RC 21126 (94460) 11MAR98
Computer Science/Mathematics
8 pages

IBM Research Report

Flexible Batching and Consistency Mechanisms for Building Interactive Groupware Applications

Bodhi Mukherjee, Sri Doddapaneni

IBM Research Division
T.J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

Sumeer Bhola, Mustaque Ahamad

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Flexible Batching and Consistency Mechanisms for Building Interactive Groupware Applications

Sumeer Bhola* Bodhi Mukherjee† Sri Doddapaneni† Mustaque Ahamad*

Abstract

This paper presents our state sharing support for building object-oriented interactive groupware in wide-area distributed environments. We motivate and present an asynchronous model for updating replicated state, which supports atomicity of updates across multiple shared objects. Coupled with our flexible marshalling framework, this model allows existing application data-structure classes to be easily extended and made shareable. To solve the problem of replica consistency we use a novel combination of three mechanisms, (1) global locks, (2) detection of incorrect update ordering, and (3) cloning a subset of the shared objects and state reinitialization. Finally, to reduce network load due to fine-grain user interaction, we provide a framework for application specified event batching called Late Event Modification.

1 Introduction

With the ability to extend web browser functionality using Java, plugins, ActiveX etc., simple interactive groupware applications are being made available to a wide population of users. Examples include internet chat and simple whiteboards. However, for building complex object-oriented collaborative applications, developers need a simple and powerful programming model, which at the same time should allow good response time and efficient implementation in a wide-area distributed environment. Appropriate support for sharing state is critical for supporting collaborative applications. As we are dealing with wide-area environments, we use replication of state to improve response time. These high-level requirements for interactive groupware, coupled with replicated state, lead to three critical issues that we address in this paper.

1. *Programming model* : From a system design viewpoint we want to decouple the programming model from any concurrency control implementation (including our own). For example, when an application issues an operation on a shared object, the system should have some flexibility in scheduling this operation at the various replicas. This separation between the issuing of an operation and its execution leads to an asynchronous programming model that allows the system to employ a variety of concurrency control implementations such as pessimistic locking, ordering actions via a server, optimistic with automatic rollback and others.

   From the application developers viewpoint we consider the following requirements:

   - Pipelining: It is desirable that an application be able to continue processing user input and initiate actions while its previous actions are being executed. This is useful, for example, when the user input is fine-grain, and a small lag in response is acceptable, like typing a few characters ahead of the echo. Depending on the lookahead permissible to the user, and the actual response time, this can allow for faster user interaction. The asynchronous model motivated by the system needs also meets this requirement.
   - Atomicity: As single-user applications have no source of contention, there is no need to distinguish between a group of operations which are part of a single action and a series of actions. However, with multiple users, it is necessary to specify some group of operations as indivisible so that they are scheduled for execution at the same time, and to ensure that other operations are not interleaved with them. Hence, the programming model should support atomic actions that access multiple shared objects.
   - Support for legacy applications: It should be easy to convert existing single-user application data-structures into shared data using class extensions.

   We provide an asynchronous model for specifying atomic operations on the shared state using *update events*. This model only requires shared objects and events to implement a simple `Marshallable` interface. This interface allows us to provide powerful state sharing semantics which are *not possible* to implement with simpler interfaces like Java Object Serialization.

2. *Consistency of shared state* : Potential inconsistencies in the replicas can arise due to different ordering of the events at different processes. Most systems take two extreme approaches when dealing with consistency of replicated state, (1) a fully optimistic approach with rollbacks and reexecution, and (2) a pessimistic approach utilizing locking. The first approach frees the application pro-

*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332.

S. Bhola was supported by an IBM fellowship.

†IBM T.J. Watson Research Center, Yorktown Heights, NY 10598.

grammer from the burden of consistency maintenance, but the effect of jitter in the user interface due to automatic rollback could be a problem. The second approach has the problem of not allowing enough freedom of interaction. It has been observed that strict locking is usually not necessary for collaborative applications because of implicit social protocols [5] employed by collaborating users.

We use an intermediate approach which uses optimistic execution along with a combination of three mechanisms which progressively expose the application writer to more of the distributed nature of the application.

(a) Global Locks: This can be used by the application to enforce correct ordering.

(b) Detection of Conflicts: The system detects conflicts due to incorrectly ordered updates and informs the application.

(c) Cloning and Reinitialization: The application can use this to construct an application specific conflict resolution policy.

3. *Application specific event batching*: Fine-grained user interaction with a GUI leads to a lot of updates on the shared state. For efficiency reasons it is not good to propagate all these updates to the other users in the collaboration. We use a novel event batching technique (Late Event Modification) to resolve this tension between interactivity and performance.

This work was done in the context of the VILLA project whose goal is to provide a framework for building complex object-oriented interactive groupware. The current prototype is implemented in Java and provides state sharing, event sharing and real–time audio and video support (using the Bamba [1] system). The authentication and access control framework can support implementation of electronic commerce applications like collaborative banking. It provides high–level collaboration establishment metaphors like Rooms, and can run both inside a web browser (as an applet) and as a stand–alone application. We have used it to build collaborative applications from scratch or have modified existing applications to make them collaborative. Examples include a collaborative drawing tool, an auctioning application, a collaborative IDE for Java, and a presentation tool for internet conferences and remote teaching.

We review related work in Section 2 and Section 3 describes the programming model. Section 4 discusses our support for maintaining consistency of state and Section 5 describes the application specific batching framework. Section 6 puts the earlier sections in perspective by discussing an application. Section 7 briefly discusses the implementation and provides some experimental evaluation. We conclude in Section 8.

2 Related Work

In this section we consider state sharing support provided by general distributed systems, and systems targeted for building interactive groupware. The two facets of such systems that are important to us are (1) the programming model, and (2) the algorithms used for state consistency or message ordering.

Java RMI [2] provides transparent invocation of remote objects using local proxies. This is similar to the CORBA [6] invocation model. Though the basic RMI and CORBA specifications do not deal with replication, they can be extended to deal with replicated objects in a transparent manner [8]. The advantage of such an approach is that it hides most of the complexity of dealing with replicated objects, by preserving the synchronous invocation mechanism used for most applications. However it does not permit changes in the concurrency control policy without effecting the invocation response time, and it does not allow atomic updates across a group of objects.

DistView [10] provides support for object replication while permitting some objects to choose not to be replicated. Each replica of a shared object has a proxy object which is used to capture method invocations. Shared objects can have references to other shared objects through these proxies. An invocation on a shared object can invoke other shared objects, and consistency is maintained using pessimistic locking.

Grove [4], ORESTE [7], COAST [11] and DECAF [12] use optimistic concurrency control with system guaranteed state consistency. The ORESTE algorithm considers a shared document consisting of a set of objects, which are fully replicated at all the participating processes. Each event modifies exactly one shared object. Locally generated timestamps are used to order all the events, and information about commutativity and masking of events is used to minimize rollback. The Grove text editor uses similar ideas, but they use transformation of operations instead of undo, to fix state inconsistency.

Both COAST and DECAF provide an asynchronous update model using transactions, with rollback and reexecution to guarantee state consistency. In COAST, the shared objects have to be subclassed from the abstract COAST replicated object class. In an object-oriented language with only single inheritance (like Java), this prevents existing data-structures to be extended and made shared. DECAF allows partial replication of the shared set of objects. They also provide optimistic and pessimistic views, which is an interesting mechanism to expose or hide the underlying optimism from the end user. However, all applications have to be built using the shared primitives and aggregates provided by their framework. Our consistency and event batching mechanisms are complementary to their approach, and we feel that it provides semantics that are very relevant for collaborative applications. We also permit shared objects to have pointers to other shared objects, and dynamic addition and deletion of these objects.

3 Programming Model

In this section we discuss our programming model. A *set* abstraction is used to represent a collection of related shared objects. Objects in a set can have pointers to other objects in the same set, which can be cyclic. Applications in a collaboration can dynamically join

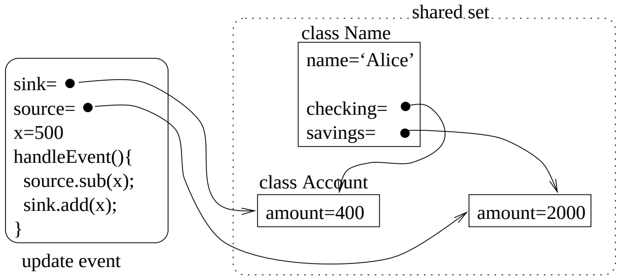

Figure 1: Updating the Shared Set

or leave a set. All the objects in a set are fully replicated at all the applications that have joined that set of objects. The three main operations on a set are

- `addObject(Object o)` Adds o to the set where o can have pointers to the objects already in the set.
- `deleteObject(Object o)` Delete object o from the set.
- `updateEvent(EventHandler e)` This method is used to issue an update to the objects in the set. The update event e encapsulates the actual update, and can atomically read and write a subset of those objects. The `handleEvent()` method of e, when executed, does the actual update. Figure 1 shows an event object with its associated `handleEvent` method to update two of the three shared objects in a set.

The `addObject` method adds the object to the local set replica, sends a message with the new object state to other processes[1] and then returns. In contrast, the other two are asynchronous, i.e. they are executed locally and remotely at some time in the future.

3.1 Marshalling

To propagate the state of the newly added objects or the update events to other processes in the collaboration, we need to marshal the state. Most systems assume that an object knows how to marshal itself, and therefore expect it to implement a simple marshal method to which they pass an output stream as a parameter. However, in our system the marshalling of an object is very context dependent. For example, let A be an object being added to the set, which has a pointer to B, which is an object already in the set. A needs to be marshalled, as its state has to be sent to the other replicas. But when marshalling A, B is not marshalled, and a machine independent handle is substituted for it. However, when a new process joins the set, the state of both A and B have to be marshalled, so that the newcomer can be initialized. More complex scenarios arise when we discuss object cloning. However, the basic inference is that the system needs to provide a context for the marshalling, which it does by using a `MarshalHelper` object. The `Marshallable` interface, implemented by the objects, is given below.

[1]Each application in the collaboration runs on top of our system ‘process’.

```
public interface Marshallable {
  /* out: output stream to write the state */
  public void marshal(DataOutput out, MarshalHelper mh);
  /* in: input stream to read the state */
  public void unmarshal(DataInput in, MarshalHelper mh);
}
```

For example, the class `Name` from figure 1 would implement `Marshallable` as follows

```
public void marshal(DataOutput out, MarshalHelper mh) {
  out.writeString(name);
  mh.genericMarshal(out, checking);
  mh.genericMarshal(out, savings);
}
public void unmarshal(DataInput in, MarshalHelper mh) {
  name = in.readString();
  checking = (Account) mh.genericUnmarshal(in);
  savings = (Account) mh.genericUnmarshal(in);
}
```

Thus, a call to `MarshalHelper` is interposed whenever marshalling crosses object boundaries in the object graph. This allows the `MarshalHelper` to do something else instead of, or before calling, the marshal method of an object.

4 State Consistency

This section discusses our three level approach to state consistency. Firstly, we provide global locks, in which the lock release is ordered after the events issued while the lock was held. This can be used to guarantee correct ordering of events at all processes.

Secondly, for applications in which locking is not always desirable, the system also does conflict detection.

4.1 Conflict Detection

The conflict detection algorithm depends on the architecture of the system, therefore we will assume the architecture of our current prototype. The processes are logically arranged in a star configuration with a server in the center. All events flow through the server, and the order in which the server receives the events is the ‘correct’ order. Events originating at a process are immediately executed locally and also sent to the server. This optimistic execution can result in inconsistency. The server distributes the events to all the processes, including the source, using FIFO communication channels.

Let $X = O_1, O_2, \ldots, O_m$ be the current set of shared objects. An event e issued on this set has a read–set $e.R \subseteq X$ and a write–set $e.W \subseteq X$ [2]. Each process P maintains an `unstable` queue of events. These are the events which originated at P and have been executed locally, but have not yet been reflected back by the server. Let a remotely originating event e'' be received at P. The possible reasons for conflict are.

1. write $\rightarrow$ write : e'' writes an object which is also written by an event in the unstable queue.
2. read $\rightarrow$ write : e'' reads an object which is written by an event in the unstable queue.

[2]This information is explicitly provided by the application, but there is some potential for automatically inferring it.

```
remoteEventReceived(e) {
   S_err = S_read = ∅
   ∀ o ∈ e.W
      if (o.numOfW > 0) // write->write
         S_err = S_err+ o // objects in error
      if (o.numOfR > 0) // write->read
         //events reading this set are conflicting
         S_read = S_read+o
   ∀ o ∈ e.R
      if (o.numOfW > 0) or (o.inError)
         // writes could depend on an incorrect read
         S_err = S_err+ e.W // read->write,error->read
   while (unstable.notEmpty)
      e' = unstable.next()
      q.add(e')
      if (e'.R ∩S_err ≠ ∅) or (e'.R ∩S_read ≠ ∅)
         S_err = S_err+ e'.W
   unstable = q
   ∀ o ∈ S_err, o.inError = true
   error = error +S_err
}

localEventReceived(e) {
   unstable.add(e)
   if (error ∩ e.R ≠ ∅)
      error = error + e.W
      ∀ o ∈ e.W, o.inError = true
   ∀ o ∈ e.R, o.numOfR++
   ∀ o ∈ e.W, o.numOfW++
}

localEventReflected(e) {
   unstable.next() // remove from queue
   ∀ o ∈ e.R, o.numOfR--
   ∀ o ∈ e.W, o.numOfW--
}
```

Figure 2: Detection of Inconsistent State

3. write → read : e'' writes an object which is read by an event in the unstable queue.

Figure 2 gives the simplified algorithms for maintaining the set of objects whose state is in error at P. These objects are in the `error` set. For each object `o` the fields `o.numOfR` and `o.numOfW` give a count of the number of times `o` is read and written by events in the `unstable` queue. `localEventReceived` is the method called when an event is generated locally and `localEventReflected` is called when such an event comes back from the server. When a remote event is received, the algorithm uses the read and write set information of that event and the events in the unstable queue (which should have been ordered later), to detect errors.

And finally, to help the application fix these errors automatically or with the help of user interaction, we provide a cloning and reinitialization mechanism.

4.2 Cloning and Reinitialization

The cloning mechanism allows an application to make a copy of a *subset of objects* in the set. Pointers from inside the original subset to objects outside are preserved in the cloned subset. Pointers between objects inside the original subset are mapped into the cloned subset. Figure 3 shows the effect of cloning a subset of objects in the set. A `CloneRelation` object maintains the original (parent) to clone mapping. The CloneRelation object and the cloned subset can be added to the set of objects which provides a mechanism to export incorrect state to other clients, so that different versions of the same object can be compared. It can then be used to reinitialize the state of the originals from the clone.

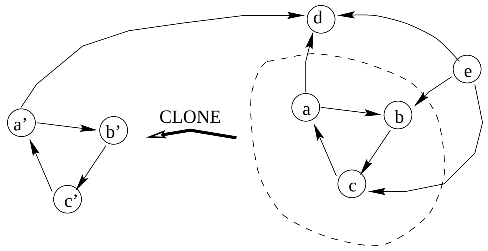

Figure 3: Cloning a Subset

Figure 4 depicts how these mechanisms can be used to fix diverging state. The system detects that the replicas of B at client 1 and 2 are different and informs the application. The applications clone both A and B, as meaningfully interpreting B alone is not possible. These clones are exported to each other by adding them to the set of shared objects. Now that the clients have local copies of the differing states, they can do a comparison and fix one of these to represent the correct state. In the example, they fix B''. B'' is then used to reinitialize the originals by client 1.

This cloning of some objects and then reinitialization of a subset of those objects is also made possible by our flexible marshalling interface. When reinitializing a subset of the objects in the clone relationship, the subset is marshalled in the context of the clone, but unmarshalled in the context of the original. The clone relation can also be nested i.e. clones can be made of clones, and then reinitialization can be done by following the path back to the parent.

The reinitialization is done using a special type of update event, called the reinitialization event. This event does not read any objects, but reinitializes the state of all the objects in its write set. Figure 5 shows how this event is processed by the conflict detection subsystem. The processing for a local reinitialization event reflected back from the server remains the same.

5 Event Batching

In our programming model, shared objects are modified only through update events which are issued using the `updateEvent` method. The system executes these events locally as well as multicasts them to all the processes in the set, which execute them to bring about the same state change. To get good response time for the issuer's actions, these events should be fine–grained. For example, in a text editor every key–stroke may have to be packaged as an event so that

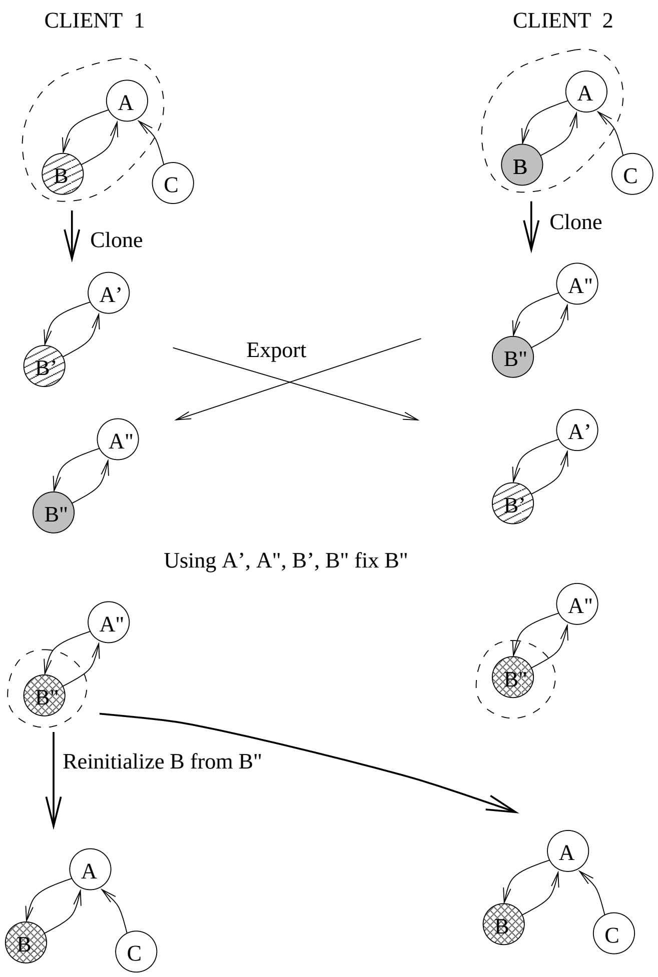

Figure 4: Fixing Incorrect State

```
remoteReinitEventReceived(e) {
     ∀ o ∈ e.W
          o.inError = false
          error = error - o
     remoteEventReceived(e)
}

localReinitEventReceived(e) {
     ∀ o ∈ e.W
          o.inError = false
          error = error - o
     localEventReceived(e)
}
```

Figure 5: Handling the Reinitialization Event

the issuer (say Alice) sees the result of her actions immediately. But in most cases it is not necessary for other users to see the key–stroke generated by Alice immediately. In fact, Alice may not want them to see her modifications until they are in a presentable form.

The simplest solution to this problem is to collect several events and package them together in a message before doing the multicast. However the space overhead of such a solution is going to be high for the following reasons.

1. If Alice is in a very experimental mode, there might be a lot of undoing of previous actions which could allow for elimination or compression of certain events. However, as the system does not know this, it cannot do so.

2. Each event is packaged with some system information before being sent to other processes. When events are simply batched, a lot of overlapping system information cannot be eliminated.

Our solution, Late Event Modification (LEM) enhances the event interface to allow applications to modify the event objects after posting them to the set. However, these modifications can only be made before the event is marshalled and sent out to other processes. For the purposes of explaining this application specific batching scheme, we will assume the following about the client system implementation: There is a SharedEvent Thread which maintains an event `queue` for events waiting to be executed, and executes their `handleEvent` methods in order.

5.1 LEM Architecture

To support LEM, we provide an abstract event class, `OpenEvent`, which is extended by the application to implement the required event functionality. Figure 6 shows the `OpenEvent` class with the dashed box showing the abstract methods implemented by the application, and the monitor object which maintains the current `state` of the event. The abstract methods (the `RealEvent` interface) are:

- `handleSetEvent()` : The actual event handler which is executed by the SharedEvent Thread when the event reaches the front of the event `queue`.

- `updateBeforeExec(Object arg)` : This handles any updates to the event object before `handleSetEvent` is executed. Typically, this will just update the state of the event object.

- `updateAfterExec(Object arg)` : This handles any updates after the `handleSetEvent` has been executed. This has to update the state of the event and the relevant shared objects.

A locally generated event goes through three states after the application posts it to the set and before it is marshalled and sent out to the server.

1. BEFORE: The `handleSetEvent` method has not yet been executed by the SharedEvent Thread.

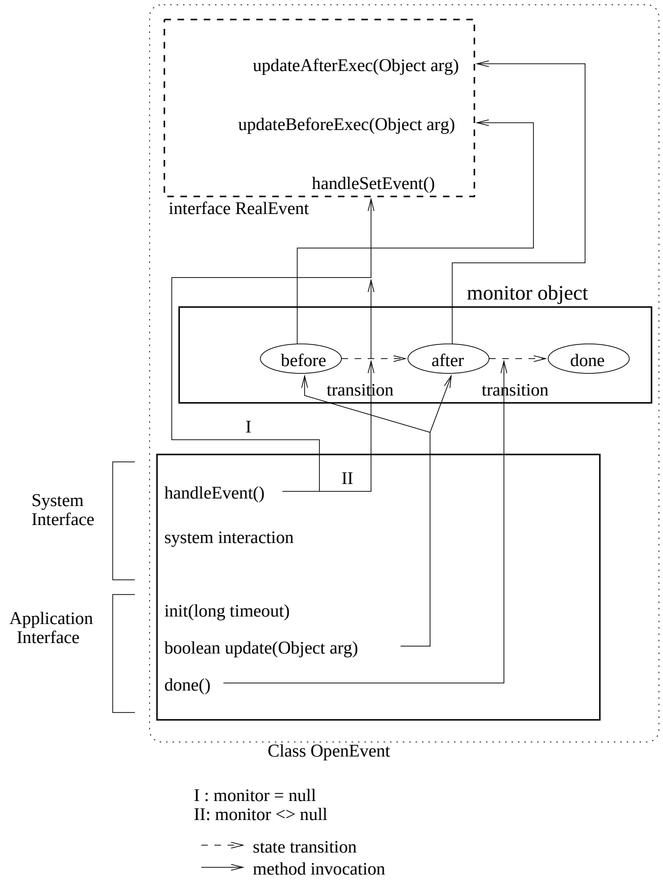

Figure 6: The OpenEvent abstract Event Class

2. AFTER: The `handleSetEvent` method has been executed.

3. DONE: The event has been marshalled. No more updates to the event are permitted.

To use these concrete event classes, the application creates one of these event objects and issues it using the normal `updateEvent` method. It can then use the `update` method implemented by the abstract class to modify the event object. This results in a call to `updateBeforeExec` if the event state is BEFORE and `updateAfterExec` if the state is AFTER. If the state is DONE, neither of the two is called and a value of *false* is returned.

This class provides two mechanisms to control the transition to the DONE state (1) explicit, by invoking the `done` method, and (2) implicit, by using a `timeout` value. The `timeout` value specifies the maximum number of milliseconds after the execution of the `handleSetEvent` that the event can stay in the AFTER state. Note, that this does not prevent the application from explicitly calling the `done` method. Also note that the `done` method can be called explicitly by the application thread or can be used inside the `RealEvent` methods. For example, consider a text editor where every keystroke has to be echoed immediately. A keystroke will result in posting of a new event or updating the previous one. Suppose we want a transition to the DONE state whenever the user presses the enter key. This could easily be coded by making the `updateBeforeExec` and `updateAfterExec` methods check if `arg = enter` and then calling the `done` method.

6 Drawing Board Application

In this section we discuss one of the applications implemented using our system, to demonstrate the utility of our mechanisms. It is a modification of the DrawTest example applet provided with the Java Development Kit.

This application allows the users to share a drawing board on which they can draw and edit lines and curves. The lines and curves are drawn by pressing down on the left-mouse button, dragging it to the end point (or along a curve), and releasing the button. A shared object is maintained for each line or curve, where the line is represented as two points and the curve as a list of points. The editing operations permitted on objects are translation, and changing the end-points (only for lines).

The application uses the conflict detection support provided by Villa to detect concurrent updates on the same object. It then allows the user to clone and export the complete set of objects to others. After deciding on the correct state, reinitialization of the complete set is done.

In addition, it uses Late Event Modification to give the user feedback while drawing or translating a line or curve. We explain this by considering the case of Alice drawing a line. The initial mouse press fixes the initial point of the line and adds the line object to the set. Though the final point is not fixed till Alice releases the button, she wants to see the current position of the line while dragging the mouse. However, others do not need to see each of these mouse drag events. LEM allows us to solve this problem by posting an `updateEvent` with the current final point and then continuously modifying the final point value in the event object as Alice drags the mouse. The `done` method is called on the event when the mouse button is released.

This application can be enhanced in two ways. Firstly, instead of cloning the complete set, it could clone the object in error and the objects in its physical neighborhood. After users fix the errors, reinitialization is done of the single object in error. Secondly, it can use the atomicity property of update events, to translate a group of objects.

7 Implementation

Our current implementation [9] uses a central server (for a given collaborative session), and client processes logically arranged in a star configuration. Both the server and the clients are implemented in Java. Each collaboration is managed by a *VillaActivity* object at the server, which can be configured to utilize different communication protocols and provide different levels of authentication. The default configuration uses TCP for communication. The basic system provides a group

	LAN	WAN
Set Create	20	130
Set Join	65	162

Table 1: Set Manipulation Operations (time in ms)

Size of Event (bytes)	Async Opt (ms)	Sync Opt (ms)	Sync Pess (ms)
10	4.41	4.40	6.27
1000	4.44	4.52	6.99
100000	30.1	30.1	55.7

Table 2: Time for `updateEvent` (LAN)

abstraction, which is used to build the state and event sharing support described in this paper.

The system is fault-tolerant to client crashes, and the server does periodic checkpointing so that the shared state can be recovered if the server fails. Also, as most collaborations happen between a small group of people, scalability is not a major issue. By using replicated servers and reliable group communication algorithms utilizing virtual synchrony [3], fault-tolerance can be provided.

7.1 Evaluation

This section provides some measurements of basic system costs and compares the cost of choosing certain options. We used two experimental platforms, both with one client and one server. The local area setup uses two SUN ULTRA 1 machines connected by fast Ethernet. The wide area setup uses a SUN ULTRA 1 as the client and a SUN SPARCstation 20 as the server, connected via the Internet. In this setup, the physical distance between the two machines was about 6 miles.

Table 1 shows the time to create and join a set of shared objects. Note that the wide-area times are appreciably larger, but as these operations are performed only on startup, we have not made any attempt to optimize them.

Table 2 shows the time to issue an `updateEvent` in the local area setup. In 'Async Opt' the application does not wait for the previous update to actually execute before issuing the next one, while in 'Sync Opt' it does. In both these cases, the event is directly inserted into the local event queue and then sent to the server. In 'Sync Pess' the application waits for the event to execute before issuing the next one, and the event is only executed locally after it is ordered by the server. As expected, ordering through the server is more expensive than the optimistic case. However, we note that the 'Async Opt' and 'Sync Opt' times are effectively the same.

To explain this, Figure 7 looks at the relative costs of various operations done during an `updateEvent` invocation in the optimistic case.

After doing some 'local processing', which includes packaging the event as a message and conflict detec-

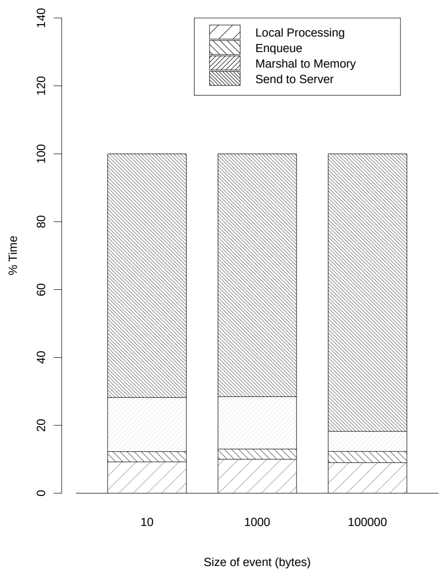

Figure 7: Cost of `updateEvent`

tion, the event is 'enqueued' in the local event queue. This enqueuing includes acquiring a lock on the event queue, and accounts for only about 3% of the total time spent in `updateEvent`. This coupled with a low thread context switch time of $35\mu s$, means that the asynchronous execution of the event in a different thread is a negligible factor in performance. A significantly higher percentage is spent in marshalling the message into a memory buffer and then copying it out to the server.

We now study the effects of using Late Event Modification. We consider two scenarios from our Drawing Board application. When drawing a line, the mouse-down fixes the first point, however the second point is not fixed till the mouse is released. Instead of issuing many update events which update the current second point, we use LEM to update the event in place. Similarly when drawing a curve we use LEM to append more points at the end of the event. To understand the quantitative gain, we did a study of the number of mouse drag events generated when dragging the mouse 1 centimeter on the screen. Depending on the speed of drag, it varied between 5 and 35. We assume an average of 20 mouse drag events for the following discussion.

The percentage reduction in number of events in both drawing a line or curve, and the saving in space for drawing a line is equal and shown in figure 8. The percentage reduction in total event space for drawing a curve is shown in figure 9 with different values of y, where y is the number of bytes of system overhead per event. The value of y in our current unoptimized prototype is about 120, and we expect that we can reduce it to about 60 without sacrificing the maintainability of the system code. Even in such a case LEM leads to about an 80% reduction in space for a 1cm curve.

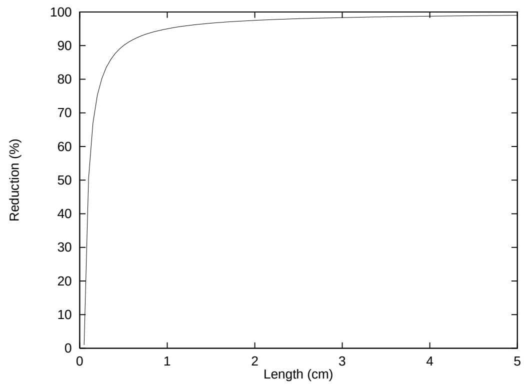

Figure 8: Reduction in number of Events (both), Event space(line)

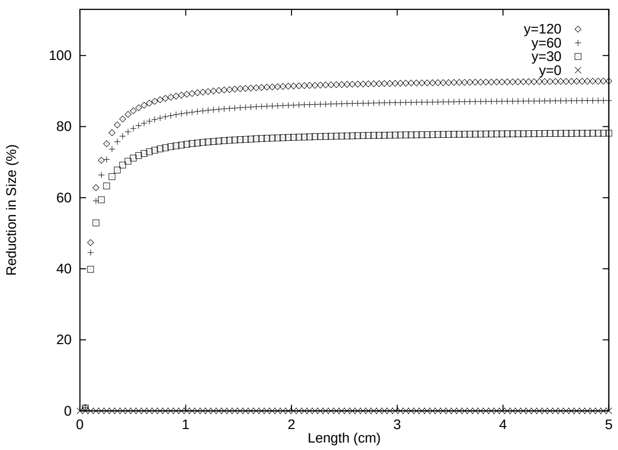

Figure 9: Reduction in Event space (curve)

These experiments show that the performance overhead of asynchronous execution in a separate thread is negligible, while allowing the flexibility of changing the concurrency control policy. Late Event Modification is shown to have reduced the number of events by over 90%, and the total event space by over 90% in one case and by over 80% in the other.

8 Conclusions

We have presented an asynchronous programming model which is decoupled from any concurrency control implementation. We believe that adoption of such a model provides sufficient flexibility to both the application programmer and the system implementor. Events and shared objects just have to implement our `Marshallable` interface, which can be achieved by extending existing application classes. Our experience shows that this interface is trivial to implement. However, defining the update event objects can be sometimes cumbersome. For this purpose, we propose to provide automatically generated proxies, which do this packaging for the common case in which the event invokes a method on a single object. We believe that our consistency mechanisms are a good middle-ground between fully optimistic and pessimistic solutions. Conflict detection is very cheap to do, and cloning and reinitialization has proven to be a very effective way to involve the end-user in fixing the state. Our event batching framework, Late Event Modification, is efficient because it uses application semantics, and at the same time doesn't impact application readability or maintainability because it is very high-level. In fact, we have found it to be the most useful feature of our system.

References

[1] *Bamba.* www.alphaworks.ibm.com/Bamba.

[2] *Java Remote Method Invocation.* java.sun.com/products/jdk/rmi/index.html.

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD'89*, pages 399–407, 1989.

[5] Saul Greenberg and David Marwood. Real time groupware as a distributed system: Concurrency control and its effect on the interface. Technical Report 94/534/03, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada, 1994.

[6] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* 2.0 ed., 1995.

[7] Alain Karsenty and Michel Beaudouin-Lafon. An algorithm for distributed groupware applications. In *Proceedings of ICDCS'93*, pages 195–202, 1993.

[8] R. Kordale, M. Ahamad, and M. Devarakonda. Object caching in a corba compliant system. *USENIX Computing Systems Journal*, 9(4), 1996.

[9] Bodhi Mukherjee, Sumeer Bhola, and Sri Doddapaneni. Villa: An event based middleware for real-time collaboration on the internet. In *Proceedings of the Fourth IEEE International Workshop on Community Networking*, 1997.

[10] Atul Prakash and Hyong Sop Shim. Distview: Support for building efficient collaborative applications using replicated objects. In *ACM CSCW'94*, pages 153–164, 1994.

[11] Christian Schuckmann, Lutz Kirchner, Jan Schummer, and Jorg M. Haake. Designing object-oriented synchronous groupware with COAST. In *ACM CSCW'96*, 1996.

[12] Robert Strom, Guruduth Banavar, Kevan Miller, Atul Prakash, and Michael Ward. Concurrency control and view notification algorithms for collaborative replicated objects. In *Proceedings of the 17th ICDCS*, 1997.