the replication graph update transaction, and, then, apply the graph update at a common virtual time. If the underlying communication layer reports failures during this protocol, the protocol is repeated until all the fail notifications are successfully applied to the graphs. This consensus-based protocol is slower than our primary-site based commit protocol for other transactions, but, in practice, is satisfactory because ungraceful site crashes are rare compared to other transactions.

4 View Notification

This section describes the algorithms for implementing the view notification semantics given in Section 2.5.

When a transaction implementation object completes executing at a site (that originated the transaction or not), the DECAF infrastructure initiates *view notifications* to be sent to all the view objects attached to the model objects updated in the transaction. View object attachments are always local, i.e., views are always attached to model objects at the same site. Thus, a view notification is simply a method call to the `update` method implemented by the view object. The `update` method can contain arbitrary code that takes a state snapshot by reading the view's attached model objects and recomputes its display. The infrastructure guarantees that such a state snapshot is implicitly a consistent atomic action.

For every view notification initiated, a *snapshot object* is created internal to the DECAF infrastructure. All the snapshots associated with a particular user level view object are managed internally by a *view proxy* object. Each model object contains the set of view proxies corresponding to its views, which it notifies upon receiving an update or a commit.

Since a snapshot is an atomic action, it is assigned a virtual time t_S. Each snapshot is assumed to read *all* the model objects attached to the view at *VT* t_S. These reads may be optimistic; hence, as described in Section 3.1, their validity depends upon the read values being the latest (RL guess) and the read values being committed (RC guess). An RC guess, but not an RL guess, is needed for values updated at t_S. Confirming RL guesses involves remote communication with the primary copies of the objects read in the snapshot. If the RL and RC guesses are confirmed for each attached object, the snapshot is said to *commit*.

Optimistic and pessimistic views differ in two respects. First, they differ in the time at which view notifications are scheduled. Optimistic notifications are scheduled as early as possible, i.e., as soon as a transaction executes locally and updates an attached model object. Pessimistic notifications are scheduled after it is known that the snapshot will be valid, i.e., that the view will read consistent committed values. Second, they differ in the lossiness of notifications. Pessimistic views are notified losslessly of every committed update in monotonic order of updates, whereas optimistic views are only notified of the latest update. That is, if a straggling update occurred at a *VT* earlier than that of the latest snapshot, that update is not notified to an optimistic view. Subsections 4.1 and 4.2 describe these behaviors in more detail.

As described in Section 2.5, view notifications are incremental, i.e., each notification provides only that part of the attached model object state that has changed since the last notification. However, for the sake of simplicity, the algorithms presented in this section do not incorporate incrementality; each snapshot is assumed to read the set of attached model objects in its entirety. Furthermore, notifications may be bundled to enhance performance, i.e., a single view notification may be delivered for multiple model objects that were updated in a single transaction.

4.1 Optimistic Views

Fig. 8 shows an optimistic view *V* attached to model objects *A* and *B*. The view proxy object *VP* manages delivery of view notifications to *V*. *A* and *B* have committed current values (i.e., values with the latest *VT*) at *VT*'s 100 and 80, respectively. A transaction *T* runs at *VT* 110 and updates *A*, which notifies its view proxy *VP*.

The primary requirement of optimistic views is fast response. Consequently, as soon as *VP* is notified, it performs the following:

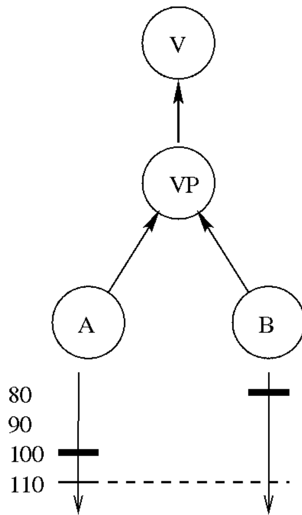

Fig. 8. View notification.

1) It creates a snapshot object and assigns it a *VT* t_S equal to the greatest of the *VT*'s of the current values of all attached model objects. In this case, $t_S = 110$.
2) It schedules a view notification, i.e., calls the view *V*'s `update` method.

At the end of the snapshot, the snapshot object records that all attached model objects were read at t_S. In order for the snapshot in this example to commit, two guesses must be confirmed (as before, we ignore guesses related to the graph):

1) An RC guess that the update to *A* by transaction *T* at *VT* 110 has committed. This requires receiving a COMMIT message from the site that originated transaction *T*. (No RL guess is required for *A*.)
1) An RL guess that the *VT* interval from 80 to 110 is update free for *B*. This requires sending a CONFIRM-READ message to *B*'s primary copy and waiting for the response.

Eventually, if these guesses are confirmed, then the snapshot commits, and a *commit notification* is sent to *V*, i.e., its `commit` method is called.

If, on the other hand, an RC guess turns out to be false, the view proxy reruns the snapshot with a new t_S. In the example of Fig. 8, if the RC guess was denied as a result of transaction *T* at *VT* 110 aborting, a new snapshot is run. This snapshot will have $t_S = 100$, since that is now the greatest *VT* of the current values of all attached model objects. Notice from this example that optimistic view notifications are not necessarily in monotonic *VT* order.

In the case that an RL guess is denied by the primary copy, that means that the requested interval is not update free and, thus, a straggler update is yet to arrive at the guessing site. In this case, the straggler itself will eventually arrive and cause a rerun of the view notification. In the example in Fig. 8, if the RL guess was denied as a result of a straggler update to *B* at *VT* 105, the update at *VT* 105 will trigger a new view notification at $t_S = 110$.

This algorithm implements the liveness rule for optimistic views that an update notification is followed either by a commit notification or, in the case of an invalid optimistic guess or a subsequent update, a new update notification.

An optimistic view proxy maintains at most one uncommitted snapshot—the one with the latest t_S—at any given time. If a new update arrives before the current snapshot has committed, then we're obliged to notify the new update to the view due to the responsiveness requirement. The system may as well discard the old snapshot since there is no way to notify the view of its commit (as we don't expose *VT*'s to views). As a result, an optimistic view gets a commit notification only when the system quiesces, that is, when no new updates are initiated in the system before existing updates are committed.

4.2 Pessimistic Views

Recall that the system makes two guarantees to a pessimistic view:

1) never to show any uncommitted values, and
2) to show *all* committed values in monotonic order of applied updates.

A pessimistic view proxy initiates a snapshot at every *VT* for which one or more of its attached model objects receive a committed update. However, it doesn't schedule a view notification for the snapshot until the snapshot commits. Snapshot committal depends on

1) the validity of model object reads at the snapshot's t_S, and
2) whether its preceding snapshots have already committed (this is due to the monotonicity requirement).

When one or more snapshots commit, the view is notified, once for each committed snapshot, in *VT* sequence. Thus, unlike an optimistic proxy, a pessimistic proxy manages several uncommitted snapshots.

A pessimistic view proxy thus contains a list of snapshot objects sorted by *VT*. It also contains a field *lastNotifiedVT*, which is the *VT* of the last update notification.

To illustrate pessimistic view notification, let us say that the view *V* in the example of Fig. 8 is a pessimistic view. Suppose that *lastNotifiedVT* = 80. Suppose, further, that the snapshot at *VT* 100 is as yet uncommitted and, thus, *A*'s committed update at *VT* 100 is not yet notified. When the transaction at *VT* 110 commits, it informs the model object *A*, which, in turn, informs the pessimistic view proxy *VP*. *VP* creates a snapshot object, assigns it a $t_S = 110$, and records the following guesses:

1) An RL guess that the *VT* interval from 100 to 110 is free of committed updates for *A*. This stems from the monotonicity requirement. This requires sending a CONFIRM-READ message to *A*'s primary copy and waiting for a response.
2) An RL guess that the *VT* interval from 100 to 110 is free of committed updates for *B*. This requires a CONFIRM-READ message as above.

Eventually, if all the guesses made by a particular snapshot object are confirmed and, if it is the earliest snapshot since *lastNotifiedVT*, it commits. A view notification is scheduled, and *lastNotifiedVT* is updated. This may result in other contiguous committed snapshots after *lastNotifiedVT* to be scheduled as well.

A straggling committed update, say at *VT* 105 for *B* in the example, may cause an RL guess to be negated. In this case, when the straggling committed update is notified to the proxy, a new snapshot is created at *VT* 105 as given above. Additionally, the RL guess made by the succeeding snapshot at *VT* 110 (guess 2 above) is revised to be for the *VT* interval from 105 to 110 for *B*.

This algorithm implements the consistency and monotonicity requirements for pessimistic views.

5 Discussion

5.1 Performance Analysis

DECAF provides fast response and fast commit to views, assuming a low conflict rate (i.e., transaction retry rate) in typical collaborative use.

5.1.1 Commit Latency

Let the average network latency of a single point-to-point message be t ms. Assume that processing of a message is negligible compared to the average latency. Say a transaction updates m model objects, each of which has its own replicas. The site originating the transaction applies the updates immediately, giving interactive response time for notifications to optimistic views at the originating site. Each of the replica sites receives the update with one message, in t ms, resulting in low latency for notifications to optimistic views at the replica sites.

The transaction commits when all of the m primary copies confirm back to the originating site (after $2t$ ms), and each of the replica sites receives a commit message. Thus, a transaction commits in $2t$ ms at the originating site and in $3t$ ms at other sites. If a transaction's read/write set includes only objects from a single primary site, then the protocol is even faster. If the single primary site happens to be the same as the originating site, then the transaction commits immediately at the originating site and in t ms at all other sites; if the single primary site is anywhere else, then the transaction commits in t ms at that site and in $2t$ ms elsewhere. A commit notification to an optimistic view is as fast as an update notification to a pessimistic view, which is described below.

5.1.2 View Notification Latency

Say a pessimistic view has n objects attached to it. A particular transaction may update some or all of these n objects. An update notification is scheduled to the view after the updating transaction commits (which takes $3t$ ms) *and* all of the n primary copies confirm that there are no conflicts (see Section 4.2). However, these latter confirmations proceed concurrently with the confirmations required for the transaction's commit. Thus, it takes only $2t$ ms to notify an update to a pessimistic view at the originating site. Furthermore, it takes no more than $3t$ ms to notify an update to a pessimistic view at a nonoriginating site. This is because, for objects that are updated in the transaction, confirmations are eagerly distributed by the primary copy when the originating site requests confirmation; thus, it takes $2t$ ms from the start of the transaction. For objects that are being viewed but were not updated by the transaction, the view proxy explicitly requests confirmation as soon as the update is received, after t ms. Thus, confirmation responses will have arrived in $3t$ ms.

In general, an optimistic view notification will occur $2t$ ms before the corresponding pessimistic view notification. On the other hand, an optimistic view may experience one of the following three types of deviations from the "ideal" notification sequence of one notification per committed transaction:

1) *lost updates*, which occur when an update message for a model object M arrives with a virtual time earlier than that of a previously processed update message for M, in which case the message with the earlier virtual time does not yield a notification;
2) *update inconsistencies*, which occur when an update is delivered to an object M, but the transaction is later rolled back;
3) *read inconsistencies*, which occur when a view is observing model objects $M1$ and $M2$, and receives an update notification for $M1$, and, subsequently, an update message for $M2$ arrives with an earlier virtual time.

In an application in which all operations are "blind writes" (e.g., a whiteboard or a collaborative form), there are no update inconsistencies, because concurrency control tests never fail. However, lost updates and read inconsistencies may still occur.

5.1.3 Scalability

In DECAF's environment, there are two different dimensions of scalability: size of collaboration group and size of network. Although we believe our concurrency control algorithm scales well to large groups, in our opinion, in such environments, issues of fault tolerance dominate and, therefore, we consider the question of scalability to very large groups to be beyond the scope of this paper.

With respect to size of networks, the main advantage of the DECAF algorithms derives from its use of a different primary site for each collaboration set, rather than the use of a network-wide global sweep. Previous systems for groupware (e.g., COAST [15], ORESTE [10]) have employed algorithms based on a global sweep, such as Jefferson's Global Virtual Time algorithm [9]. In such systems, commit speed depends upon the frequency of global sweeps. In Jefferson's original applications of Global Virtual Time, there was no hurry to commit, since commit messages were primarily used to garbage-collect message queues. But, in a collaborative application, commit messages are needed to enable the notification of pessimistic views, and, therefore, they directly affect responsiveness. In a hypothetical example of a very large network with large numbers of relatively small replica sets (e.g., replicas at sites *A*, *B*, and *C*, at sites *C*, *D*, and *E*, at *E*, *F*, and *G*, etc.) the sweep to compute a GVT can be very time-consuming, since it is proportional to the size of the network. But, in our algorithm, each replica set will have its own primary site, and each transaction will require confirmations from a very small number of such primary sites.

5.2 Status and Experience

A prototype implementation of the DECAF framework has been completed in the Java programming language. The

framework currently supports scalar model objects, transactions, and optimistic and pessimistic views. The implementations of these objects use the algorithms described in this paper.

DECAF has been evaluated so far in two types of scenarios: realistic groupware applications and performance benchmarks. The realistic applications were useful for evaluating subjective factors, such as ease of programming, learning curve, etc., but were not suitable for performance analysis. The benchmarks were simpler applications designed to ascertain the working limits of the algorithms and the differential properties of optimistic and pessimistic views.

5.2.1 Groupware Applications

Several collaborative applications have been successfully built using the current prototype implementation. These include several groupware applications that allow an insurance agent to help clients understand insurance products via data visualization and to fill out insurance forms, a multi-user chat program, and simple games. Our preliminary experience is that it is easy to write new applications or to modify existing programs to use our MVC programming paradigm. Optimistic views have been very useful due to their fast response, and also due to the low conflict rate in typical use. Pessimistic views have also been useful for views that want to track all committed changes to the values of model object.

5.2.2 Benchmarks

Benchmarks were built to evaluate

1) how quickly optimistic and pessimistic views responded under light load conditions;
2) how conflict rate affected lost messages and rollbacks.

A fuller discussion appears in our research report [3].

Latency of optimistic and pessimistic views was measured under a range of artificially induced network delays, and the observed latencies closely matched the analytical expectations.

Under loaded conditions, transactions involving only blind-writes were measured to determine the impact on optimistic views due to lost updates. Even at rates of one update per second from both parties of a two-party collaboration, the lost update rate was below 20.1 percent.

Many collaborative applications do not care about lost updates, since the end user is only interested in knowing the latest value of a shared object: a lost update will usually be indistinguishable from two updates in rapid succession. However, "glitches" due to temporary inconsistencies might be more troublesome in some applications.

Measurements show that, for transactions involving both reads and writes and one party updating once per second on the average, an update rate by a second party of once per three seconds or more produced rollback rates below 2 percent; at higher update rates, rollbacks were frequent enough to produce significant rates of update inconsistencies. This suggests that it may be desirable to suppress optimism when conflict rates exceed a certain threshold.

5.3 Future Work

DECAF is currently being integrated into a complete environment for programming collaborative applications. This environment includes facilities for collaboration session management, audio and video streaming, and a variety of pluggable components with which application programmers can build entire collaborative solutions. In addition, several optimizations described in this paper are forthcoming, including commit delegation, faster commit of snapshots, and incremental propagation. We are also incorporating a persistence store and recovery from a variety of failures into the algorithms of DECAF.

6 RELATED WORK

The DECAF framework is designed for collaborative work among a small and, possibly, widely distributed collection of users. Consistency, responsiveness, and ease of programming are important objectives.

ISIS [4] provides programming primitives for consistent replication, although its implementation strategies are pessimistic.

Interactive groupware systems have different performance requirements and usage characteristics from databases, leading to different choices for concurrency control algorithms. First, almost all databases use pessimistic concurrency control because it gives much better throughput, a major goal of databases. In interactive groupware systems, on the other hand, pessimistic concurrency control strategies are not always suitable because of impact on response times to user actions—ensuring interactive response time is often more important than throughput. Second, possibilities of conflicts among transactions is lower in groupware systems because people typically use social protocols to avoid most of the conflicts in parallel work.

Optimistic protocols based on Jefferson's Time Warp [9] were originally designed for distributed simulation environments. They have been successfully applied in other application areas as well [8]. However, one important characteristic of distributed simulation is that there is usually an urgency to compute the final result, but not necessarily to commit the intermediate steps. In these protocols, the primary purpose of "committing" is to free up space in the logs, not to make the system state accessible to view. But, in a cooperative work environment such as ours, fast commit is essential. The delay associated with waiting for, at most, a single primary site per model object in DECAF is typically considerably less than a Time Warp style global sweep of the system would be.

The ORESTE [10] implementation provides a useful model in which programmers define high-level operations and specify their commutativity and masking relations. One drawback is that there are no high-level operations on multiple objects nor are there ways of combining multiple high-level operations into transactions. To get the effect of transactions, one must combine what are normally thought of as multiple objects into single objects and, then, define new single-operand operations whose effects are equivalent to the effects of the transaction. One must then explicitly specify the interactions between the new operations and all the other operations.

There is also a subtle difference between the correctness requirements in DECAF and in ORESTE. This difference results from the fact that ORESTE only considers quiescent state—the analysis does not consider "read transactions" (e.g., snapshots) that can coexist with "update transactions". For instance, in the ORESTE model, a transaction that changes an object's color can reasonably be said to commute with a transaction that moves an object from container A to container B, since for example, starting with a red object at A and applying both "change to blue" and "move to B" yields a blue object at B, regardless of the order in which the operations are applied. But, once views or read-only transactions or system state in nonquiescent conditions is taken into account, some sites might see a transition in which a blue object was at A and others a transition in which a red object was at B.

Finally, in ORESTE a state cannot be committed to an external view until it is known that there is no straggler; this involves a global sweep analogous to Jefferson's Global Virtual Time algorithm. As described in Section 5.1, this could be problem when there is a widely distributed network of independent collaborations.

A recent system, COAST [15], also attempts to use optimistic execution of transactions with the MVC paradigm for supporting groupware applications. Key differences with our system are the following. First, COAST only supports optimistic views. Second, concurrency algorithms used in COAST assume that all model objects in the application are shared among all participants. Furthermore, the optimistic algorithm implemented in COAST is based on a variation of the ORESTE algorithm discussed above. COAST cannot, therefore, be used in applications that require pessimistic views or require one application to share one set of model objects with one application and another set of model objects with another set of applications.

7 CONCLUSIONS

The DECAF framework's major objectives are ease of programming, and responsiveness in the context of systems of collaborating applications.

The ease of programming is achieved primarily through hiding all concerns about distribution, multi-threading, and concurrency from users. Programmers write at a high level, using the Model-View-Controller paradigm, and our implementation transparently converts operations on model objects to operations on distributed replicated model objects. The View Notification algorithm automatically schedules view snapshots at appropriate times and also allows views to respond efficiently to small changes to large objects. Model objects for standard data types (Integers, Strings, etc.) and collections (e.g., Vectors) are provided as part of the DECAF infrastructure.

The responsiveness results from the use of optimism combined with the fast commit protocol of the primary copy algorithm. If a transaction updates objects A and B, then a view of B', a replica of B, sees the commit as soon as A's primary site and B's primary site have each notified the originating site that the updates are nonconflicting, and the originating site has notified B''s site that the transaction has committed. This is a small delay, even for a pessimistic view. Users can get even more rapid response time using optimistic views, and most of the time their optimistic view will later be committed with the same speed as the pessimistic view.

Our experience with using DECAF has shown the architecture and algorithms to be well suited for a variety of groupware applications.

ACKNOWLEDGMENTS

We gratefully acknowledge Gary Anderson's input to the design of our framework. He has also built several collaborative applications and components on top of our framework.

REFERENCES

[1] H.M. Adbel-Wahab and M.A. Feit, "XTV: A Framework for Sharing X Window Clients in Remote Synchronous Collaboration," *Proc. IEEE Tricomm '91: Comm. for Distributed Applications and Systems*, Apr. 1991.

[2] D.F. Bacon and R.E. Strom, "Optimistic Parallelization of Communicating Sequential Processes," *Proc. Third ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, Apr. 1991.

[3] G. Banavar, S. Bhola, K. Miller, B. Mukherjee, and M. Ward, "The Tradeoff of Responsiveness versus Regularity in Optimistic Groupware," Forthcoming IBM T.J. Watson Research Center technical report, 1997.

[4] K. Birman, A. Schiper, and P. Stephenson, "LightWeight Causal and Atomic Group Multicast," *ACM Trans. Computer Systems*, vol. 9, no. 3, pp. 272-314, Aug. 1991.

[5] W. Chu and J. Hellerstein, "The Exclusive-Writer Approach to Updating Replicated Files in Distributed Processing Systems," *IEEE Trans. Computers*, vol. 34, no. 6, pp. 489-500, June 1985.

[6] C. Cowan and H.L. Lutfiyya, "A Wait-Free Algorithm for Optimistic Programming: Hope Realized," *Proc. Int'l Conf. Distributed Computing Systems*, pp. 484-493, Piscataway, N.J., 1996.

[7] C. Ellis, S.J. Gibbs, and G. Rein, "Concurrency Control in Groupware Systems," *Proc. ACM SIGMOD '89 Conf. Management of Data*, pp. 399-407, 1989.

[8] D. Jefferson and A. Motro, "The Time Warp Mechanism for Database Concurrency Control," *Proc. Int'l Conf. Data Eng.*, pp. 474-481, Los Angeles, Feb. 1986.

[9] D.R. Jefferson, "Virtual Time," *ACM Trans. Programming Languages and Systems*, vol. 7, no. 3, pp. 404-425, July 1985.

[10] A. Karsenty and M. Beaudouin-Lafon, "An Algorithm for Distributed Groupware Applications," *Proc. Int'l Conf. Distributed Computing Systems*, 1993.

[11] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object Oriented Programming (JOOP)*, vol. 1, no. 3, pp. 26-49, Aug. 1988.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[13] M.A. Linton, P.R. Calder, and J.M. Vlissides, "InterViews: A C++ Graphical Interface Toolkit," *Proc. USENIX C++ Workshop*, p. 11, Santa Fe, N.M., Nov. 1987.

[14] A. Prakash and H.S. Shim, "DistView: Support for Building Efficient Collaborative Applications Using Replicated Objects," *Proc. Fifth ACM Conf. Computer-Supported Cooperative Work*, pp. 153-164, Oct. 1994.

[15] C. Schuckmann, L. Kirchner, J. Schummer, and J.M. Haake, "Designing Object-Oriented Synchronous Groupware with COAST," *Proc. Computer Supported Collaborative Work CSCW*, Boston, Nov. 1996.

[16] R.E. Strom and S.A Yemini, "Synthesizing Distributed and Parallel Programs Through Optimistic Transformations," *Current Advances in Distributed Computing and Communications*, Y. Yemini, ed., pp. 234-256. Rockville, Md.: Computer Science Press, 1987.

Robert Strom graduated from Harvard University in 1966, and completed his postgraduate studies at Washington University in 1971. Since 1977, he has been a research staff member at the IBM T.J. Watson Research Center. His main interests are in programming languages, distributed systems, and environments for distributed computing. His major research contributions include the Hermes distributed programming language, a secure, high-level, object-based distributed language; and Optimistic Recovery, a protocol for asynchronous recovery in distributed systems. His current area of research is enhancing and broadening the publish/subscribe paradigm.

Guruduth Banavar was awarded a PhD in computer science by the University of Utah in 1995 for his work in developing a new object-oriented class component technology and applying it in innovative ways to flexibly compose a variety of new and legacy components. He is a research staff member at the IBM T.J. Watson Research Center. His current research interests are in middleware and application frameworks for developing distributed applications.

Kevan Miller received a BS degree in computer science from Furman University in 1982 and an MS degree in computer science from Columbia University in 1995. He is an advisory software engineer at the IBM T.J. Watson Research Center. He joined IBM in 1982 in IBM's Networking Division. He has worked at the T.J. Watson Research Center since 1990. His research interests include distributed computing, application frameworks, and software engineering.

Atul Prakash received his BTech degree in electrical engineering from the Indian Institute of Technology, New Delhi, and his MS and PhD degrees in computer science from the University of California at Berkeley. He is an associate professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include computer-supported cooperative work (CSCW), distributed systems, security, multimedia, and software engineering. He has served on several program committees, including the ACM CSCW Conferences, the European CSCW conferences, and the IEEE ICDCS conference. His work has been supported by both government and industry, including the U.S. National Science Foundation, the U.S. National Security Agency, NASA, Hitachi Software Engineering Ltd, IBM, and Bellcore. The work described in this paper was done while Dr. Prakash was on sabbatical leave to the IBM Watson Research Center and during subsequent consulting with IBM.

Michael Ward received a BS in mathematics in 1973 from the University of Illinois. He is a senior software engineer at the IBM T.J. Watson Research Center, where he manages the Collaborative Frameworks group. His current research interests are in middleware to support distributed applications on wide-area networks.