A Deadlock-Free Multi-Granular, Hierarchical Locking Scheme for Real-time Collaborative Editing

Jon A. Preston and Sushil K. Prasad
Department of Computer Science
Georgia State University
Atlanta, GA 30302-3994
{jon.preston@acm.org, sprasad@cs.gsu.edu}

ABSTRACT

We describe a new scheme for enabling concurrent read and exclusive write access to a shared document while maximizing concurrent collaboration and removing the need to merge multiple disparate versions of the document. We employ a multi-granular, hierarchical locking mechanism by breaking the shared document into sections and subsections and representing these using a tree structure. The algorithm presented supports deadlock-free concurrent access through pipelined insertion and deletion of users from root down to the leaves. The lock obtained is maximized to the largest permissible sub-hierarchy of the document to minimize communication costs. Our insert and delete algorithms allow for locks to be dynamically promoted or demoted depending upon the requests made by other users in the collaborative space.

Categories and Subject Descriptors

H.5.3 [Group and Organization Interfaces]: Computer-supported cooperative work

Keywords

Computer-supported cooperative work (CSCW), collaborative editing, synchronous, consistency maintenance, distributed computing, client-server architecture

INTRODUCTION

This work is motivated by the idea of combining the benefits of optimistic and pessimistic concurrency control in a collaborative editing system. Our previous work indicates that providing multi-granular locking can improve concurrent access to a shared document in a collaborative environment. Consequently, we have developed suitable data structure and algorithms that allow users multi-granular shared access to a document with no write conflicts. All users see the same version of the document in real time, although this apparent synchronized view employs lazy updates to minimize client-server communications. The shared access to the document is transparent to the user in that the user is not required to explicitly request a lock for a section of the document; the system handles the lock request automatically. Similarly, locks are released automatically as necessary. Our algorithm avoids the problem of merging two versions of a document by providing exclusive write access. Traditionally, lack of concurrency is a key limitation of systems that employ such exclusive write access to a shared document, but our system overcomes this lack of concurrency by using a multi-granular (i.e., multi-level) locking scheme that locks sub-hierarchies of the shared document. Furthermore, each user gets exclusive write access to the largest sub-hierarchy possible to enable infrequent communication to server through relatively large messages delivering a bunch of local updates. Our system is novel in that it avoids the merge problem associated with systems that employ optimistic locking and improves concurrent access and throughput when compared to systems that employ pessimistic locking. The principle contribution of our work is an algorithm that manages these multi-granular locks and automatically increases and decreases the lock level in the document-tree hierarchy to maximize exclusive access to the shared document while minimizing communication costs.

LEAVE BLANK THE LAST 2.5 cm (1") OF THE LEFT COLUMN ON THE FIRST PAGE FOR THE COPYRIGHT NOTICE.

First, we discuss the data structure used to enable multi-granular locking. We then detail the properties of the data structure that must be maintained to enable multi-granular locking. We then discuss the detailed algorithms to add and remove users. Subsequently, we discuss existing approaches and architectures that support synchronous collaboration on a shared document and contrast with our work. Finally, we summarize our contribution and discuss future work based upon the algorithm presented. Throughout this paper, we use the term document to include word processing documents as well as source code and other user-editable files.

OVERVIEW

We begin by first discussing our previous simulation results that motivated our current work in supporting multi-granular locking. Then, we present the properties and the tree data structure for our proposed algorithms.

Previous Simulation Results

We generated and ran a simulation using the DevsJava discrete event systems toolkit [16] to validate the claim that fine-granular locking can reduce edit collision. Nine different configurations of documents distributed over multiple servers and various distributions of client users with varied editing patterns/behaviors were simulated. Adding fine-granular locking to the document management server reduced the document checkout (lock) failure rate by an average of 67%. Based upon these initial simulation results, our scheme of multi-granular locking showed much promise and motivated this research.

The Tree Structure

A document may be represented via a tree data structure wherein each node represents a sub-hierarchy of the document. The root node represents the entire document, the sub-trees represent distinct sub-hierarchies of the document, and the leaves contain the actual text. When a sub-tree (i.e. child node) is present, the parent node in the tree acts as a place holder for ownership/lock data (as specified later) and contains no text data; consequently, only leaf nodes contain the content of the document. Nodes store ownership/lock information such that if a section of the document is owned by a user u, then all subsections (i.e. child nodes in the tree) are also owned by the same user u.

Without loss of generality and to simplify proofs presented in this paper, we assume that a document may be represented by a binary tree. Figure 1 demonstrates that a document may be mapped to a tree, and then such a tree may be mapped to an equivalent binary tree. Notice that an in-order traversal of the binary tree generates the original order of the document.

Access to the Node

Two principles will guide the behavior of our system:

- When you are writing to the document, you have exclusive write access to that section of the document that you are modifying (i.e., you have a lock on the node that represents the section of the document).
- When you are reading a section of the document, you always have the most recent (fresh) copy of the content at the node that represents that section of the document.

Because of the first principle, we must provide users with mutual exclusion and the ability to lock a node in the structure. Because of the second principle, we must provide updates to all interested users that are viewing a given node (i.e. when the content at that node is changed, the change is broadcast to the users viewing that node's content). Alternatively, if a user u_i updates a section of the document at node n_i but no other user is viewing that node's content, then the changes may remain local in cache on the machine of user u_i. This "dirty cache" must be flushed to the server when another user request access to the node n_i – either through a write (lock) request or a read request.

Maintaining the Largest Sub-tree

It is advantageous to maintain a lock on the largest sub-tree that is permissible; a lock on a sub-tree rooted at node n_i is permissible for user u_i so long as no other user has a lock on any node within the tree rooted at node n_i. By maximizing the sub-tree that any user owns, we minimize the communication costs of the system with regard to cache updates. For example, if a user u_i owns the entire tree (the entire document), then all changes to the document can be stored locally in the user's cache. If another user u_j enters the system and requests a section of the document, then the section of the tree owned by user u_i is reduced to accommodate the insertion of user u_j (if possible). Only that portion of the tree that had been modified (marked dirty cache) by u_i that are part of the sub-tree now owned by u_j must be sent to u_j; the other portion of u_i's cache remain local to u_i. The result is a minimization of messaging within the system by reducing cache updates/flushing.

Finding the Correct Path from the Root

It is possible to self-route along the path from the root to any leaf node in O(1) per node [9]. This holds because each leaf node represents a unique location (section) of the document. These sections may be identified using a unique binary number, with each digit in the identifier denoting whether the node exists left or right of its parent (i.e. a 0

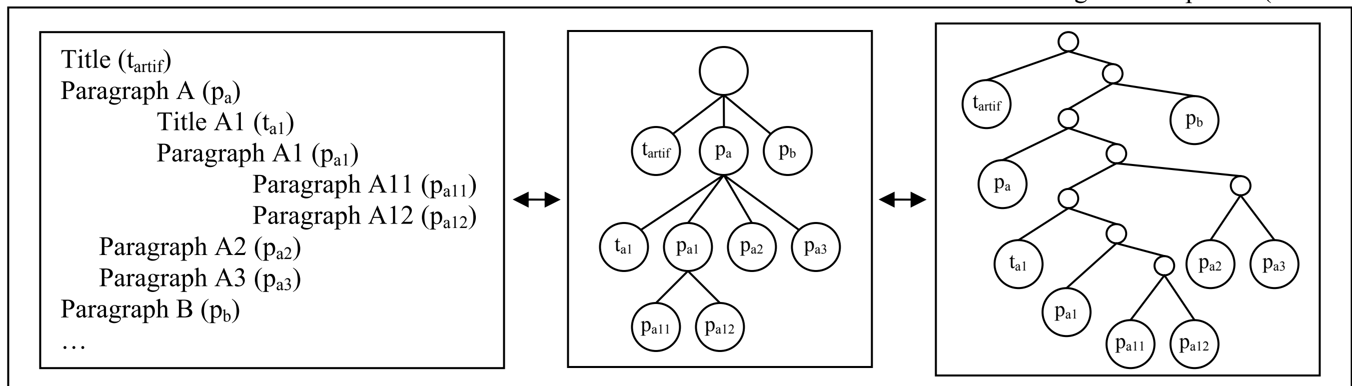

Figure 1: Mapping from document to tree to binary tree

denotes move left, and a 1 denotes move right). Thus leaf node identified with "1011" would be right, left, right, right (Fig. 2). This is implemented in the NEXTINPATH(n,w) method in the insert/delete algorithms presented later, where n is the current node in the path and w is the destination node.

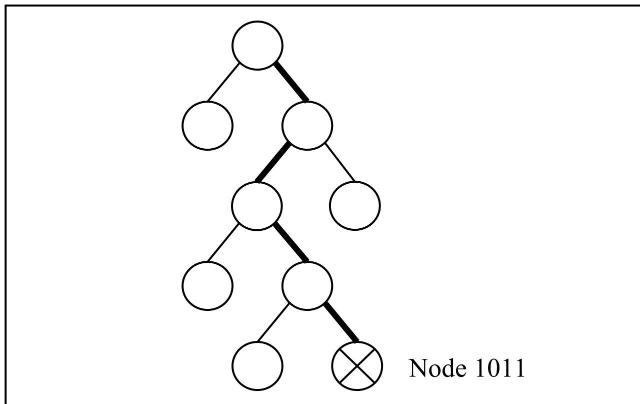

Figure 2: Path to uniquely-identified node

DATA STRUCTURE AND ALGORITHMS

Each node in the tree representing the document is colored in such as way as to represent the current state of availability of the node (sub-hierarchy of the document). There are three states that each node may be in:

- ***White***. When a node n is white, it is not locked (i.e. not owned) by any user and is available. All sub-nodes of n must be white (implying that the entire section and any subsections are not owned/locked).
- ***Black***. When a node n is black, it is currently locked (i.e. owned) by a user and is not available. All sub-nodes of n must be black (implying that the entire section and any subsections are owned/locked).
- ***Grey***. When a node n is grey, it is not locked, but there exists at least two nodes within the tree rooted at n (the grey node) that are black.

Possible child configurations of grey nodes are as shown in Fig. 3. Note that we do not show symmetric equivalents.

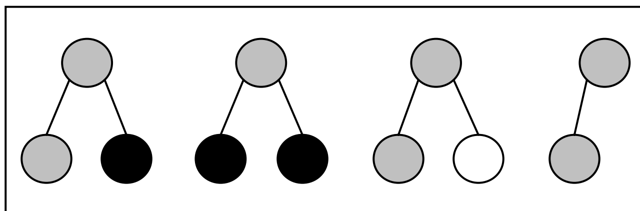

Figure 3: Grey Node Configurations

The Grey-count of a Node

Each node n_i in the tree maintains a numeric value that denotes how many nodes in the sub-trees of n_i are colored black. This is defined as the grey-count of the node n_i. This value is useful in determining if the node can be colored white or grey when a request to delete a user occurs as explained further later.

By definition, a node colored white has a grey-count of 0. Likewise, a node colored black has a grey-count of 0 as we do not recursively examine how many nodes are in the sub-trees of a black node.

The Node Structure

The node structure in the tree contains:

- Left and right children pointers
- Color (which is white, black, or grey)
- Grey count (tracks how many sub-trees of this node are owned by users)
- Owner (which, if the color is black, denotes which user owns the sub-tree rooted at the node)
- Original request (if the color is black, this denotes what leaf node in the sub-tree rooted at this node was originally requested to make this node black)
- Sibling pointer (the node's parent's other child)

Deadlock-Free Implementation

Two operations must be supported: when a user enters the system, i.e., opens a document (insert user), and when a user leaves the system (remove user). These operations require that the tree is accessed only in a top-to-bottom pipelined fashion to avoid race conditions. We enforce the policy that nodes must be accessed in a top-down manner such that we only modify the tree data structure in the following path:

- acquire a lock for the parent node
- acquire a lock for the child node
- release the lock for the parent node

This "handshake lock" technique, as employed by [9], ensures that a race condition on concurrent access to the tree data structure is avoided.

Algorithm for Inserting a User

The basic idea behind the INSERTUSER algorithm is to traverse the tree from top to bottom toward the desired leaf node along an insertion path and eventually obtain an exclusive lock on either an ancestor node that represents the largest sub-tree that contains the requested leaf node, or else on the leaf node itself.

The INSERTUSER algorithm works from top-to-bottom by examining nodes in the path from the root to the destination node. As it traverses this path, if a white node is found, then the insert succeeds and the node becomes owned by the requesting user (and painted black). If a grey node is found, it continues down. If a black node is reached, then we need to *demote* (push down) this black node (its current owner/user), turn this node into grey thus making room for the new insert request to continue down. Demotion works by recalling the originating node that was requested that is responsible for coloring this node black and moving the ownership of that user (and the black coloring) down the tree hierarchy while ensuring that the leaf node needed by

that user is contained within the sub-hierarchy. If the black node reached is a leaf node, then we can't demote any further, and the insert operation fails.

As we traverse down the path from the root to the destination node, we increase the grey-count of each grey node in the path by one; this is required as we are inserting a new black node into the tree down the path and the grey-count is responsible for tracking how many nodes are painted black below a grey node. It is optimistically assumed that the insert will succeed, but if the insert fails, then we must "undo" the artificially-inflated grey-counts along the path from the root to the destination node. We "undo" this failed insert by invoking the REMOVEUSER method (which reduces the grey-count of the grey nodes in the path from the root to the destination node by one).

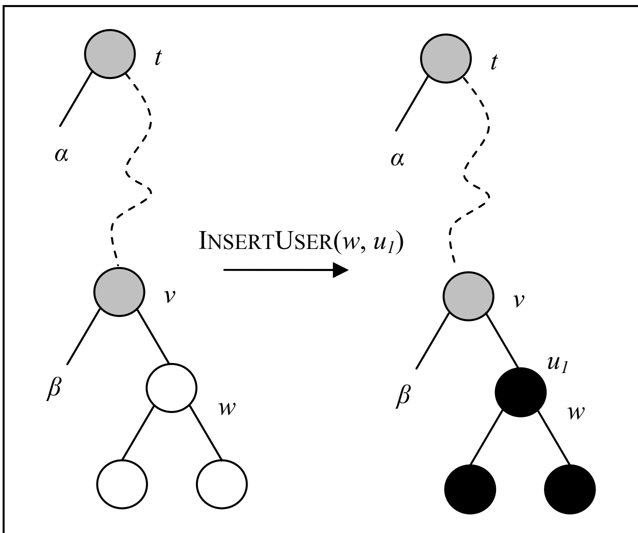

Figure 4: The INSERTUSER operation without demotion

It is assumed that when a user enters the collaborative space, the node (document section) that the user wishes to edit/lock is specified. This is logical and reasonable in that a user only enters the document if he/she wants to edit the document (since no lock is necessary to view the document). As a result, we know a priori the destination node for the insert operation and use this to guarantee that we do not attempt to insert a user u_i at a node n already owned by u_i; thus no duplicate ownership is permitted.

The lock operation is successful only on a white node, as it is the only type of node that is available for locking. If a node w is colored white, then the node and all its children are not owned by any user (See Fig. 4). Consequently, when we lock a node w, we color it and all its children black (logically).

To successfully complete an insert, it may be possible to demote ownership of a node down the tree. When the INSERTUSER operation encounters a black node in the process of trying to insert user u_1 at node w, it must demote u_2's ownership of v down; in the case shown in figure 5, u_2 wanted x and u_1 wanted w, so the conflict was resolved, but as you can see in the algorithm in figure 6, this demotion process may be recursive in the case where the node desired by the user being inserted lies along the path from n to w. In this recursive case, we simply defer the work of resolving the conflict to the sub-tree. Eventually, the insertion will either reach a white node and succeed or reach a black node that cannot be demoted (a black node that is a leaf) and fail.

Additionally, we must color all nodes in the path from w to the tree's root node t grey. The grey coloring is accomplished as the algorithm traverses down the tree (so, for example, the root is painted grey before its child is painted grey). This top-to-bottom approach preserves the deadlock free condition.

Note that nodes within the sub-trees not along the path from the root to the destination – shown as the sub-trees α and β in figures 4 and 5 – are unaffected by the INSERTUSER operation.

The detailed pseudocode for INSERTUSER and its associated routines is shown in Fig. 6. Note that by the algorithm presented, it is possible for an insert operation to fail (i.e. we are attempting to insert a user at node w where w is previously owned and is a leaf node in the tree). In this case, the algorithm INSERTUSER artificially inflates the grey-count of the nodes in the ancestry path from the root to w as it attempts to insert the user u_i at w. The algorithm compensates for this by invoking the REMOVEUSER algorithm on user u_i and node w to restore the correct grey-count values in the ancestry path. Of course, this invocation of REMOVEUSER is guaranteed to fail because u_i does not own w (or the INSERTUSER operation of u_i on node w would have succeeded initially); this failure is intentional and does not affect ownership of w (i.e. the original owner retains w). One side effect of these temporarily inflated grey-count values is that a promotion of a sibling of w may be delayed, but the algorithm ensures this promotion will eventually occur as required when the grey-count values are corrected; all other tree properties asserted earlier (coloring, ownership, structure, etc.) all remain correct.

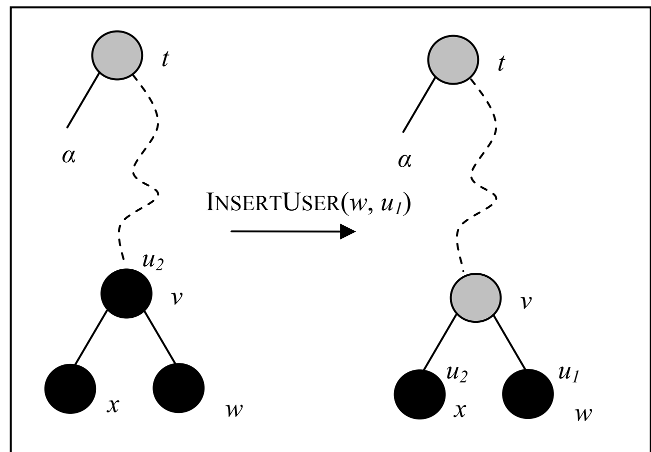

Figure 5: The INSERTUSER operation with demotion

```
INSERTUSER(w, u_i)
    if w.owner ≠ u_i
        RECURSEINSERT(ROOT, w, u_i)

RECURSEINSERT(n, w, u_i)
    if n.color = white
        then SETOWNER(n, u_i, w)
    else if n is a leaf node
        then RECURSEREMOVE(ROOT, w, u_i)
            return failure
    else if n.color = grey
        then n.greyCount = n.greyCount + 1
            RECURSEINSERT(NEXTINPATH(n, w), w, u_i)
    else b = NEXTINPATH(n, w)
        a = NEXTINPATH(n, n.originalRequest)
        SETOWNER (a, n.owner, n.originalRequest)
        n.color = grey
        n.greyCount = 2
        if a ≠ b
            then SETOWNER (b, u_i, w)
            else RECURSEINSERT(b, w, u_i)

SETOWNER(w, u_i, r)
    w.color = black
    w.owner = u_i
    w.originalRequest = r
```

Figure 6. Insertion of a User

Algorithm for Removing a User

When a user leaves the system, the user must release a lock on the node or the sub-hierarchy that the user was editing. Thus removing a user u_i is equivalent to unlocking the node n that is owned by u_i.

The basic idea behind the REMOVEUSER algorithm is to traverse the tree from top to bottom and release (mark white) the ancestor node that represents the largest sub-tree that contains the leaf node owned by the user being removed. Of course, we ensure that the user being removed actually owns the node in question. The REMOVEUSER algorithm works from top-to-bottom by examining nodes in the path from the root to the destination node. As it traverses this path, if a black node is reached that is owned by the user to be removed, then the removal succeeds and the node becomes available (and painted white) as shown in figure 7. As we traverse down the tree, we decrease the grey-count value of each grey node by one. If the grey-count of a node drops from two to one, then we know that after removing this user, there will be only one user left in the sub-tree; if this is the case, then we should *promote* this remaining user as shown in figure 8. Promotion involves moving the ownership of the sibling node being deleted to the highest available sub-tree with no ownership conflicts; the highest node will be this node whose grey-count just went from two to one.

If a node is colored black, then the node and all its children are owned by the same user. As stated previously, all nodes in the sub-tree of a black node are also colored black. As a result, when we unlock a node w and color it white, we also unlock and color white all nodes in the sub-trees of w.

The following visualizations (figures 7 and 8) demonstrate the unlock effect on the two possible configurations. Note that since we are performing a REMOVEUSER operation on node w, it must be black; additionally, node t must be grey, and node v may not be white. Consequently, the following two cases shown below are the only ones possible.

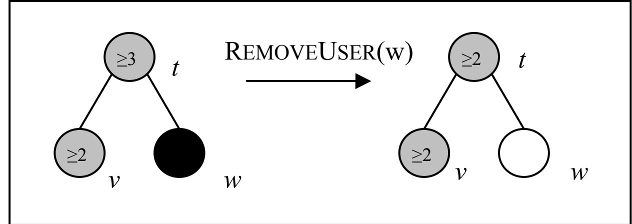

Figure 7: Case 1 of the REMOVEUSER operation

In the above case (figure 7), we cannot promote an owner of a node rooted at v because there must exist more than one owner (or v would be colored black). This w remains white, and t and v remain grey.

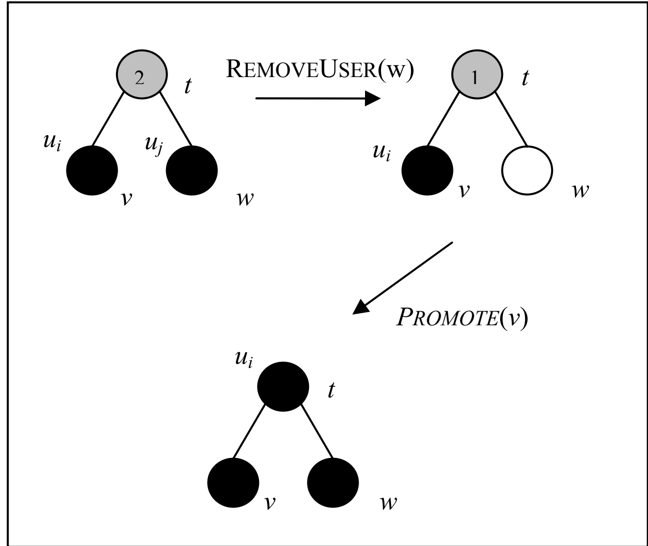

Figure 8: Case 2 of the REMOVEUSER operation

Note that in the above case (figure 8), it is possible (and advantageous) for the user u_i who owns the lock on node v to have his/her ownership "promoted" to node t such that user u_i now owns node t (and as a result owns nodes v and w). This is possible since the user u_j who gave up node w no longer conflicts with user u_i for ownership of t.

In the process of promotion, we must determine which node should be promoted. We avoid having to track why a node w is colored grey (i.e., maintaining a set of black-colored nodes within the sub-tree rooted at w) and

consequently which node should be promoted because we maintain an integer grey-count. The process of promotion is simplified in that when the grey-count is reduced from 2 to 1, we know a promotion should occur. It is an interesting fact of our data structure that the node to be promoted must be a sibling of the node being removed. There does exist a special case in which grey-count is artificially-inflated due to a failed insert; in this case, the grey-count may fall to 1 and the promotion may involve the node being removed (as this "removal" is actually repairing a failed insert); alternatively in the special case, the grey-count may fall to 0 and the promotion does not occur and the node w is painted white. These cases are handled in the algorithm in figure 9.

```
REMOVEUSER(w, u_i)
    if w.owner = u_i
        then RECURSEREMOVE(ROOT, w, u_i)

RECURSEREMOVE(n, w, u_i)
    if n.color = black and n.owner = u_i
        then RELEASEOWNER(n)
    else if n.color = grey
        then n.greyCount = n.greyCount – 1
        if n.greyCount = 1 and w.sibling.color = black
            then
                SETOWNER(n,                w.sibling.owner,
w.sibling)
        else if n.greyCount = 1 and w.color = black
            then SETOWNER(n, w.owner, w)
        else if n.greyCount = 0
            then RELEASEOWNER(n)
        else RECURSEREMOVE(NEXTINPATH(n,w), w, u_i)

RELEASEOWNER(w)
    w.color = white
    w.owner = NIL
    w.originalRequest = NIL
```

Figure 9: Removal of a User

In the case where promotion is possible, the node to be promoted (v) must be the sibling of the node owned by the user being removed (w). More formally:

Lemma: If in the process of removing a user u_w at node w we arrive at an ancestor node t with a grey-count of 2 (before the removal of u_w at w), there must exist exactly 2 nodes in the sub-tree rooted at t that are siblings and are colored black.

Proof: Assume for the sake of contradiction that this is not the case; then there must exist a situation where the black node w to be removed either has a white sibling, a grey sibling, or no sibling. If the sibling to w is white, then the parent of w should already be painted black and we would not need to examine w for removal (i.e., we should remove the higher node). Likewise, if w has no sibling, then the parent of w should be black and we would not need to examine w for removal (i.e., we should remove the higher node). If the sibling to w is grey, then there must exist at least two nodes that are painted black as child nodes to this sibling node; but this is not possible or the grey-count of t would have to be greater than 2. Thus, the sibling to w must be black. Further, there can exist no other nodes in contention for promotion with this sibling node in the sub-tree rooted at t or the grey-count of t would not be 2.

A special case does exist when promotion occurs after failed inserts (and the resulting artificially-inflated grey-counts exist); in this case, it could be possible that when reducing the grey-count to the proper values, the grey-count is reduced to 1 or 0. In the case of a reduction to 1, either the node that failed to acquire or it's sibling must be promoted (whichever is black). In the case of a reduction to 0, no promotion is possible and the grey node should be painted white. This is implemented in figure 9.

The following (figure 10) shows the execution of REMOVEUSER(w, u_1) and the effects on node coloring, grey-count and ownership. Note that u_2 becomes the owner of the entire sub-tree rooted at v because u_1 is no longer in contention for any of those nodes.

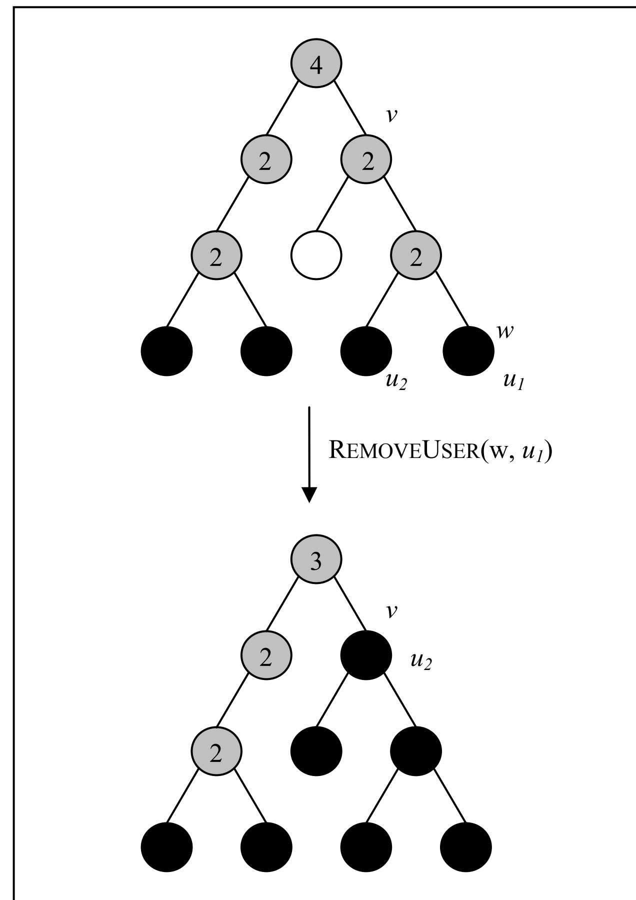

Figure 10: REMOVEUSER (with multi-level promotion)

Performance Analysis

It is critical to observe that it is not necessary to re-color the nodes below a black node to be all black or to re-color

the nodes below a white node to be all white. This holds true because once a black or white node is reached, the algorithms presented here look no further down the tree; as our coloring scheme for the tree originally claimed, all sub-tree nodes of a black node are black, and all sub-tree nodes of a white node are white. This is a logical coloring, and no work is incurred to ensure this coloring (thus you will not see any code to ensure proper color adjustment of sub-tree nodes in the algorithms presented above).

It is permissible for the value of the color attribute of a node to be "incorrect" yet be logically correct with respect to our algorithms since the INSERTUSER and REMOVEUSER algorithms both work from top to bottom in the tree structure; consequently, it is not possible to reach an "incorrectly" stored color value in a node *w* without having first traversed the correctly-colored ancestor node *t* that is responsible for logically coloring *w* property (either black or white). For example, if the value stored in the attribute *w.color* is grey, but *t* is an ancestor of *w* and *t.color* is white, then logically *w.color* must be white.

Proper coloring of nodes in the top-to-bottom access of the algorithms presented is ensured by the SETOWNER and RELEASEOWNER functions. These two functions guarantee that the coloring, ownership, and originally-requested leaf node are properly maintained.

The ability to make this color assertion logically based upon node ancestry via the top-to-bottom INSERTUSER and REMOVEUSER algorithms saves computation time and makes the algorithms more efficient.

Both algorithms access the tree from top to bottom; INSERTUSER makes at most two passes down the tree, and REMOVEUSER makes one pass down the tree. As a result, both INSERTUSER and REMOVEUSER run in O(*h*) where *h* is equal to the height of the tree.

As shown previously, promotion of ownership from node *w* to a higher node in *w*'s ancestry occurs when a user who owns the sibling of *w* leaves and removes contention with *w*. As we previously proved, the node to promote must be a sibling of the node being removed, or in the special case of promotion after artificially-inflated grey-counts, the node to promote may be *w*'s sibling; alternatively in the special case, no promotion may occur at all. In any of these three scenarios, promotion occurs in O(1).

RELATED WORK

Other researchers in CSCW have adopted operational transformations and merging to ensure consistency control. [13] argues that continuous coordination of development is critical in avoiding resynchronization (merging) and isolated development while being scalable and user-centric. [12] presents a multi-version single-display (MVSD) technique to resolve conflict between concurrent updates and ensuring consistency maintenance and group undo. [11] argues for decoupling the concurrency control mechanism from the editing system and points out that users in a collaborative environment often edit disparate sections of the shared document. Their approach implements an implicit turn taking mechanism and a "relaxed WYSIWYG" and operational transformations to avoid inconsistencies among replicas of the shared document. Our approach avoids the need to merge and transform operations via multi-granular locking. As [8] states, "users may prefer 'prevention' to 'cure'", and our system takes this approach in avoiding the potentially considerable effort to merge disparate versions of the document. Since we utilize pessimistic locking, no transformations or merges are needed.

Other research in CSCW and CES have suggested multi-granular locking as a means of improving concurrent access. [10] raises the concern from a software engineering perspective of increasing parallelism of development and improving awareness through optimistic concurrency control. [3] discuss the need to overcome the development bottlenecks associated with file locking in SCM (software configuration management). The complexities of such systems is hypothesized to be 1-2 orders of magnitude greater than file-level locking systems. [7] supports the idea of fine-grain locking in hierarchical documents; many documents contain semantic structure that would allow for such fine-grain locking (for example, software code and word processing documents).

Most notable in our system is the fact that the locking scheme is transparent to the user. This approach was advocated by [5]. [4] and [6] also point out that requiring users to manage the maintenance of fine-grain locking is onerous and prohibitively costly, outweighing any benefit of such increased concurrency. This is in contrast to other systems such as MACE (which required the user to specify the locked region) and DistEdit (which attempts to lock the smallest region possible) as described by [6]. The approach we advocate in this system supports and reflects the dynamic nature of collaborative interactions and the need for flexibility in the control and consistency protocols employed [14].

Our work is in line with the heterogeneous, open-systems approach advocated by [2] in that our algorithm is not dependant upon any specific client editor. [15] advocates using Web Services to achieve interoperability and heterogeneity in a collaborative system. Our algorithms presented here work well within open-systems, Web Services framework in adversarial/competitive and congenial collaborative environments.

Most recently, The NetBeans Collaboration Project [1] demonstrates how shared source code can be edited in real time (synchronously) to perform code reviews (supported by instant messaging). The NetBeans system is used for code review processes (where little multiple real-time editing is intended) whereas our system supports synchronous editing throughout the document's lifecycle.

CONCLUSION AND FUTURE WORK

In this paper, we have described a scheme to support deadlock-free hierarchical locking in a real-time collaborative editing system. Our approach has been positioned in relation to other transformation-based (optimistic) and lock-based (pessimistic) collaborative system control schemes. As a result of the hierarchical nature of our locking policy, the approach described in this paper may be considered a hybrid of optimistic and pessimistic as it enables the flexibility and increased concurrency of an optimistic approach while maintaining the consistency (and as a result avoids the merge problem) of a pessimistic approach.

The algorithms presented to insert and remove a user act in a top-to-bottom fashion and ensure that the algorithms avoid deadlock and may be executed efficiently and concurrently. Our policy to maximize the sub-tree locked by a user minimizes messaging and communication costs among processes within the system.

This paper also contributes to the field of CSCW in that the algorithms presented and the multi-granular locking policy advocated are applicable to any document that is hierarchical in nature and contains semantic information that facilitates document decomposition into subsections.

While this paper describes our work in ensuring deadlock-free access to multi-granular locking on a shared document, the algorithms presented here do not address the issue of modification of the tree structure itself. The work presented here assumes that the tree's structure is consistent; as a result, modification to existing sections of an document is supported, but deletion of existing sections and insertion of new sections is not yet addressed We thus intend to expand this work and address the issues of inserting a new section into the document, splitting a section into two sections, and deleting a section from the document.

Additionally, this paper discusses caching and inter-process communication minimization, but the cache update/flush policies of this system merit further study.

REFERENCES

1. Netbeans Collaboration Project. http://collab.netbeans.org. Sun Microsystems. Viewed August, 2005.
2. V. Bharadwaj and Y.V. R. Reddy. A Framework to Support Collaboration in Heterogeneous Environments. SIGGROUP Bulletin. Vol 24, No 3. December 2003. pp. 103-116.
3. M. Chu-Carroll, J. Wright, and D. Shields. Supporting Aggregation in Fine Grained Software Configuration Management. SIGSOFT 2002/FSE-10. Charleston, SC. November 2002.
4. M. Chu-Carroll and S. Sprenkle. Coven: Brewing Better Collaboration through Software Configuration Management. SIGSOFT 2000. San Diego, CA. November 2000.
5. C. Ellis and S. Gibbs. Concurrency Control in Groupware Systems. ACM SIGMOD Conference on Management of Data, pages 399-407. ACM Press, May 1989.
6. S. Greenberg and D. Marwood. Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface. ACM Conferences on Computer Supported Cooperative Work, pages 207-217. ACM Press, Nov. 1994.
7. B. Magnusson, U. Asklund, S. Minör. Fine-Grain Revision Control for Collaborative Development. Proceedings of ACM SIGSOFT'93. Los Angeles, CA. December 1993.
8. J. Munson and P. Dewan. A Concurrency Control Framework for Collaborative Systems. Proceedings Computer Supported Cooperative Work, pages 278-287. ACM Press, Nov. 1996.
9. V. Nageshwara Rao and Vipin Kumar. Concurrent Access of Priority Queues. IEEE Transactions on Computers. Vol 37, No 12. pp. 1657-1665. 1988.
10. C. O'Reilly. A Weakly Constrained Approach to Software Change Coordination. 26th International Conference on Software Engineering. 2004.
11. H. Shen, C. T. Cheong, and C. Sun. CoStarOffice: Toward a Flexible Platform-independent Collaborative Office System. IWCES'04.
12. D. Sun, S. Xia, C. Sun, and D. Chen. Operational Transformation for Collaborative Word Processing. CSCW'04. Chicago, IL. November 2004. CHI Letters. Vol 6, Issue 3. pp. 437-446.
13. A.van der Hoek, D. Redmiles, P. Dourish, A. Sarma, R. S. Filho, and C. de Souza. Continuous Coordination: A New Paradigm for Collaborative Software Engineering Tools. Workshop on Directions in Software Engineering Environments. ICSE'04.
14. Y. Yang and D. Li. Separating Data and Control: Support for Adaptable Consistency Protocols in Collaborative Systems. CSCW'04. November 2004. CHI Letters. Vol 6, Issue 3. pp. 11-20.
15. M. Younas and R. Iqbal. Developing Collaborative Editing Applications using Web Services. IWCES'03.
16. B. P. Zeigler and H.S. Sarjoughian. Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models. Technical Document, University of Arizona. 2003.