Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), pages 1551-1557, Las Vegas, NV, June 2000.

A Novel Concurrency Control Algorithm in Distributed Groupware

Mihail Ionescu, Bogdan Dorohonceanu, and Ivan Marsic

Center for Advanced Information Processing (CAIP), Rutgers University
96 Frelinghuysen Road, Piscataway, NJ 08854-8088, USA

{mihaii, dbogdan, marsic}@caip.rutgers.edu

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

Abstract. *We present a new approach 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 general structure of the algorithm is application independent, which it makes it suitable for general collaboration frameworks. The algorithm applies to a set of applications that use tree as an internal data structure. This is not a serious constraint since many applications use XML (extensible Markup Language) for data representation and exchange and parsing XML documents results in tree structures. An example application of the algorithm is implemented in the DISCIPLE collaboration framework. The example applications are a group text editor and a whiteboard. We discuss the use of awareness widgets to increase the efficiency of collaborative work.*

Keywords: Groupware, distributed algorithms, concurrency control.

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 can be used to allow geographically dispersed people to edit text documents [2], [1], to draw in a shared whiteboard or to hold a brainstorming meeting [11]. The collaboration systems are characterized by the following specific requirements [1]: (1) *Good responsiveness*—the response to local user's actions should be rapid (ideally as in the 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 a distributed framework, called DISCIPLE [5], for building fault tolerant collaborative applications from single-user beans (Java Beans-compliant components, applets, and applications). A Bean is a reusable software module capable of publishing or registering its interfaces under certain contexts [10]. Beans use *events* to communicate with other Beans. A Bean that wants to receive events (a listener Bean) registers its interest with the Bean that fires the event (a source Bean). DISCIPLE loads Beans at run-time and makes them collaborative by creating dynamic adapters for sending and receiving the events.

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 concurrency control algorithm presented here attempts to fulfill all of the above requirements. However, as it is detailed later in the paper, we impose some restrictions regarding the third requirement, but we argue that these restrictions are natural in accomplishing a common task in a collaborative environment.

The paper is structured as follows. Section 2 briefly reviews the DISCIPLE framework, focusing on event-based collaboration and dynamic loading of applications. Section 3 presents our algorithm for solving the concurrency control problem based on a specific example and also discusses the possibility of generalizing the algorithm to any type of applications. Finally, we summarize our major findings and outline the future work.

2. 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, $\Gamma = <S_{\alpha 1},\ldots S_{\alpha n}>$.

Each site hosts an application that collaborates with the applications from remote sites. Each application implements a set O with arbitrary number of operations: Op_1, Op_2, … Op_n. 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* which is a passive data object; examples include a text document, a CAD object or a 2D or 3D drawing. The application document is modified by the operations included in the set O. We say that the application documents at two sites α and β are the *same* at the time t if the following two conditions hold: (a) $S_\alpha = S_\beta$ where S_α is the application document at the site α at time t and S_β is the application document at the site β at the same time, and (b) no messages are in transit between the sites. The configuration of sites is *correct* if any two distinct sites α and $\beta \in \Gamma$ have the same application documents. A key issue in distributed frameworks is maintaining the correctness of the configuration in the presence of concurrent access.

2.1.Ordering of Events

We assume that the events transmitted in the network cannot be lost but no order is guaranteed at the receiving. Generally, the operations defined

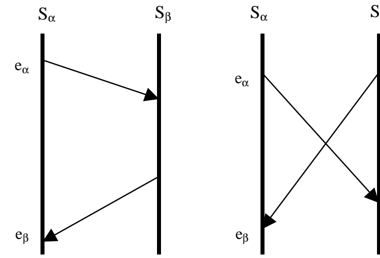

Figure 1: **Non-overlapping and overlapping operations.**

by applications do not commute. Following the Lamport's approach [3] 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.

The precedence property does not guarantee correctness of the configuration. For example, consider a two-site system $\Gamma = <S_\alpha, S_\beta>$ 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.

2.2.Dynamic Adapters

DISCIPLE is organized in two independent layers [5]: the communication layer called *collaboration bus*, dealing with event exchange, dynamic joining and leaving, concurrency control and crash recovery and *the graphical user interface*

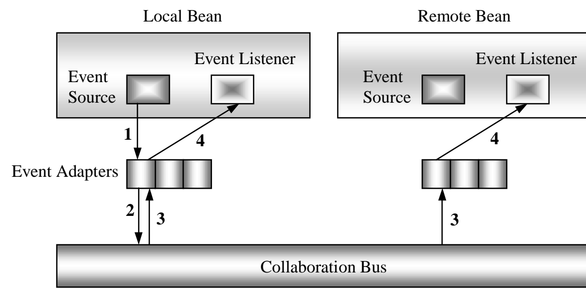

Figure 2. Event capturing and symmetric distribution scheme in DISCIPLE: (1) The Event generated by the Event Source in the Local Bean, instead of being delivered directly to the local Event Listener, is intercepted by the associated Event Adapter and (2) sent to the Collaboration Bus. (3) The bus multicasts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

*layer***,** offering a standard user interface to every application bean imported into DISCIPLE.

According to the Java event model [10], 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 of the declared events. Whenever an event for which an object 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.

Event adapters are needed since a collaboration module cannot know the methods for arbitrary events that an application programmer may come up with. 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 basic structure of DISCIPLE is presented in Figure 2.

Our goal is to make Beans collaborative without altering their source code to adapt them to our 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 [4]. 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. Unfortunately, true multicast is not always feasible since many network routers and firewalls do not support IP multicast addresses. To overcome this problem, we implemented a simulated multicast protocol based on TCP. In this case, the first site that gets started in a configuration becomes a *centralizer* for the configuration. Each new site sends to the centralizer an authentication message upon joining the session. The centralizer accepts the new site and establishes a connection with it. When a site wants to send an event it, sends the event to the centralizer, which multicasts it to all the sites.

2.3. Collaboration Using XML

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 [9] as the communication medium. Our applications use XML to represent the data model and exchanged messages. XML supports the notion of separation between view and data. The same data can be rendered in different ways to an on-screen representation.

The *data-centric* groupware paradigm separates model events (document modifications) and view events (view modifications). It offers the following advantages over conventional, application-centric groupware:

- Model-view separation at the document level via XML/XSL documents
- Simple communication protocol with a standard message format based on ASCII XML messages
- A single data structure (tree) and an associated limited set of operations

The last item is particularly important for advanced concurrency control techniques, since it offers a single data structure for application documents. 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, and consequently, the concurrency control algorithm presented here is based on tree data application documents.

3. The algorithm

This section presents our approach to solving the concurrency control problem. For simplicity, the algorithm is first presented on a particular example (a text editor) and then generalized to other applications.

A simple text processing system might have a character string as its application document [2] and define two operations O_1 and O_2, where:

O_1=*insert(X;P)* = insert character X at position P

O_2=*delete(X)* = delete the character at position P

In this very simple case, any two operations that overlap (Figure 1) will also be logically concurrent, so they will conflict. In [2] a transformation based algorithm is proposed in order 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 [12]. We believe that this problem can be solved in an easier manner by modifying the structure of the application document.

We assume that the internal data structure of the application document is a tree (Figure 3) and each node in the tree has a unique ID for all sites. Our algorithm is entirely distributed and resilient to site failures, in the sense that the remaining sites can continue without interruption. Operations are executed at their originating site immediately and

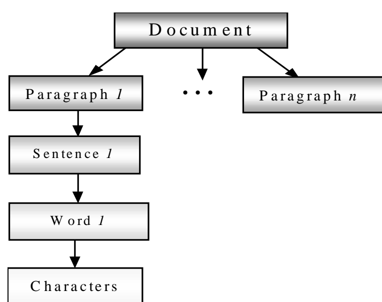

Figure 3. Tree structure of the application document for a text editor.

locks are not necessary in order to access the data. We say that a non-leaf vertex is *simple* if it belongs to the level before the leaf-level in the tree structure.

Intuitively, the algorithm works as follows. When a user inserts a character, the local site computes the path from the root to the inserted character and broadcasts this path to all the other participating sites, along with the position of the inserted character relative to the word in which it is inserted. Paragraphs end with a carriage return character, sentences end with a dot character or with a carriage return character, and words end with the space character or dot character or carriage return character. These special characters are treated differently. When a special character gets inserted, the local site creates a new vertex in the tree by assigning it an ID and broadcasts this information to the other sites. The same procedure applies in case the user deletes a character. If the character is a special character the corresponding vertex is removed from the tree.

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

However, when the operations overlap and access the same vertex, the correctness of the configuration is no longer maintained. To solve this problem, we impose an additional restriction that no more than two users can access the same simple vertex concurrently (this is assumed but not enforced). Even if this restriction is in contradiction with the general requirement of *unconstrained* collaboration, we believe that it is natural in any collaborative application. For example, in the text editor case, it is unlikely that more than two users will insert or delete characters in the same word. In other collaborative applications (e.g., a shared whiteboard), it is also very unlikely that more than two users will want to manipulate concurrently the same figure.

When a site detects that two overlapping operations access the same simple vertex, it initiates a synchronization operation with the other involved site, as there cannot be more than

two sites involved according to the restriction introduced above.

3.1.Data Structures

Tree structure. Each site α maintains its application document in a tree structure, T_α. Each non-leaf vertex has an associated unique ID, which is the same at all sites. The additional information maintained at each node is application-specific. For a text editor example, each non-leaf vertex must keep a starting and an ending point relative to the beginning of the document. However, no matter what the application is, each vertex keeps also an integer value, that is used to detect if the operations overlap or not. This value can be obtained using the method *val(objID)*.

Events. Events are tuples of the form $e = <\alpha, d, p>$, 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. In *p* there is also transmitted an integer value associated with the accessed node. There are four methods for getting from an event *e* the site id, data, the path to the accessed vertex id along with the accessed vertex, and the number associated with the vertex: `siteID(e)`, `data(e)`, `path(e)`, `val(e)`.

3.2.Algorithm

We now give a precise formulation of the concurrency algorithm executed at the site α.

```
Initialization:
     Tα ← empty

Generate operations:
     receive operation o from the user
        interface
     if (o is a create operation)
          newID ← create a new ID
          create a new vertex with associated
               id
          p ← path from root to the new vertex
          Tα ← Tα + p
          parID ← id of the parent of the
               newly created vertex
          increment the value associated with
               the vertex parID
          create event e from id, p and o
          broadcast e to all other sites
     if (o is a delete operation)
          id ← the id of the accessed vertex
          p ← path from root to the accessed
               vertex
          Tα ← Tα - p
          modify all the vertices in the tree
               depending on the deleted vertex
          parID ← id of the parent of the
               deleted vertex
          increment the value associated with
               the vertex parID
          create event e from id, p and o
          broadcast e to all other sites
     otherwise
          id ← the id of the accessed vertex
          p ← path from root to the accessed
               vertex
          d ← information of the operation
               relative to the accessed vertex
          increment the value associated with
               the vertex
          execute the operation o
          create event e from id, p, d and o
          broadcast e to all other sites
     endif

Receive operations:
     receive e=<β, o, p> from the network
     if (o is a create operation)
          p ← path(e)
          parID ← id of the parent of the
               newly created vertex
          Tα ← Tα + p
          val ← val(parID)
          newVal ← val(e)
          if (val>=newVal)
               synchronize(α,β)
          else
               increment the value associated
        with           the vertex
     if (o is a delete operation)
          parID ← id of the parent of the
               deleted vertex
          p ← path(e)
          Tα ← Tα - p
          modify all the vertices in the tree
               depending on the deleted vertex
          val ← val(parID)
          newVal ← val(e)
          if (val>=newVal)
               synchronize(α,β)
          else
               increment the value associated
        with           the vertex
     otherwise
          id ← the id of the accessed vertex
          p ← path from root to the accessed
               vertex
          d ← information of the operation
               relative to the accessed vertex
          execute the operation o
          newVal ← val(e)
          if (val>=newVal)
               synchronize(α,β)
          else
               increment the value associated
        with           the vertex
end
```

The initialization part simply sets the tree to empty. The second section receives a local operation. If the operation is to create a new vertex in the tree (e.g., inserting a carriage return character, which will create a new paragraph) a new 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., and a sentence can have ids: 1.1.1, 1.1.2, etc.

The `synchronize()` procedure is a point-to-point communication between the two involved sites and it assures that, in case that two operations overlap and the operations refer the same vertex, the application documents of the two sites are the same. In our current implementation, the site with the smallest id updates the application document of the other site. There are also other possible ways to obtain a common application document.

3.3. Analysis

We analyze here the case when two users concurrently modify the same vertex. We 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 `synchronize()` procedure is executed, the second user will have also the same 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. Telepointers are widgets that allow a given user to track remote users' cursors [6]. Figure 4 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.

During the 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 (and destroy the word, which is equivalent to deleting an vertex) and the second user wants to add an "s" to the second word, and 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 2, the site changes its application document to "Goodstudents". But when the insert operation arrives at site 1, the

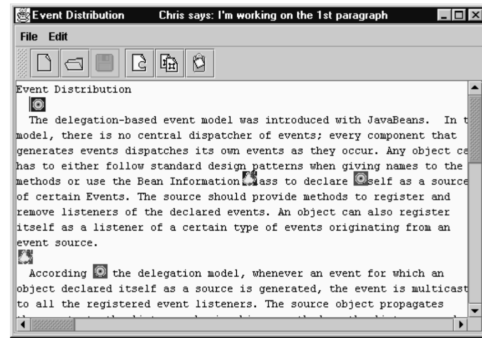

Figure 4. The text editor application in the DISCIPLE framework. The user sees own and remote users' cursors. If two cursors overlap it means that those two users work in the same area of the application, and, therefore, they may perform concurrent actions.

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. When an operation on a marked vertex is requested, the receiving site will initiate `synchronize()`with the sending site on the parent vertex of the deleted one. To reduce the dimensionality of the tree, we implemented a garbage collector that regularly scans all the structure and deletes the marked nodes if they were not used within a certain amount of time.

3.4. Extending the Algorithm

To demonstrate the generality of the algorithm, we also developed a classic shared whiteboard called Flatscape, in which two or more users can collaborate to create a complex 2D drawing. In this application, the tree structure is also a natural solution for the application document structure, as we define a number of figures (rectangles, circles, etc.), each one with its properties. We impose an analogue restriction, in the sense that maximum two users can access concurrently the same figure.

There are some important differences between the two types of application. In texteditor we have a fixed number of levels in the tree structure, while in Flatscape the number of levels can vary significantly because of the grouping and ungrouping features for the scene objects. In our current implementation, we do not support group and ungroup, so the number of levels in the tree is also constant (equal to 2). Another important difference is that for the texteditor all the insert and delete operations are performed relative to vertices that are in the lower levels of the tree (e.g. words), while in Flatscape all the insertions

are relative to the root of the tree (the document itself). Nevertheless the algorithm performs very well on both types of applications.

4. Summary and future work

Asynchronous execution of processes and communication delays potentially create nondeterminism in distributed/replicated systems. Imposing a total ordering to the events can solve this problem at the price of degrading the responsiveness of the system, especially in systems with a large latency. There are other solutions to this problem such as those based on operation transformations [2], undo op erations [7], or using the concept of constructor-based hierarchical lock compatibility tables [8]. This paper presents a solution to the concurrency control problem based on the tree structure of each application document, which reduces the overhead of applying operational transforms or undo operations. We implemented the algorithm for two different applications using standard XML parsers to create and maintain the tree structure of each application document.

Because the local operations are executed immediately without locking policies, the responsiveness of the system is good, very close to the single-user application responsiveness. The algorithm is completely distributed since no centralized information is assumed. If any 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 (i.e., the simple vertices can be recursively divided into simpler objects).

Our continuing work focuses on better solutions to the situations when conflicts occur such as situations discussed in Section 3. We are implementing a new mechanism that will support concurrency control based on operation transformations [2] for the overlap operations that are logically concurrent.

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] 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.

[2] Ellis C. A., Gibbs S. J. "Concurrency control in groupware systems". *Proceedings of 19th ACM SIGMOD Conference of Management of Data*, pp.399-407, Seattle, WA, 1989.

[3] Lamport L. "Time, clocks, and the ordering of events in a distributed system". *Communications of the ACM*, 21(7):558-565, July 1978.

[4] Liao T. Lightweight reliable multicast protocol specification, 1999. http://webcanal.inria.fr/lrmp.

[5] Wang W., Dorohonceanu B., and Marsic I. "Design of the DISCIPLE synchronous collaboration framework". *Proceedings of Internet and Multimedia Systems and Applications (IASTED '99),* pp.316-324, Nassau, The Bahamas, October 1999.

[6] Dorohonceanu B., Sletterink B., and Marsic I. "A novel user interface for group collaboration". *Proceedings of 33rd Hawaii International Conference on System Sciences (HICSS-33)*, Maui, Hawaii, January 2000.

[7] Karsenty A. and Beaudoin-Lafon M. "An algorithm for distributed groupware applications". *Proceedings of International Conference on Distributed Computing Systems (ICDCS'93)*, pp.195-202, May 1993.

[8] Munson J. and Dewan P. "A concurrency control framework for collaborative systems". *Proceedings of ACM Computer-Supported Cooperative Work* (CSCW'96), November 1996.

[9] W3C Architecture Domain: Extensible Markup Language, 1999. http://www.w3.org/XML

[10] Sun Microsystems, Inc. Java Beans specification: http://www.javasoft.com/beans/index.html

[11] Hymes C., Olson M.: "Unblocking brainstorming through the use of a simple group editor". *Proceedings of the ACM Computer Supported Cooperative Work (CSCW 1996)* November 1996

[12] 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.