

An Arbitration Scheme for Concurrency Control in Distributed Groupware

Mihail Ionescu and Ivan Marsic

Center for Advanced Information Processing (CAIP)

Rutgers University

Piscataway, NJ 08854-8058 USA

+1 732 445 0549

{mihaii, marsic}@caip.rutgers.edu

ABSTRACT

We present a novel algorithm, called dARB, for solving the concurrency control problem in completely distributed collaborative applications. The main advantages of our approach are the simplicity of use and good responsiveness, as there are no lock mechanisms. The algorithm is application independent, which makes it suitable for general collaborative applications. The only constraint is that the applications use a tree as the internal data structure. The examples applications of the algorithm are implemented in the DISCIPLE collaboration framework. The applications are a group text editor and a collaborative 3D virtual environment called cWorld. We also discuss the use of awareness widgets to increase the efficiency of collaborative work.

Keywords

Groupware, distributed algorithms, concurrency control.

INTRODUCTION

Cooperative editing systems are multi-user systems where the actions of one user are instantaneously propagated to all the other participating users. An instance of a collaboration system is informally called a *session*. These systems allow geographically dispersed collaborators to edit text documents [8,13] or to draw on a shared whiteboard. The collaboration systems are characterized by the following specific requirements [13]: (1) *Good responsiveness*—the response to local user's actions should be rapid (ideally as in single user applications), (2) *Distributed and fault tolerant*—cooperating users may reside on different machines that may crash at any time and (3) *Unconstrained*—multiple users are allowed to concurrently and freely edit any part of the document at any time. These requirements are independent of the application semantics.

Building collaborative applications from scratch is a difficult and challenging task. We developed an application amework, called DISCIPLE [18], for building fault tolerant collaborative applications from single-user beans (JavaBeans-compliant components, applets, and applications). A Bean is a reusable software module capable of publishing or registering its interfaces under certain contexts [15]. Examples of components include a button in a graphical user interface, a small interest rate

calculator, an interface to a database manager, or a module for image processing.

One of the most significant problems in designing and implementing cooperative editing systems with replicated architecture is maintaining the consistency of replicated documents. A major task in consistency maintenance is concurrency control. The algorithm presented here attempts to fulfill all of the above requirements. We assume that the data structure of the application document is a *tree*. Operations are executed immediately at their originating site and locks are not required to access the data. However, as detailed in the paper, not all of the concurrent accesses are automatically solved. When the system cannot find a solution, the users are notified about possible inconsistencies, which they can solve manually using group awareness widgets.

The paper is structured as follows. The next section briefly reviews the DISCIPLE framework, focusing on event-based collaboration. Then we present the dARB algorithm for solving the concurrency control problem based on a specific example. Subsequently, we present a generalization of the algorithm to any type of application and give a sketch of the formal proof of correctness. Finally, we present related work and summarize our major findings.

DISCIPLE FRAMEWORK DESIGN

We define a collaboration session as a set of participant systems connected by a communication network. A participant system is called a *site* and the condition imposed is that there should be one site per user. Any site can and should communicate with any other site. A *configuration* is defined as a structure that holds the active sites, $\Gamma = \langle S_{\alpha 1}, \dots, S_{\alpha m} \rangle$.

Each site hosts an application that collaborates with the applications from remote sites. Each application implements a set O of operations: $Op_1, Op_2, \dots, Op_m$. A site sends events to the others in an operation-independent manner. When a site receives an event, it identifies and performs the operation(s) specified by the event.

We assume that the application state at each site is represented as an *application document* object. This is a feasible assumption for all generalized editors that deal

with structured data. Examples include a text document, a CAD object, a spreadsheet or a 3D virtual world. The application document is modified by the operations included in the set O . The configuration of sites is *correct* or *consistent* if for any two distinct sites α and $\beta \in \Gamma$ the following condition holds: If at any time the system were to become quiescent with no messages in transit between the sites, then their application documents must be the same $S_\alpha = S_\beta$. A more dynamic view of consistency is presented in [3], where the system is considered consistent if the communicating sites never observe contradictions in their states, which are detectable by comparing the contents of messages they exchange (p.337). A key issue is maintaining the correctness of the configuration in the presence of concurrent access.

We assume that the events transmitted in the network cannot be lost but the order of receiving is not guaranteed to be the same at all sites. This can lead to incorrectness since the operations defined by applications do not generally commute. Following the Lamport's approach [10] we define a partial ordering of all events in terms of local generation and execution sequences of the operations associated with the events.

Given the events e_α and e_β , generated at the sites α and β , then e_α *precedes* e_β if and only if: (i) $\alpha = \beta$ and e_α was generated before e_β or (ii) $\alpha \neq \beta$ and the execution of corresponding operation at the site β for e_α happened before the generation of e_β . We define the *precedence property*: if one event e_α precedes another e_β , then the execution of the corresponding operation at each site of the event e_α happens before the execution of e_β 's operation.

The precedence property does not guarantee correctness of the configuration. For example, consider a two-site system $\Gamma = \langle S_\alpha, S_\beta \rangle$ and assume that the applications on both sites are the same with the event orderings as shown in Figure 1. In the non-overlapping case, the event e_α precedes the event e_β and both applications execute the operations associated with the two events in the same order, resulting in convergence. However, when the operations overlap there is no precedence between e_α and e_β and the configuration will be incorrect because the corresponding operations do not commute. We say that two operations are *logically non-concurrent* if the following conditions hold: (i) the two operations overlap and (ii) the two operations commute, in the sense that the application document is the same regardless the order of executing the operations.

One possible solution to the correctness problem is to extend the partial ordering offered by the precedence property to a total ordering. Given a configuration Γ with each site sending its own events $e_{\alpha 1}, e_{\alpha 2}, \dots, e_{\alpha k}$, the *total ordering* exists if every site receives all the events in the same order. The only restriction is that the partial ordering is preserved. A more formal definition is given in [3]. Total ordering is the most general method of maintaining the configuration correctness. However, as demonstrated in [1],

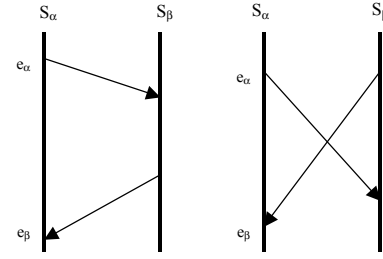


Figure 1: Non-overlapping and overlapping operations.

total ordering degrades the responsiveness of the system relative to partial ordering. The concurrency control presented here offers a more responsive alternative.

DISCIPLE also offers a totally ordered channel using a slightly modified logical clocks algorithm (Lamport alg.). We introduce the notion of event *priority*, in the sense that the events contain a tuple $\langle \text{siteID}, \text{priority}, \text{clock} \rangle$. When a site delivers a message it first orders the messages based on priority, followed by Lamport clock, and finally by the site ID. Each sending site computes the priority dynamically.

Dynamic Adapters

DISCIPLE is organized in two independent layers [18]: the communication layer, called *collaboration bus*, dealing with event exchange, dynamic joining and leaving, concurrency control and crash recovery and *the graphical user interface layer*, offering a standard user interface to every application bean imported into DISCIPLE.

Java Beans use *events* to communicate with one another. A Bean that wants to receive events (a listener Bean) registers with the Bean that fires the event (a source Bean). According to the JavaBean event model, any object can declare itself as a source of certain types of events. A source has to either follow standard design patterns when giving names to the methods or use the Bean Information class to declare itself a source of certain events. The source should provide methods to register and remove listeners for the declared events. Whenever an event for which an object has declared itself as a source is generated, the event is multicast to all the registered listeners. The source propagates the events to the listeners by invoking a method on the listeners and passing the corresponding event object.

DISCIPLE loads Beans at run-time and makes them collaborative by creating dynamic adapters for sending and receiving events. Event adapters are needed since a collaboration module cannot know the methods for arbitrary events that an application programmer may develop. Event adapters are equivalent to object proxies (stubs, skeletons), with the difference that the event adapters need to be registered as listeners of events so that the collaboration module is notified about the application's state changes. The process of event replication in DISCIPLE is illustrated in Figure 2.

A key feature of our framework is making Beans collaborative without the need to alter their source code to adapt them to the framework. DISCIPLE loads the Bean

and examines the manifest file in the Bean's JAR file for the information to automatically create the adapters. The adapters are generated with the code necessary to intercept the events, pass them to DISCIPLE to be multicast remotely and back locally, receive them after being multicast into the network, and pass them to the local bean. The code is then automatically compiled and the Bean's class path updated to contain the adapter classes.

The sites communicate with each other using a reliable multicast protocol [11]. A multicast session is uniquely defined by a multicast IP address and a port number. There is no notion of server in DISCIPLE, so when a site joins, it simply joins the multicast group. This architecture is scalable and resilient to site failures.

DISCIPLE allows importing and sharing arbitrary Java Beans. A class of beans of particular interest here are data-centric beans that use XML markup language for information exchange. Our applications use XML to represent the application document. The XML parser parses the source XML document and generates a tree, the so-called DOM tree (Document Object Model). Therefore, the DOM tree arises naturally as the application document for a class of DISCIPLE beans. This is particularly important for advanced concurrency control techniques, since it offers a single data structure for application documents.

THE dARB ALGORITHM

Data Structures

Tree structure. Each site α maintains its application document in a tree structure, T_α . Each vertex has an associated unique ID, which is the same at all sites. The vertex also keeps the count of operations successfully applied to it, which is used to detect if the operations overlap or not. The count can be obtained using the method `count(vertexID)`. The vertices also maintain a set of application-specific properties, which define the vertex *state*. The three operations that apply on a tree are: create

vertex (Op_1), delete vertex (Op_2), and edit vertex properties (Op_3). Even though some vertices may reference other vertices to implement behaviors (as in spreadsheet cells), the behavior structure is external to the tree.

Events. Events are tuples of the form $e = \langle \alpha, d, p \rangle$, where α is the sending site's identifier, d is the information associated with the event (operation and parameters) and p is the associated path from the root of the tree to the referred vertex to which the operation will be applied. The priority (as described in the previous section) and the local clocks are included in d . In p there is also transmitted the operation count associated with the vertex. There are four methods for getting from an event e the site id, data, path to the accessed vertex id along with the accessed vertex, and the count associated with the vertex: `siteID(e)`, `data(e)`, `path(e)`, `count(e)`.

Algorithm

We now present a precise formulation of the concurrency algorithm executed at any site, say α .

Initialization:

$T_\alpha \leftarrow \text{empty}$

Local operation:

```

receive operation o from the user interface
if (o is a create operation)
    newID  $\leftarrow$  create a new ID
    create a new vertex with associated id
    p  $\leftarrow$  path from root to the new vertex
    execute the operation o
    parID  $\leftarrow$  id of the parent of the newly
        created vertex
    increment count(parID)
    create event e from id, p and o
    broadcast e to all the other sites
if (o is a delete operation)
    id  $\leftarrow$  the id of the accessed vertex
    p  $\leftarrow$  path from root to the vertex
    execute the operation o
    modify all the vertices in the tree
        depending on the deleted vertex
    parID  $\leftarrow$  id of the parent of the vertex
    increment count(parID)
    create event e from id, p and o
    broadcast e to all other sites
otherwise // o is modify property operation
    id  $\leftarrow$  the id of the accessed vertex
    p  $\leftarrow$  path from root to the vertex
    d  $\leftarrow$  information about the property
        changes of the vertex
    execute the operation o
    increment count(id)
    create event e from id, p, d and o
    broadcast e to all other sites
endif

```

Remote operation:

```

receive e =  $\langle \beta, o, p \rangle$  from remote
if (o is a create operation)
    parID  $\leftarrow$  id of the parent of the newly
        created vertex
    execute the operation o
    if (count(parID)  $\geq$  count(e))
        performArbitration()
    else

```

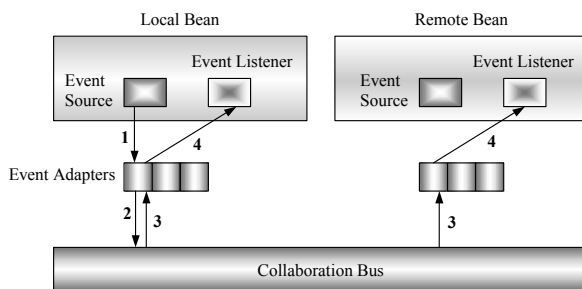


Figure 2: Event interception and symmetric distribution scheme in DISCIPLE: (1) The Event generated by the Event Source in the Local Bean, instead of being delivered directly to the local Event Listener, is intercepted by the associated Event Adapter and (2) sent to the Collaboration Bus. (3) The bus multicasts the event to all the shared Beans (remote and local). (4) Each Event Adapter receives the multicast event and delivers it to all listeners.

```

        increment count(parID)
    endif
    if (o is a delete operation)
        parID ← id of the parent of deleted
            vertex
        p ← path(e)
        execute the operation o
        modify all the vertices in the tree
            depending on the deleted vertex
        if (count(parID) ≥ count(e))
            performArbitration()
        else
            increment count(parID)
        endif
    otherwise // o is modify property operation
        id ← the id of the accessed vertex
        p ← path from root to the vertex
        d ← information about the property
            changes of the vertex
        execute the operation o
        newCount ← count(e)
        if (count(parID) ≥ count(e))
            performArbitration()
        else
            increment count(id)
        endif
    endif
end

```

The initialization part simply sets the tree to empty. The second section processes a locally generated operation. If the operation is to create a new vertex in the tree (e.g., in a text editor example, inserting a new-line character creates a new paragraph), the vertex is created with a unique id. To assure the uniqueness of the id at all sites, the vertex id is created in a recursive manner. Each site maintains a sequence number for each level on the tree. For the text editor, the site will maintain a sequence number for paragraphs, sentences and words. To create a new vertex id, we simply append the sequence number to the path in the tree. For example, for the site with the id=1, a paragraph can have the ids: 1.1, 1.2, etc., a sentence can have ids: 1.1.1, 1.1.2, etc.

When an operation is applied to a vertex, its operation count is incremented, just as in the Lamport's algorithm for obtaining the partial order. However, when the operations overlap and access the same vertex (e.g., several users are trying to concurrently write in the same word), the correctness of the configuration is no longer maintained. To solve this problem, we add an *arbitration phase* that attempts to automatically maintain the correctness of the configuration. When a conflict is detected the `performArbitration()` procedure is called.

In the case where operations do not overlap, partial order preserves the correctness of the configuration. If the operations overlap but access different vertices (e.g., inserting characters in different words), then the operations are logically non-concurrent, so the correctness of the configuration is also preserved by partial ordering alone.

When a site detects that two or more overlapping operations try to access the same vertex, it initiates the arbitration phase with the other involved sites by sending a

special event on the totally ordered channel as in Figure 3. The three sites detect at time t_α , t_β and t_γ that concurrent accesses have been made on the same vertex. Each site at this moment broadcasts a special message that prevents other sites from accessing this object while the arbitration is in progress (this message is not shown in Figure 3).

After that, the site sends on the totally ordered channel a special event m with a computed priority. All the sites receive these events in the same order (because they are sent on a totally ordered channel). The event sent with the greatest priority arrives first at every site. The site that receives this event and identifies that it is the source of the event *wins* the arbitration process. In our example, site β wins the arbitration, because the event m_β arrives first on the totally ordered channel. The winning site sends to the other sites an *update* message u (the thick lines in the figure) specifying the current state of the accessed vertex. Each site that receives this message updates its internal state and resumes the collaborative work.

The algorithm is general and applies to any application that has a tree for the application document data structure. The algorithm can be implemented as a separate module, and as long as the application provides the tree in a standard format, the same implementation can be used with different applications. Here we give two example applications.

Example 1: A Simple Text Editor

A basic text processing system might simply have a character string as its application document as in [8] and define two operations:

$Op_1 = \text{insert}(X;P) = \text{insert character } X \text{ at position } P$

$Op_2 = \text{delete}(P) = \text{delete the character at position } P$

In this case, any two operations that overlap (Figure 1) are also logically concurrent, so they conflict. In [8] a transformation based algorithm is proposed to maintain the correctness of the configuration. Although this is a powerful solution, it is hard to implement and there are counter examples to it [5]. Our approach of adopting a different data structure for the application document solves the problem in an easier manner.

The application document structure of our TextEditor is shown in Figure 4. The character strings in words represent a property of the word vertex. When a user types a character, an input module converts the low-level event into one of the tree operations:

Op_1 : A space character creates a new word vertex. A punctuation mark creates a new sentence vertex and a new word vertex. A new-line character creates a new paragraph vertex, a new sentence vertex, and a new word vertex.

Op_2 : Deleting the above characters deletes the respective vertices from the tree.

Op_3 : All other characters modify the property of the word vertex. The character type and the insertion/deletion

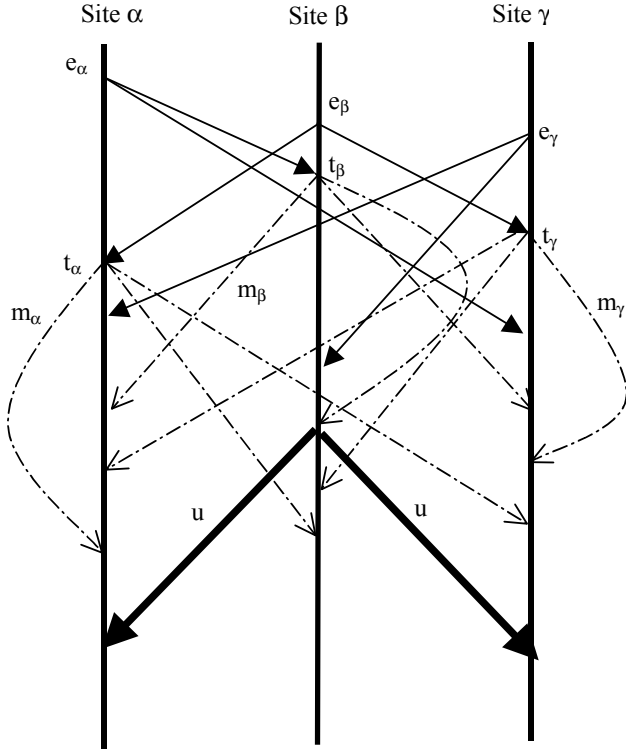


Figure 3: Arbitration phase for three sites. The concurrent events e are detected at times t , which triggers sending special prioritized events m to the ordered channel. The winning site sends the update messages u to the others.

position relative to the beginning of the word specify the property modification operation.

The operation is then processed locally and broadcast to the remote peers. The insertion or deletion of a single one character (e.g., a new-line) can significantly alter the tree topology. However the cost of manipulating the tree structure at each site is low compared to network latencies and can be neglected.

Example 2: Collaborative Virtual Environment

To demonstrate the generality of the algorithm, we also developed a 3D collaborative virtual environment, called cWorld, where the users can create environment objects dynamically and manipulate them, as shown in Figure 5. The operations on the application document's tree are:

Op_1 : creates simple geometric figures (box, cylinder, cone,

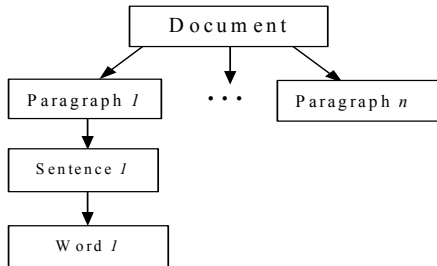


Figure 4: Tree structure of the application document for the TextEditor application.

sphere), lights (directional, point, spotlight), and furniture objects (desk, cabinet, bookcase), which are aggregate objects (non-leaf vertices). The room object is a non-leaf vertex as is an arrow telepointer.

Op_2 : deletes any of the above listed vertices

Op_3 : sets the vertex properties, such as color, texture, position, dimensions, visibility, and manipulation constraints (e.g., furniture objects are constrained from 'flying'). The lights have extra properties, such as direction, attenuation, spread angle. The user can also apply 3D affine transformations to the figures.

When a vertex is inserted in the tree, in cWorld the insertion vertex is at the root of the tree, whereas for the TextEditor the insertion can be made at any level. Nevertheless the algorithm performs very well on both types of applications, as detailed below.

Analysis

A key issue for correctness of the arbitration phase which maintains the correctness of the configuration is choosing the priority of the event that is sent to the ordered channel. To better understand this problem we present an example using the text editor.

Consider that initially all the sites have the same state, the string "First sentence. Second one". It is easy to imagine the associated tree. Suppose now that two users concurrently modify the word "first", one by adding 1 at the end and the other by deleting the first letter. After the arbitration, one of them wins and updates the state to all the other participating sites. In this case the priority does not matter because any site that wins the arbitration phase will correctly update the documents at the other sites.

However, suppose that the site α adds a space between 'r' and 's' (to make "fir st") and the site β adds a letter at the end of the word. The space is a special character and inserting it changes the topology of the tree at the site α by adding a new vertex for the word "st" and the former word "first" becomes now "fir". But if site β wins the arbitration, it sends its word "Firstl" to the first site, which updates the state to "Firstl st", so the correctness of the configuration is no longer maintained. Also, if site α wins the arbitration, it sends the word "fir" and again the correctness of the configuration is no longer maintained.

For this reason we assign different priorities to different operations. When a site generates a message to send to the totally ordered channel it also generates a priority based on the operation type. The operations create/delete vertex are assigned a greater priority than the operation that modifies the vertex properties (1).

Using the modified Lamport's algorithm we can be sure that the site with the largest priority wins the arbitration. If the priority assigned by the winning site is 1 (i.e., the operation modifies a property), the site sends only the state of that vertex, since all the other concurrent accesses were also modifying the properties. However, if the priority

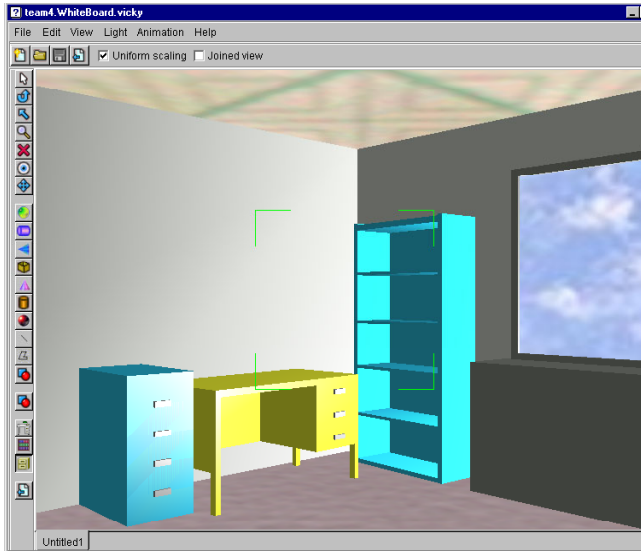


Figure 5: A sample CVE built using cWorld. The participants design a shared office space by creating and moving the furniture until they reach an agreement.

assigned by the winning site is greater, the site should send not only the state of the vertex itself but maybe also the state of the parent or the grandparent.

In the text editor example above, the site α will win the arbitration but will also send the state of the whole sentence to the others. If the special character had been a period or a question mark (usually punctuation marks), the site would have to send the state of the whole paragraph, or perhaps the entire document. Sending the entire document is not a desirable option because it can involve a large amount of data.

In our implementation, when the winning site realizes that it has to send the whole document it instead sends a special message, which informs all the sites that a concurrent access could not be resolved, and the users are responsible to solve the inconsistency, using the awareness widgets, before resuming work.

Although the algorithm does not automatically solve all the concurrent accesses, it still has very good performance as detailed below, since the likelihood of concurrent access is very low. Usually, the number of concurrent accesses to the same word is very small and majority of these accesses are automatically solved.

During testing of the text editor, we noticed, however, that the algorithm fails to maintain the correctness of the configuration in some special cases. Suppose that two sites have the string “Good student” as the same initial application document. Suppose also that the first user wants to delete the space character (which is equivalent to deleting a vertex) and the second user wants to add an ‘s’ to the second word, and that the two operations overlap. After executing each of the local operations, the first user will have the string “Goodstudent” and the second “Good students”. When the delete operation arrives at the site β ,

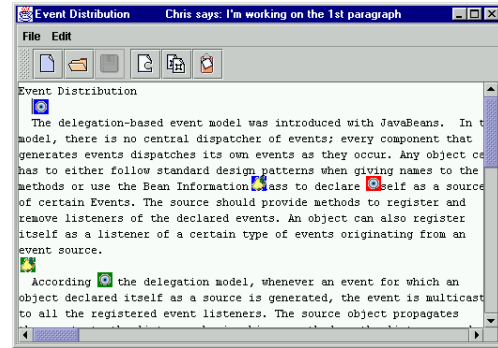


Figure 6: The text editor application in the DISCIPLE framework. The user sees his/her own and remote users’ cursors. If two cursors overlap it means that those two users work in the same area of the application, and, therefore, they may perform concurrent actions.

the site changes its application document to “Goodstudents”. But when the insert operation arrives at site α , the vertex it was referring to (the word “student”) does not exist anymore, so the operation cannot be executed. To solve this problem, when a delete operation occurs, the vertex is not actually deleted from the tree structure, but only marked for deletion. To reduce the dimensionality of the tree, we implemented a garbage collector that regularly scans the tree and deletes the marked vertices if they were not used for a certain amount of time.

Inconsistency Resolution by Awareness Widgets

We analyze here the case where two users concurrently modify the same vertex. Assume that both sites have the same initial application document, the string “abcd”. If the first user makes an `insert(1, "x")` and the second performs an `insert(1, "y")`, the first user will after that see “xabcd” and the second will see “yabcd”. After the `performArbitration()` procedure is executed, and the first site won, the second user will have the state “xabcd”. This situation can be very confusing for the second user, for he will not understand what happened. In order to make the user aware of other users actions, we developed in DISCIPLE several types of group-awareness widgets [7]. Telepointers are widgets that allow a given user to track remote users’ cursors. Figure 6 shows the text editor bean imported in DISCIPLE with telepointers. In addition, the users can exchange messages, post small notes, and annotate regions of the bean window.

PERFORMANCE EVALUATION

In order to evaluate the performance of our algorithm, we have to select performance metrics. As argued also in [1], we believe that the main parameter for evaluating a concurrency control algorithm is the User Latency, which is defined as the time interval between a user’s input action and its response seen by the same user.

Our test configuration is composed of up to six 266 MHz Pentium-based workstations connected in a 100 Mb/s LAN.

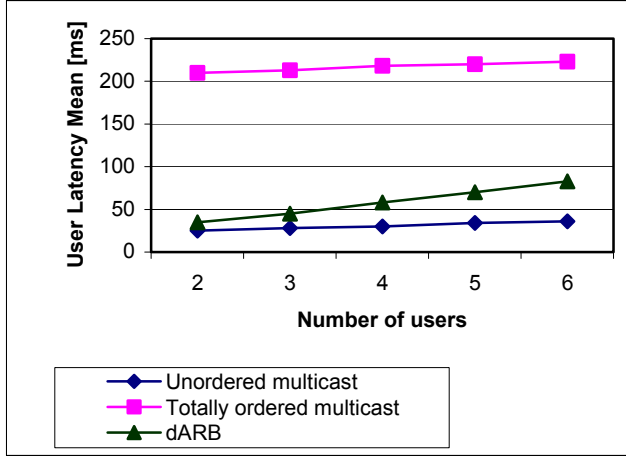


Figure 8: User latency vs. number of users for the TextEditor.

Wide-area network latency was simulated in software with the possibility of varying the latency dynamically. The users are given a task to write a group term paper using Texteditor and to create and arrange office furniture using cWorld.

Figure 8 shows the response time for the TextEditor application for a constant (and small, about 100 ms) latency by varying the number of users involved in the collaborative session. As expected, for the unordered and totally ordered multicast, the performance scales very well with the number of users. The performance of the dARB algorithm also scales well with the number of users, but the overhead introduced increases because the probability of concurrent accesses to the same object increases.

With these results in mind, we can now focus on studying the variance of user latency versus the latency in the system and fixing the number of users to a constant value, 4 in our experiments. Figure 9 shows the user latency for the TextEditor application varying the latency between the sites. For simplicity we consider that the latency is the same between all the sites at a given time.

As it can be seen, the performances of dARB for TextEditor are very good, close to the performances of the unordered reliable multicast.

We should also add that, during our experiments the percentage of concurrently accessing the same word was very low (under 5% of the total operations), and the algorithm solved a great majority of them automatically (over 80%). Because the bandwidth in the network was sufficiently large, the fact that during some arbitrations the data sent was large (as when the state of a whole paragraph is sent) did not affect the overall performance of the algorithm.

Figure 10 presents the same measurements made for the cWorld application. As can be seen, dARB has lower performance in this case because the arbitration phase is

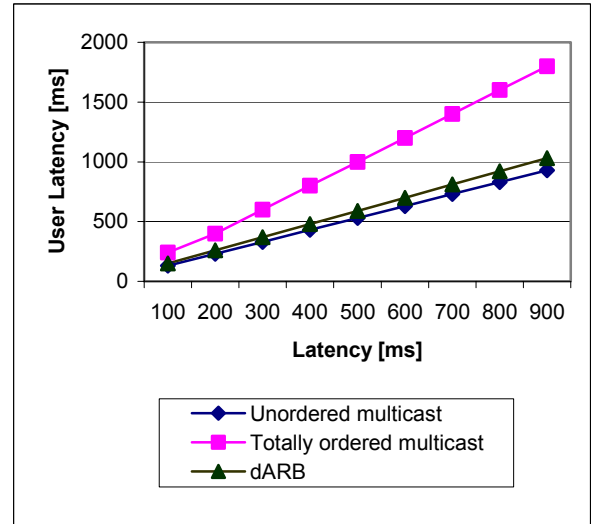


Figure 9: User latency vs. latency for the TextEditor.

more frequently invoked than in the previous case. As mentioned before, all the insert operations access the same vertex, the root of the tree. Although at first it appears that the insert operations do not conflict (i.e., the order in which two objects are created does not matter), this is not so. Assume that user α inserts a cylinder and user β concurrently inserts a cube. If no arbitration is made, the users will end up with different Z-orders of the inserted objects: α will have first the cylinder and after that the cube, while user β will have the opposite. This situation can affect the correctness of the configuration if later the cylinder gets moved to overlap, it will end up in front (or behind) the cube. Thus, we have to arbitrate at all concurrent insertions by sending the winning site's Z-order to all the other participants.

Another reason for decreasing the performance of dARB for cWorld is the greater granularity of 3D objects (compared to words in the TextEditor), which increases the probability that two or more users will concurrently try to modify the properties of the same object. Nevertheless, the performance remains very good compared to total ordering of events, especially for large latencies in the system.

SUMMARY AND FUTURE WORK

Asynchronous process execution and communication delays potentially create nondeterminism in distributed/replicated systems. Imposing a total ordering on the events can solve this problem at the price of degrading the responsiveness of the system, especially in systems with a large latency. This paper presents a solution to the concurrency control problem based on the tree data structure of application documents, which reduces the overhead of applying operational transforms or undo operations and does not require the maintenance of extensive operation logs. We implemented the algorithm

for two applications using a standard XML parser to create and maintain the tree structure for application documents.

Because local operations are executed immediately without locking policies, the responsiveness of the system is very good, close to that of a single-user application. The algorithm is completely distributed since no centralized information is assumed. If a site crashes, the remaining sites are able to continue the collaborative work. The algorithm is also very flexible in the sense that its granularity can be dynamically increased or decreased depending on the application specific requirements. For example, for the TextEditor the leaf vertices can be recursively divided into simpler objects (single characters). The measurements indicate that the algorithm scales very well with the number of users and that the introduced user latency is very close to that of reliable multicast alone.

Our continuing work focuses on increasing the capabilities of the applications (cWorld, TextEditor) in order to obtain useful collaborative applications for evaluation. We are also exploring dynamic joining or leaving a configuration in DISCIPLE in a robust and consistent manner. Further information on DISCIPLE is available at:

<http://www.caip.rutgers.edu/disciple/>

ACKNOWLEDGMENTS

This research is supported by NSF KDI Contract No. IIS-98-72995 and by the Rutgers Center for Advanced Information Processing (CAIP).

REFERENCES

1. Bhola, S., Guruduth, B., and Ahamad, M. Responsiveness and consistency tradeoffs in interactive groupware. *Proc. of the ACM Conference on Computer Supported Collaborative Work (CSCW'98)*, pp.79-88, November 1998.
2. Bhola, S., Mukherjee, B., Doddapaneni, S., and Ahamad, M. Flexible batching and consistency mechanisms for building interactive groupware applications. *Proc. of the Int'l Conf. on Distributed Computing Systems (ICDCS'98)*, May 1998.
3. Birman, K. P. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.
4. Chen, D., and Sun, C. A distributed algorithm for graphic objects replication in real-time group editors. *Proc. of the Int'l ACM Conf. on Supporting Group Work (GROUP'99)*, pp.121-130, November 1999.
5. Cormack, G. A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication. *University of Waterloo Technical Report*, CS-95-08, 1995.
6. Dewan, P. Architectures for collaborative applications. In *Computer Supported Co-operative Work*, M. Beaudouin-Lafon (Ed.), John Wiley & Sons, Chichester, England, pp.169-193, 1999.
7. Dorohonceanu, B., Sletterink, B., and Marsic, I. A novel user interface for group collaboration. *Proc. of the 33rd Hawaii Int'l Conference on System Sciences (HICSS-33)*, January 2000.
8. Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. *Proc. of the 19th ACM SIGMOD Conf. on the Management of Data*, pp.399-407, 1989.
9. Karsenty, A., and Beaudoin-Lafon, M. An algorithm for distributed groupware applications. *Proc. of the Int'l Conference on Distributed Computing Systems (ICDCS'93)*, pp.195-202, May 1993.
10. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.
11. Liao, T. Lightweight reliable multicast protocol specification. Internet draft: <http://webcanal.inria.fr/lrmp>, 1999.
12. Ressel, M., Nitsche-Ruhland, D., and Gunzenhauser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. *Proc. of the ACM Conference on Computer Supported Collaborative Work (CSCW'96)*, pp.288-297, November 1996.
13. Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63-108, March 1998.
14. Sun, C., and Sosc, R. Optional locking integrated with operational transformation in distributed real-time group editors. *Proc. of the 18th ACM Symposium on Principles of Distributed Computing*, May 1999.
15. Sun Microsystems, Inc. JavaBeans API Specification, <http://www.javasoft.com/beans/>
16. Sung, U., Yang, J., and Wohn, K. Concurrency control in CIAO. *Proc. of IEEE Virtual Reality Conference*, pp.22-28, 1999.
17. W3C Architecture Domain: Extensible Markup Language, 1999. <http://www.w3.org/XML>
18. Wang, W., Dorohonceanu, B., and Marsic, I. Design of the DISCIPLE synchronous collaboration framework. *Proc. of the 3rd IASTED Int'l Conference on Internet, Multimedia Systems and Applications*, pp.316-324, October 1999.