



Tree-Based Concurrency Control in Distributed Groupware

MIHAIL IONESCU & IVAN MARSIC*

Center for Advanced Information Processing (CAIP), Rutgers – The State University of New Jersey, Piscataway, NJ 08854-8088, USA (*Author for correspondence: Phone: +1 732 445 4208; Fax: +1 732 445 4775; E-mail: marsic@caip.rutgers.edu)

Abstract. We present a novel algorithm, called dARB, for solving the concurrency control problem in distributed collaborative applications. The main issue of concurrency control is resolving the conflicts resulting from simultaneous actions of multiple users. The algorithm reduces the need for manual conflict resolution by using a distributed arbitration scheme. The main advantages of our approach are the simplicity of use and good responsiveness, as there are no lock mechanisms. Our algorithm requires the applications to use a tree as the internal data structure. This makes it application independent and suitable for general collaborative applications. The tree requirement is reasonable since many new applications use XML (extensible Markup Language) for data representation and exchange, and parsing XML documents results in tree structures. Example applications of the algorithm, a group text editor and a collaborative 3D virtual environment called cWorld, are implemented and evaluated in the DISCIPLINE collaboration framework. We also introduce awareness widgets that users avoid generating the conflicting events and help in manual conflict resolution.

Key words: concurrency control, distributed algorithms, groupware

1. 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 (Ellis and Gibbs, 1989; Sun et al., 1998) or to draw on a shared whiteboard. The collaboration systems are characterized by the following specific requirements (Sun et al., 1998): (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.

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 require-

ments. It can be summarized as follows. We assume that the data structure of the application document is a *tree*. Since the data structure is fixed and only a few operations apply to it, the algorithm is simple and general for all applications. Operations are executed immediately at their originating site and locks are not required to access the data. If two or more concurrent actions are applied to different vertices of the tree, the partial ordering of the messages ensures maintaining the consistency of the configuration. However, if the concurrent operations try to access the same vertex, an *arbitration phase* is started. The site that wins the arbitration phase updates the state of the other participating sites and the consistency is preserved. 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. Our algorithm is entirely distributed and resilient to site failures so that in the case of failure the remaining sites can work without interruption.

The tree data structure may not be the most efficient data type for all applications, but settling on one data type simplifies the algorithm and makes it general for all applications. We chose tree because it is the data structure that results from parsing XML documents (W3C Architecture Domain, 1999). XML is now being promoted as the new Web markup language for information representation and exchange. We expect that most of the Web data will be exchanged in XML and most of the collaborative applications will benefit from choosing this data format. Some applications may suffer performance penalty due to fixing the shared document structure. For example, a spreadsheet can be more efficiently represented as a multidimensional array. The performance of a tree-based spreadsheet may degenerate for a large document. However, we believe that such cases would appear relatively rarely in practice and the gains from having a general solution far outweigh the drawbacks.

Dewan (1999) argues that any collaborative application can be seen as a generalized editor of semantic objects defined by it. A user interacts with the application by editing a rendering of these objects using text/graphics/multimedia editing commands. In addition to passive editing, in a generalized editor the changes may trigger computations or behaviors in the objects. In that sense, the framework presented here can be used in any groupware application that deals with structured documents, such as a multiuser editor or a collaborative virtual world. An earlier version of this work was presented in (Ionescu et al., 2000).

The paper is structured as follows. We first present assumptions and definitions necessary to better understand the rest of the paper. Next, we overview the related work and outline the major contributions of our work. Then we present the dARB algorithm for solving the concurrency control problem based on a specific example. We analyze the behavior of the algorithm and sketch a formal proof of correctness. We also measure the performance of the algorithm in several applications. Lastly, we summarize our major findings and outline future work.

2. Assumptions and definitions about the collaborative framework

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,

$$\Gamma = \langle S_{\alpha 1}, \dots, S_{\alpha n} \rangle.$$

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 (Birman, 1996), 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 (Lamport, 1978) 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 the 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

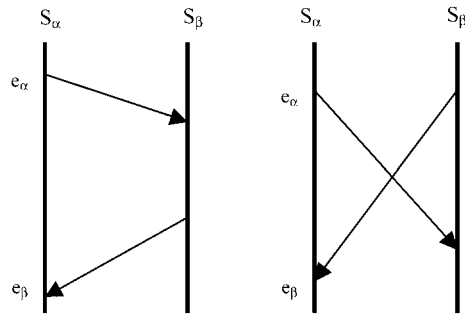


Figure 1. Non-overlapping and overlapping operations.

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 $e_{\alpha 11}, e_{\alpha 12}, \dots, e_{\alpha 1k}$, 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 (Birman, 1996). Total ordering is the most general method of maintaining the configuration correctness. However, as demonstrated in (Bhola et al., 1998a), total ordering degrades the responsiveness of the system relative to partial ordering. The concurrency control presented here offers a more responsive alternative.

Our DISCIPLINE framework also offers a totally ordered channel using a slightly modified logical clocks algorithm (Lamport algorithm). 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 lastly by the site ID. Each sending site computes the priority dynamically.

The sites communicate with each other using a reliable multicast protocol (Liao, 1999). A multicast session can be uniquely defined by a multicast IP address and a port number. The algorithm presented here does not assume the existence of a server, so when a site joins, it simply joins the multicast group. When multicast is not available, e.g., due to security constraints, we implemented a simulated multicast protocol based on TCP using a centralized server (Ionescu and Marsic, 2001). An event is first executed locally and then it is sent to the server that broadcasts it to all of the participating sites. The scheme also supports a totally ordered channel, in which case the event is sent to the server and is executed locally only after it gets reflected from the server.

DISCIPLE allows importing and sharing arbitrary Java Beans (Marsic and Dorohonceanu, 1999). 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 store and remotely exchange 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 Java beans.

3. Related work

The most important optimistic algorithms for solving the concurrency control in distributed groupware include dOPT (Ellis and Gibbs, 1989), adOPTed (Ressel et al., 1996), SOCT2 (Suleiman et al., 1997), GOTO (Sun et al., 1998), SOCT3 and SOCT4 (Vidot et al., 2000). All of these use a transformation function, called *forward transposition*, in order to maintain the consistency. Let S be a consistent state across the sites α and β , $S \bullet Op$ is the state obtained after the execution of Op and Intention(Op, S) is the intention which is realized by operation Op on the state S . The forward transposition function is formally defined as follows:

$$\text{Transpose_forward}(Op_\alpha, Op_\beta) = Op_\beta^{Op_\alpha}, \text{ where: } \forall S_\alpha, \text{ and } Op_\alpha, Op_\beta \text{ are concurrent operations, } \text{Intention}(Op_\beta^{Op_\alpha}, S \bullet Op_\alpha) = \text{Intention}(Op_\beta, S)$$

This is illustrated in Figure 2. We assume that the site β wins the conflict resolution because of the prior ordering of the sites. Then, at the site α the operation Op_β is transformed using forward transposition in order to cancel the effect of the previously applied operation Op_α . At the site β the operation Op_α is ignored. As pointed out in (Vidot et al., 2000), forward transposition needs to fulfill the following conditions, in order to achieve convergence of the copies:

Condition C1: Let Op_1 and Op_2 be two concurrent operations defined on the same state. The forward transposition satisfies C1 iff: $Op_1 \bullet Op_2^{Op_1} \equiv Op_2 \bullet Op_1^{Op_2}$.

Condition C2: For any operations Op_1, Op_2, Op_3 , the forward transposition satisfies C2 iff: $Op_3^{Op_1:Op_2} = Op_3^{Op_2:Op_1}$, where $Op_i:Op_j = Op_i \bullet Op_i^{Op_j}$.

Finding such a transposition function for an application and proving that it fulfills both conditions C1 and C2 is not an easy task for the majority of applications. Condition C2 is particularly difficult to meet, and for this reason some algorithms require only condition C1 and instead of C2 impose some other restrictions (like continuous global order for SOCT3). All of the above mentioned algorithms are developed only for the text editor case, where just two operations are defined. Attempts at extending the dOPT algorithm to a more complex application were reported (Cormack et al., 1995), but the implementation is very cumbersome

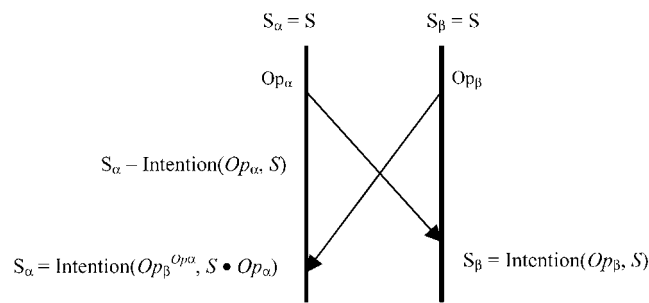


Figure 2. An illustration of the use of the forward transposition function.

and it is difficult to prove its correctness. Consider, for example, a text editor in which the user is also able to insert graphical objects. Finding a transposition function to satisfy the two conditions would be extremely difficult. Moreover, every time the application evolves (for example by adding or extending an operation), the transposition function should be revised, since the conditions C1 and C2 must be met for every operation. This makes the maintenance of a collaborative application a very difficult task and can introduce subtle errors.

A hybrid solution of pessimistic and optimistic concurrency control is proposed in (Sun and Soscic, 1999), using both operational transformations and locking. The authors argue that locking alone cannot entirely solve the inconsistency problems in group text-editors. The major advantage of their approach is that it does not incur any locking overhead in the most common case of collaborative editing. In this respect, it is similar to our solution insofar as we also incur no overhead in the most common case of collaborative work. The main difference is in the way the conflicts are resolved. Our algorithm also requires the users to resolve some consistency problems when the system cannot solve them automatically in an efficient manner.

In (Bhola et al., 1998b) and (Chen and Sun, 1999), the authors propose several ways to solve the concurrency control problem. They too structure the document into different objects and use replication and cloning of the concurrently accessed objects to let the users resolve the conflict. Our work is an alternative to these algorithms by imposing a standard structure on the document (a tree) and solving automatically as many conflicts as possible. We assume no central server, as opposed to (Bhola et al., 1998b), and assist users in avoiding and solving conflicts via awareness widgets.

For 3D collaborative environments, there are some other solutions proposed for maintaining the correctness of a configuration. In CIAO (Sung et al., 1999), the authors implement a hybrid approach by using tokens for each object. A participant starts the manipulation of an object without waiting and multicasts a message containing the object identifier, one plus the version number, and the operation. The token owner can validate the operation so the user can continue or invalidate it by sending the state of the object. This solution is suitable for the environments in which the number of objects is small and does not grow over time, but cannot

be applied to a text editor application, for example. Our algorithm presented in this paper can deal with a much larger number of objects and is independent of the application semantics.

4. Major contributions of this work

We believe that this paper makes two major contributions. First, we introduce a new optimistic concurrency control algorithm that does not need a transposition function, which can improve drastically the flexibility and scalability of a collaborative framework, as we discussed in the previous section. The algorithm is generic and can be used for a large class of applications, and is not influenced by the modification of the logic of the application. Because of its generality, we have been able to develop different types of applications without any modification in the algorithm, which was not possible with the previous optimistic algorithms. We applied our dARB algorithm on a complex graphical editor and we believe it can be used for other complex applications. Secondly, we give a quantitative evaluation of our algorithm and demonstrate that it performs much better than the pessimistic algorithms. Although a comparison between our algorithm and other optimistic algorithms would be very interesting, we are unable to perform this because no quantitative performance data is reported by any of the other algorithms that we are aware of and our possible implementation of these algorithms may not be optimal due to the lack of the implementation details that are not mentioned in the related publications.

5. The dARB algorithm

The dARB algorithm is general and can be applied to any collaborative application which uses the tree data structure for document representation. It does not require any specific knowledge about the application or its functionality.

5.1. DATA STRUCTURES

Tree structure. Each site α maintains its application document in a tree structure, T_α . Each vertex has an associated unique identifier, which is the same at all sites. The vertex also keeps the count of operations successfully applied to it, which is used to detect whether or not the operations overlap. 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 to a tree are: create vertex (Op_1), delete vertex (Op_2), and edit vertex properties (Op_3). Our current implementation does not consider the operations to add/delete properties of a vertex because we assume that all the properties of a vertex are a priori specified. However, the algorithm presented below can be easily modified to account for new operations. Even though some vertices may refer-

ence 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 exist methods on an event object e for retrieving information about the site id, data, path to the accessed vertex id along with the accessed vertex, and the count associated with the vertex: $\text{siteID}(e)$, $\text{data}(e)$, $\text{path}(e)$, $\text{count}(e)$.

5.2. 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 op from the user interface

if (op is a create operation)

$\text{newID} \leftarrow \text{create a new ID}$

 create a new vertex with associated id

$p \leftarrow \text{path from root to the new vertex}$

 execute the operation op

$\text{parID} \leftarrow \text{id of the parent of the newly created vertex}$

 increment $\text{count}(\text{parID})$

create event e from id, p and op

broadcast e to all the other sites

if (op is a delete operation)

$\text{id} \leftarrow \text{the id of the accessed vertex}$

$p \leftarrow \text{path from root to the vertex}$

 execute the operation op

 modify all the vertices in the tree depending on the deleted vertex

$\text{parID} \leftarrow \text{id of the parent of the vertex}$

 increment $\text{count}(\text{parID})$

 create event e from id, p and op

 broadcast e to all other sites

otherwise // op is modify property operation

$\text{id} \leftarrow \text{the id of the accessed vertex}$

$p \leftarrow \text{path from root to the vertex}$

```

    d ← information about the property changes of the vertex
    execute the operation op
    increment count(id)
    create event e from id, p, d and op
    broadcast e to all other sites
  endif

Remote operation:
  receive e = ⟨β, op, p⟩ from remote
  if (op is a create operation)
    parID ← id of the parent of the newly created vertex
    execute the operation op
    if (count(parID) ≥ count(e))
      performArbitration()
    else
      increment count(parID)
    endif
  if (op is a delete operation)
    parID ← id of the parent of deleted vertex
    p ← path(e)
    execute the operation op
    modify all the vertices in the tree depending on the deleted vertex
    if (count(parID) ≥ count(e))
      performArbitration()
    else
      increment count(parID)
    endif
  otherwise // op 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 op
    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 times t_α , t_β and t_γ that concurrent accesses have been attempted 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 itself as the source of the event *wins* the arbitration process. In Figure 3, 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 does not make any assumptions about the order in which different sites receive the messages, so it works unchanged in the centralized configuration (with the simulated multicast) as well.

The algorithm can be implemented as a separate module and, as long as the application provides the tree represented in a standard format, the same implementation can be used with different applications. Here we give two example applications.

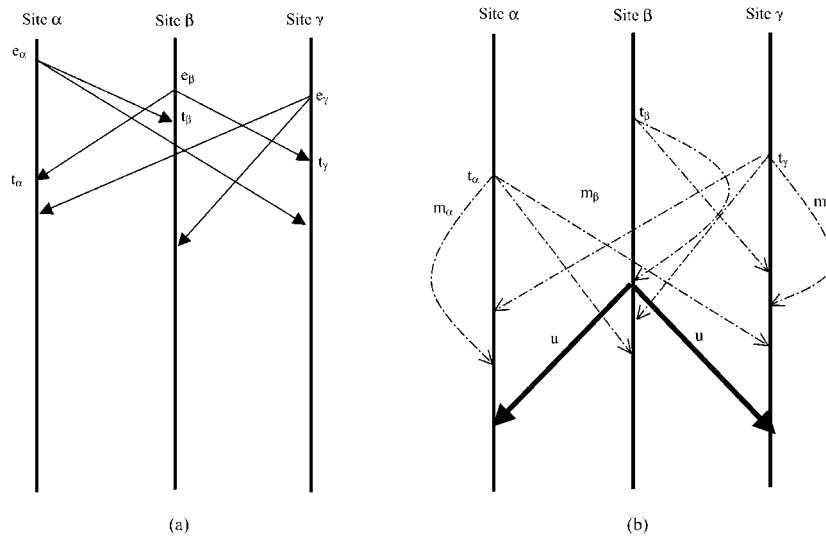


Figure 3. Arbitration phase for three sites. (a) The concurrent events e are detected at times t . (b) This triggers sending special prioritized events m to the ordered channel. The winning site sends the update messages u to the others.

5.3. EXAMPLE 1: A SIMPLE TEXT EDITOR

A basic text processing system might simply have a character string as its application document as in (Ellis and Gibbs, 1989) and define two operations:

- $Op_1 = insert(X;P) =$ insert character X at position P
- $Op_2 = delete(P) =$ delete the character at position P

In this case, any two operations that overlap (Figure 1) are also logically concurrent, so they conflict. In (Ellis and Gibbs, 1989) 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 (Cormack, 1995). Our approach of adopting a different data structure for the application document solves the problem in a simpler 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 position relative to the beginning of the word specify the property modification operation.

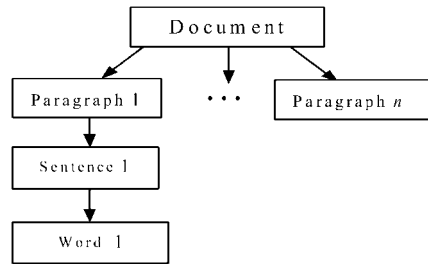


Figure 4. Tree structure of the application document for the TextEditor application.

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.

5.4. 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, 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.

5.5. UNDOING OPERATIONS

Undoing operations in collaborative systems imposes additional challenges as compared to single-user applications (Prakash and Knister, 1994). *dARB* introduces an additional problem compared to the framework described in (Prakash and Knister, 1994) in the sense that it may happen that an operation executed by a site disappears after the arbitration phase (e.g., the site lost the arbitration



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.

phase). Moreover, the history list of the operations cannot be properly maintained after an arbitration phase is over, since the winning site sends only its current state to the other sites. To address these issues we modified the selective undo algorithm presented in (Prakash and Knister, 1994). The new algorithm, called dARB Selective Undo Algorithm is an extension of the previous algorithm by introducing *checkpointing*. After an arbitration phase is completed and the new state becomes common for all the participating sites, this common state is considered as a *checkpoint*. The undo operations are allowed using the Selective Undo Algorithm but only until a new *checkpoint* is created (i.e., until a new arbitration phase is completed).

6. 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 tree 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 assigned by the winning site is greater, the site should send not only the state of the vertex itself but perhaps also the state of the parent or a whole subtree.

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.

We chose to send the winning site’s state instead of the operation(s) since sending the operation(s) would require undoing the operations performed on the receiving sites. Undoing operations is not easy since it is difficult to know what operations to undo. The disadvantage of sending states is that the message containing the winning site’s state may be quite large. 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. This situation happens very rarely in practice, except in the case of very large documents. In this case, the document will likely be structured so that individual objects are grouped into composite objects, which form many large subtrees in the document tree. The user interactions will concurrently modify the subtrees resulting in exchange of large amounts of data for every arbitration.

Although the algorithm does not always 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 , 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.

6.1. 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 DISCIPLÉ several types of group-awareness widgets (Dorohonceanu et al., 2000). Telepointers are widgets that allow a given user to track remote users’ cursors. Figure 6 shows the text editor bean imported in DISCIPLÉ with telepointers. In addition, the users can exchange messages, post small notes, and annotate regions of the bean window.

We also point out that at present dARB does not solve the causality violation and intention violation problems as defined in (Sun et al., 1998). This is because dARB accepts only *one* of the conflicting operations, the one from the participating site that performs the operation with the highest priority. This may confuse the users whose operations were not accepted by dARB. However, if the awareness widgets are activated, the user confusion can be reduced or even completely avoided. For example, it is reasonable to expect that, after seeing another user’s telepointer at the location of the intended change, the user realizes that the operation will conflict with the other user’s activity and does not perform any action in that moment. This reduces the number of conflicts and increases the global conflict resolution algorithm performance.

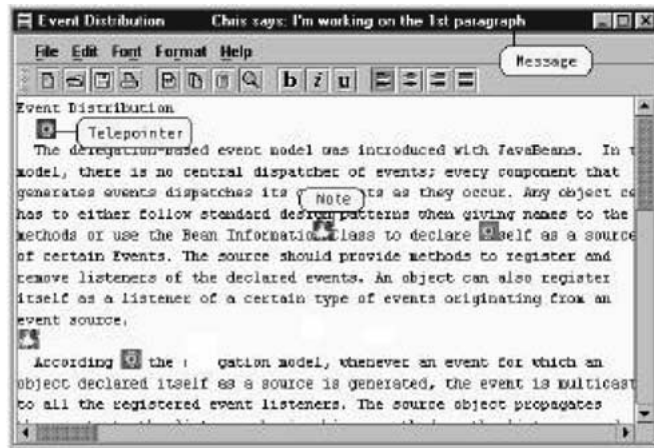


Figure 6. The text editor application in the DISCIPLÉ framework. The user sees his/her own and remote users' cursors. If two cursors overlap it means that those two users work in the same area of the application, and, therefore, they may perform concurrent actions.

6.2. ALGORITHM CORRECTNESS

This section sketches the proof of correctness of the algorithm. For the sake of simplicity, we make the following assumptions about the communication medium:

1. The latency between any two sites is the same and is *quasi-stationary* (in the sense that, for short periods of time, remains constant). Let us denote by L the value of the latency between the sites.
2. If site α multicasts an event $e_{\alpha 1}$ and after a *short* period of time, it multicasts an event $e_{\alpha 2}$ any receiving site will receive the events in the order $e_{\alpha 1}, e_{\alpha 2}$.

We say that the event e_1 is smaller than e_2 (and denote this by $e_1 \rightarrow e_2$) if e_1 precedes e_2 using the partial order defined in Section 2. We also assume that the configuration $\Gamma = \langle S_{\alpha 1}, \dots, S_{\alpha n} \rangle$ is correct at the time t_0 . We will prove that, after an arbitrary number of operations using dARB as the concurrency control algorithm, either the correctness of the algorithm is maintained automatically (in a great majority of cases), or the users are informed that the correctness is no longer maintained and they should resolve it manually before resuming the collaborative work. We consider that there are N sites in the configuration and that C sites are involved in concurrent accesses of the same vertex. It is obvious that, if the sites are accessing different vertices, the correctness of the configuration is automatically maintained by dARB.

Lemma 1. Consider a subset A of Γ , where $|A| = C$ and $C \leq N$. Each site $\alpha \in A$ accesses concurrently the same vertex in the associated document tree by sending a variable number of events q_α . We consider that the events in q_α are only the events related to the accesses to the tree structure. If there exists no site $\beta \in A$ such that

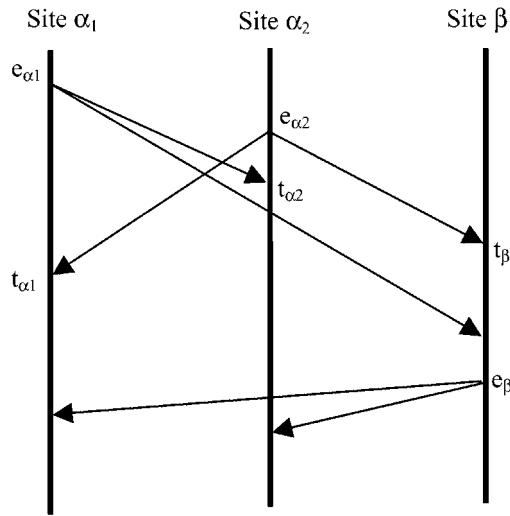


Figure 7. Illustration of the case when the condition in the hypothesis of Lemma 1 is not satisfied. $A = \{\alpha_1, \alpha_2\}$.

$e_\alpha \rightarrow e_\beta$, where and $e_\beta \in \bigcup_{\alpha \in A, \alpha \neq \beta} q_\alpha$ and $e_\beta \in q_\beta$, then all the sites in A participate in the arbitration phase for the concurrently accessed vertex V .

Proof. Suppose the opposite, i.e., suppose there exists a site β such that it accesses the vertex V and does not participate in the arbitration. Consider a site α that participates in the arbitration. Since the site β does not participate in the arbitration, this implies that its message does not conflict with any of the received messages, so it has received all the other events before sending its own event (because of the second assumption), which contradicts the condition from the hypothesis. \square

The condition in the hypothesis of this Lemma is illustrated in Figure 7. In fact, Figure 7 shows the case when the hypothesis of Lemma 1 is not satisfied.

Lemma 2. Consider the same notation as in Lemma 1. Then, either all of the sites from A participate in the arbitration phase or the algorithm can decide that a conflict has not been solved in time $O(L)$.

Proof. According to Lemma 1, all of the sites that satisfy the condition of Lemma 1 participate in the arbitration phase. Suppose now that a site β does not satisfy the condition of Lemma 1. Assume that a site α_1 won the arbitration. When starting the arbitration phase, it also multicasts a *Lock* event which arrives at the site β after at most time L (following our first assumption). If the site $\beta \in A$, it means that its event e_β was transmitted before receiving the *Lock* message, so the site α_1 receives this event after at most $2L$ time. The events sent by α_1 in the arbitration

phase should be acknowledged by every site, including β , since these messages are sent using the totally ordered channel. Following the first and second assumptions, the acknowledgement arrives after more than $2L$ time back to the site α_1 . If α_1 receives e_β and the acknowledgement before receiving the arbitration event from the site β , it can assume that β will never start the arbitration phase (as in Figure 7) so the conflict could not be solved but it was *detected*. This proves Lemma 2. \square

Theorem. The dARB algorithm is correct in the sense that it detects all the conflicts that cannot be automatically solved.

Proof. We use the same notation as in Lemma 1. We can assume without loss of generality that all the sites from A participate in the arbitration phase (otherwise the conflict is not solved but it is detected, as shown in Lemma 2). Let α denote the winning site. If α decides that the conflict can be solved (i.e., it can broadcast the state of the modified vertex), the conflict will be solved. If not, the conflict is detected, which proves the theorem. \square

7. Performance evaluation

In order to evaluate the performance of our algorithm, we have to select the performance metrics. As also argued in (Bhola et al., 1998a), 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 B2-based workstations connected in a 100 Mb/s LAN. 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 introduced overhead increases because the probability of concurrent accesses to the same object increases. Notice that the performance for unordered multicast is not a flat line (as we would expect for an unreliable multicast protocol), since we used in evaluation a reliable multicast protocol (Liao, 1999), which introduces certain overhead.

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.

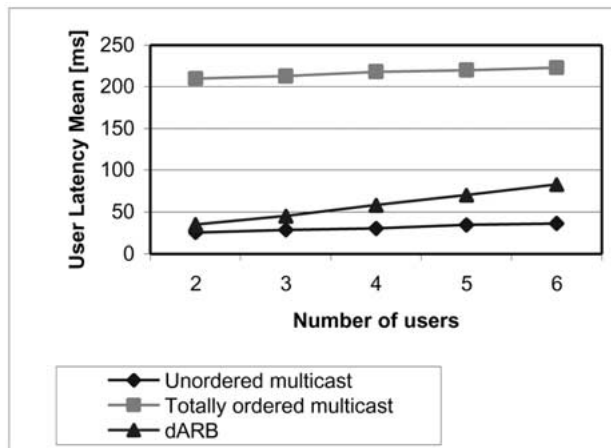


Figure 8. User latency vs. number of users for the TextEditor.

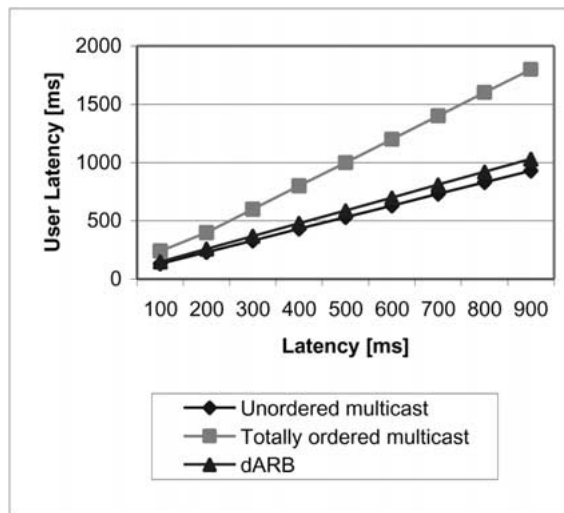


Figure 9. User latency vs. latency for the TextEditor.

As it can be seen, the performances of dARB for TextEditor are very good, close to the performances of the unordered reliable multicast. Naturally, if the conflicts are very frequent, the arbitration will be invoked very frequently and the speed will degenerate to that of the total ordering or even worse if they cannot be resolved automatically. 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%).

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

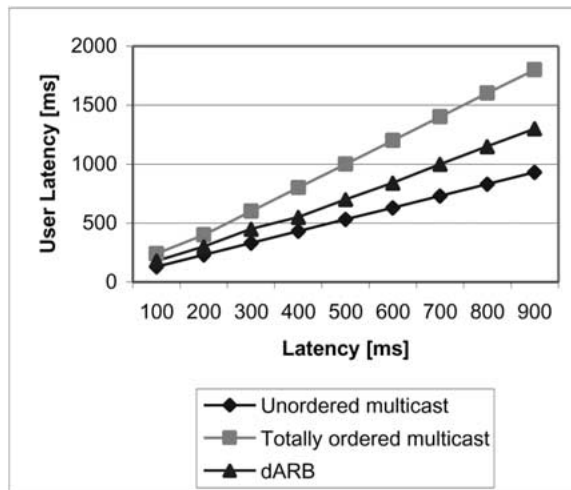


Figure 10. User latency vs. latency for the cWorld.

phase is 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.

8. 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 research focuses on better specifying the guidelines for the application developers regarding the applicability of our algorithm. In applications where the likelihood of concurrent accesses is very low, a less sophisticated algorithm (such as optimistic locking), coupled with the use of awareness widgets to show where others are working, may be sufficient. What we believe is that our algorithm is suitable for a certain class of applications, determined by the likelihood of concurrent accesses. If the likelihood of concurrent accesses exceeds certain upper threshold, the performance of our algorithm becomes worse than for total ordering. As for the lower threshold, defining this threshold is the first step and the definition should take into account the less sophisticated algorithms, such as optimistic locking. Once defined, both thresholds will be measured and offered to the application developers as a guideline in deciding where to employ the dARB algorithm. We are also interested in providing novel solutions for concurrency control when a large number of sites are involved in collaborative work. Further information on DISCIPLE is available at: <http://www.caip.rutgers.edu/disciple/>

Acknowledgments

The authors are indebted to the anonymous reviewers, whose comments helped to greatly improve the quality of the paper. This research is supported in part by NSF KDI Contract No. IIS-98-72995 and by the Rutgers Center for Advanced Information Processing (CAIP).

References

- Bhola, S., B. Guruduth and M. Ahamad (1998): Responsiveness and Consistency Tradeoffs in Interactive Groupware. *Proceedings of the ACM Conference on Computer Supported Collaborative Work (CSCW'98)*, pp. 79–88.
- Bhola, S., B. Mukherjee, S. Doddapaneni and M. Ahamad (1998): Flexible Batching and Consistency Mechanisms for Building Interactive Groupware Applications. *Proceedings of the Int'l Conf. on Distributed Computing Systems (ICDCS'98)*.

- Birman, K.P. (1996): *Building Secure and Reliable Network Applications*. Manning Publications Co.
- Chen, D. and C. Sun (1999): A Distributed Algorithm for Graphic Objects Replication in Real-Time Group Editors. *Proceedings of the Int'l ACM Conf. on Supporting Group Work (GROUP'99)*, pp. 121–130.
- Cormack, G. (1995): A Counterexample to the Distributed Operational Transform and a Corrected Algorithm for Point-to-Point Communication. *University of Waterloo Technical Report*, CS-95-08.
- Dewan, P. (1999): Architectures for Collaborative Applications. In M. Beaudouin-Lafon (ed.): *Computer Supported Co-operative Work*. Chichester, England: John Wiley & Sons, pp. 169–193.
- Dorohonceanu, B., B. Sletterink and I. Marsic (2000): A Novel User Interface for Group Collaboration. *Proceedings of the 33rd Hawaii Int'l Conference on System Sciences (HICSS-33)*.
- Ellis, C.A. and S.J. Gibbs (1989): Concurrency Control in Groupware Systems. *Proceedings of the 19th ACM SIGMOD Conf. on the Management of Data*, pp. 399–407.
- Ionescu, M., B. Dorohonceanu and I. Marsic (2000): A Novel Concurrency Control Algorithm in Distributed Groupware. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, Las Vegas, NV, Vol. 3, pp. 1551–1557.
- Ionescu, M. and I. Marsic (2001): Latecomer and Crash Recovery Support in Fault Tolerant Groupware. *IEEE Distributed Systems Online*, vol. 2, no. 7, 13 pp.
- Lampert, L. (1978): Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, vol. 21, no. 7, pp. 558–565.
- Liao, T. (1999): Lightweight Reliable Multicast Protocol Specification. Internet draft. Available on-line at: <http://webcanal.inria.fr/lrmp>.
- Marsic, I. and B. Dorohonceanu (1999): An Application Framework for Synchronous Collaboration using Java Beans. *Proceedings of the 32nd Hawaii Int'l Conference on System Sciences (HICSS-32)*, Maui, Hawaii.
- Prakash, A. and M. Knister (1994): A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 4, pp. 295–330.
- Ressel, M., D. Nitsche-Ruhland and R. Gunzenhauser (1996): An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. *Proceedings of the ACM Conference on Computer Supported Collaborative Work (CSCW'96)*, pp. 288–297.
- Sun, C., X. Jia, Y. Zhang, Y. Yang and D. Chen (1998): Achieving Convergence, Causality-Preservation, and Intention-Preservation in Real-Time Cooperative Systems. *ACM Transactions on Computer-Human Interaction*, vol. 5, no. 1, pp. 63–108.
- Sun, C. and R. Sasic (1999): Optional Locking Integrated with Operational Transformation in Distributed Real-Time Group Editors. *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*.
- Sun Microsystems, Inc. (1996): JavaBeans API Specification. Available on-line at: <http://www.javasoft.com/beans/>
- Sung, U., J. Yang and K. Wohn (1999): Concurrency Control in CIAO. *Proceedings of IEEE Virtual Reality Conference*, pp. 22–28.
- W3C Architecture Domain (1999): Extensible Markup Language. Available on-line at: <http://www.w3.org/XML>
- Wang, W., B. Dorohonceanu and I. Marsic (1999): Design of the DISCIPLINE Synchronous Collaboration Framework. *Proceedings of the 3rd IASTED Int'l Conference on Internet, Multimedia Systems and Applications*, pp. 316–324.