

# An Efficient Synchronous Collaborative Editing System Employing Dynamic Locking of Varying Granularity in Generalized Document Trees

Jon A. Preston and Sushil K. Prasad

*Department of Computer Science*

*Georgia State University*

*Atlanta, GA USA*

*{jon.preston@acm.org, sprasad@gsu.edu}*

**Abstract**—The primary goals in a synchronous collaborative editing system (CES) involve ensuring a high level of concurrent access while maintaining the properties of the CCI model. We revisit the idea of applying lock-based concurrency control algorithms to manage access to a shared document; this research overcomes the traditional problem of reduced concurrent access inherent in pessimistic concurrency control by dynamically managing the size of the portion of document locked based upon user demand, scaling up and down the lock granularity to accommodate user write requests. We present algorithms to efficiently maximize concurrent access while utilizing caching techniques to reduce communication costs. We also discuss how OT and other optimistic concurrency control techniques may be incorporated within our approach – leveraging best practices of both techniques. We conclude with an analysis of the communication and computational costs of our approach and compare these costs to costs incurred using OT-based concurrency control.

## I. INTRODUCTION AND MOTIVATION

Imagine a scenario in which a geographically distributed team can work together, sharing ideas, collaboratively editing a shared document in real-time, and interacting as closely and productively as a team of workers within the same room. This is one of the goals of the field of Computer Supported Collaborative Work (CSCW) and in particular the subfield of Collaborative Editing Systems (CES). Research in this field have made great contributions to achieving the goal of providing synchronous (real-time), concurrent access to a shared document. Most notable are algorithms [7][4] that work to achieve a high level of concurrent access through optimistic concurrency control and Operational

Transformation (OT) or similarly intelligent techniques to merge changes on other users' copies while ensuring consistency, causality-preservation, and intention-preservation – the widely accepted CCI model [15].

Motivated by configuration management systems, such as Revision Control System (RCS), that employ pessimistic concurrency techniques to avoid the problems of merging conflicting, concurrent changes made to shared documents, we revisit the idea that pessimistic locks offer potential CES research opportunities. We recognize that using pessimistic locks reduces the concurrent access, thus the CES community has lately focused on the optimistic approach that requires OT and similar merging techniques; but what if the locks were dynamic and could grow and shrink automatically? By dynamically managing lock granularity, we may allow maximum concurrent access among many authors and avoid the need to merge disparate versions of the shared document, which can not be resolved occasionally.

Thus, we propose a set of CES algorithms that coordinate synchronous access to a shared document using locks that grow and shrink dynamically to accommodate users' write requests; this system also allows multiple readers within the same section of a document. The system caches changes locally until they must be distributed to other users, thus communication costs can be minimized. The data structures and algorithms presented in this paper are significant extensions of our previous work which was limited to algorithms for binary-tree based document representations [12] (a version of [12] is available at <http://tinyurl.com/eaye2>). This paper (i) generalizes the algorithms for n-ary document trees, which is closer to how documents are organized, and (ii) provides an experimental examination of communication costs of coordinating the locks and managing the changes cached on local clients' machines.

**Table 1 - User Actions and CES Tree Events**

<i>User Action</i>	<i>CES Tree Events</i>
Enter the CES	<ul style="list-style-type: none"> <li>Place user as a reader in the default section of the document</li> </ul>
Exit the CES	<ul style="list-style-type: none"> <li>Remove the user from the CES and flush the user cache</li> </ul>
Modify content within section A	<ul style="list-style-type: none"> <li>OBTAINLOCK for section A</li> <li>If not successful, deny the edit</li> </ul>
Move from section A to section B	<ul style="list-style-type: none"> <li>RELEASELOCK on section A</li> <li>Place user as a reader in section B</li> </ul>
Delete section A	<ul style="list-style-type: none"> <li>OBTAINLOCK for section A</li> <li>If successful, remove section A from tree</li> </ul>
Combine section A and section B	<ul style="list-style-type: none"> <li>OBTAINLOCK on section A</li> <li>OBTAINLOCK on section B</li> <li>If either fail, release any successfully obtained lock and deny the request</li> <li>Else merge sections A and B in the tree (removing section B and RELEASELOCK on B)</li> </ul>
Split section A into sections A and A'	<ul style="list-style-type: none"> <li>OBTAINLOCK on section A</li> <li>If not successful, deny the edit</li> <li>Else create a new node A' as a sibling of A, move content from A into A'</li> </ul>

We provide the overview of the data structures in Section 2, discuss each algorithm and focus on the improvements to support the n-ary tree representation of the document in a CES session in Section 3, and formally discuss and analyze communication costs of our architecture in Section 4. Finally, we provide a discussion of our future work and conclusions.

## II. OVERVIEW OF DATA STRUCTURES AND ALGORITHMS

### A. Users' Actions within the System

We assume that issues such as awareness of where other users in the CES are reading or writing as well as support for communication among users is available in our CES. Thus, our research will not address these HCI issues as such issues are covered elsewhere in the CES literature. Rather, we focus on the underlying data structures and algorithms to manage locks dynamically (transparent to the user).

We identify a set of user actions within a CES and map these actions to events within our tree-based system. These actions and associated events in our system are listed in Table 1. Note that if a lock is requested (via the OBTAINLOCK event) and the user already owns the lock, then no server request/communication is required. It is only when a user attempts to edit a section without having previously edited that section (i.e. the section is not owned by the user) that a request for the lock is required.

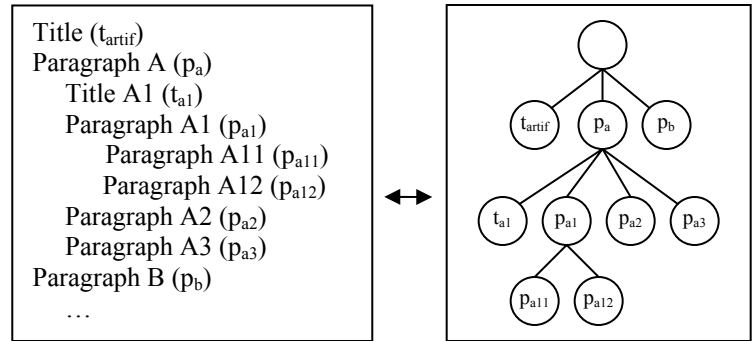
Traditionally, pessimistic (lock-based) CES concurrency control approaches have been avoided due to latency issues; if responsiveness is of concern and requests for modification are not

sufficiently responsive, then a semi-optimistic concurrency control [3] may be adopted wherein rollbacks are performed in the case of a lock request failure.

### B. Representing a Document via a Tree

Each element of a document may consist of content (text) and sub elements (sub sections). For example, Section 2 of this paper is entitled "Overview..." and consists of text as well as sub-sections. Because any section of a document may contain any number of text elements (paragraphs, sentences, etc.) and may contain any number of sub-sections, we generalize our previous algorithms [1] for inserting and removing locks from the collaborative space to work within an n-ary tree data structure that is representative of a shared document. Figure 1 demonstrates how such a tree may represent a document; note that all the document's content is stored in the leaves of the tree and non-leaf nodes are used to denote structure.

Once established, the document tree is utilized to



**Figure 1. Mapping from a document to an n-ary tree**

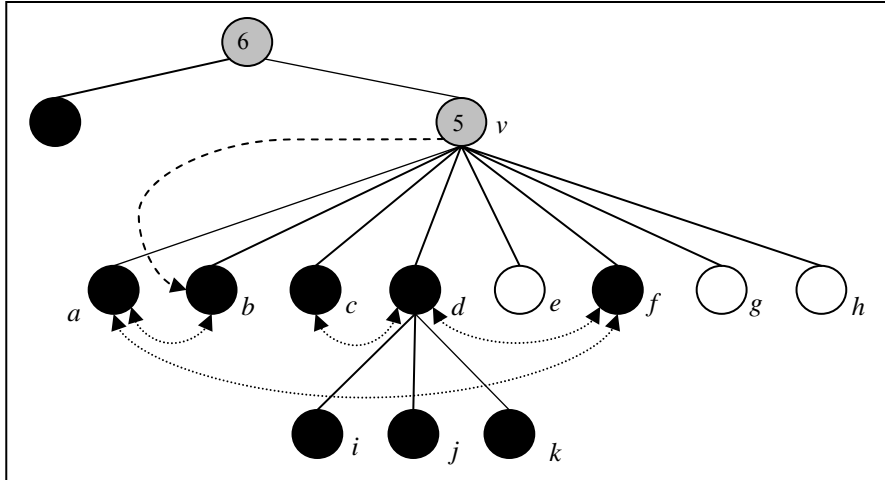


Figure 2. Black sibling references enable quick promotion

manage ownership of subsections within the document. Rather than locking the entire document, lock granularity is adjustable, ranging from the entire document (ownership marked at the root of the tree) to an atomic level (ownership marked at a leaf node in the tree). The size of a subsection is not specified within our algorithms, thus it is scalable to accommodate the semantic structure of the document being edited, similar to [11].

Rather than blocking other users from editing, lock granularity is adjusted via demotion of the lock down in the tree until the conflict among users is resolved. Additionally, when a user leaves a section of the document and makes it available to other users, conflict among users is potentially reduced; as a result, our algorithm automatically promotes the lock to a higher level within the document tree – maximizing the amount of the document owned for the remaining user. The OBTAINLOCK and RELEASELOCK operations are detailed in Figure 10. These algorithms traverse the document tree in a top-down fashion and are guaranteed to be deadlock free.

### C. Node and Path Notation

The NEXTINPATH( $n, w$ ) function of the binary-tree based algorithm assumes that a binary-encoded identifier was sufficient to uniquely identify a path from  $n$  to  $w$  as defined in [14]. Since we do not mandate a binary tree structure, we must support a new mechanism for correctly identifying the path to the desired element of the  $n$ -ary tree. We do this by extending the binary identifying path of [14] to be  $n$ -ary as follows.

First, let the identifier  $E$  denote the path  $p$  from the root to a vertex  $v$ ;  $E$  consists of a string of  $m$  entities, where  $m$  is the depth of  $v$ . If the root is desired, then  $E = ""$  (empty string) since the root is at depth 0 ( $m=0$ ).

Each entity in  $E$  specifies which sub-tree to follow in the path to  $v$ . If the path  $p$  contains the edge from vertex  $v_k$  to the  $i^{\text{th}}$  child/sub-tree of  $v_k$  (where  $v_k$  is a vertex at depth  $k$ ), then the  $(k+1)$  entity of  $E = i$  (i.e. traverse into the  $i^{\text{th}}$  sub-tree of  $v_k$ ).

Thus we uniquely identify each vertex in the tree, and the identifying string for each vertex defines a path from the root to the vertex that can be found in  $O(1)$  time. Using Figure 2 as an example, a path from the root to vertex  $i$  may be defined by  $E_i = "241"$ , and the path from the root to vertex  $f$  may be defined by  $E_f = "26"$ .

### D. Maintaining Node Coloring, Grey Count, and Black Siblings

Each node in the document tree maintains a color (white, black, or grey) to denote whether it is available, currently being written to by another user, or if two or more users are editing sub-trees, respectively. Ownership (black coloring) of a vertex  $v$  by user  $u$  implies that  $u$  owns  $v$  and the sub-tree rooted at  $v$ , and is the only user that may edit node  $v$  or its sub-tree. Each node  $n$  in the tree maintains a numeric value that denotes how many nodes in the sub-trees of  $n$  are colored black. This is defined as the *grey-count* of the node  $n$ . This value is useful in determining if the node can be colored white or grey when a request to delete a user occurs and promotion is enabled (as explained later).

A grey node  $v$  maintains references to the node's children (sub-trees); additionally, if there exists at least one black child node of  $v$ , then  $v$  also maintains a reference to the first black child node. The black child nodes of  $v$  ( $b_1, b_2, \dots, b_k$ , where  $k =$  number of black child nodes of  $v$ ) are linked together using a doubly-linked list. As an example, the black children of  $v$  are  $\{b, a, f, d, c\}$ .

### E. Viewing and Editing Sections of the Document

The dynamic lock management described in Section III focuses on granting a user exclusive access for writing to the section of the shared document. In addition to supporting dynamic, exclusive writer locks, the system also supports multiple, simultaneous readers for a section of the document. It is permissible to allow multiple users to view the changes being made by another user, and thus the  $n$ -ary tree used to manage the

write locks of the document is also used to manage the viewing positions of all users within the document.

For example, if a collaborative editing session included five users,  $U = \{u_1, u_2, u_3, u_4, u_5\}$ , where  $u_1$  was editing Section 1,  $u_2$  was editing Section 2,  $u_3$  and  $u_4$  were viewing Section 1, and  $u_5$  was viewing Section 2, this would be stored in the n-ary tree, shown in Figure 3.

### F. Maximizing the Lock Granularity and Caching Local Changes

It is advantageous to maintain a lock on the largest sub-tree that is permissible; by maximizing the sub-tree that any user owns, we minimize the communication costs of the system by utilizing caching. 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. 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  $n_i$ . 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 remains local to  $u_i$ .

## III. DETAILED ALGORITHMS FOR GENERALIZED DOCUMENT TREES

Figure 10 contains the key steps of our algorithms. All algorithms work from top-to-bottom via handshake locks to avoid deadlock; since we maintain a reference from the first black sibling up to its parent, this handshake lock must hold two nodes at a time (a node  $v$  and a child of  $v$ ). Thus as these algorithms traverse down the tree, the handshake lock will obtain a lock on  $v$ , then on  $u$ , where  $u$  is a child of  $v$ ; it is not necessary to release the lock on  $v$  immediately, but before obtaining a lock on a child of  $u$ , the lock on  $v$  must be released. The OBTAINLOCK and REMOVELOCK algorithms run in  $O(d)$  time where  $d$  is the depth of the document tree (i.e.,  $d$  is the number of hierarchies in the document tree). For most documents,  $d$  is small; for example, if a document was structured into sections, subsections, subsubsections, paragraphs, sentences, and words, then the document tree would have a height of 7 (including the root).

### A. Obtaining a Lock

A user requests a section of the document to which he wants to write, and the system attempts to obtain a

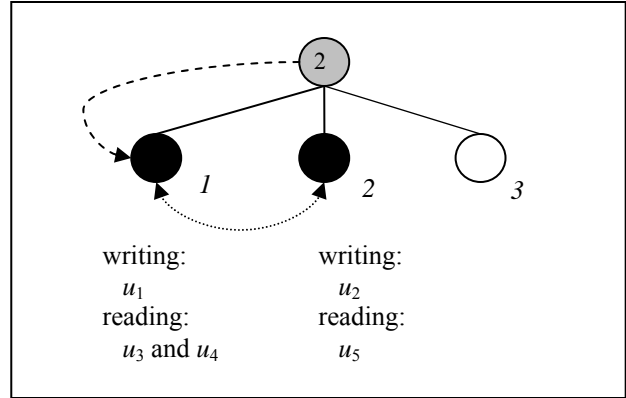


Figure 3. Tracking readers and writers

lock on that section of the document. The OBTAINLOCK 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 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 an "atomic" node, then we can't demote any further, and the insert operation fails (i.e., edit request is denied). Alternatively, if desired, optimistic concurrency control techniques such as OT may be employed at this atomic level; by keeping a list of writers, a selective multicast of all changes within this atomic section could be made to all writers, limiting the computation and communication cost to a subset of all users within the smaller section of the document.

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 REMOVELOCK method (which reduces the grey-count of the grey nodes in the path from the root to the destination node by one).

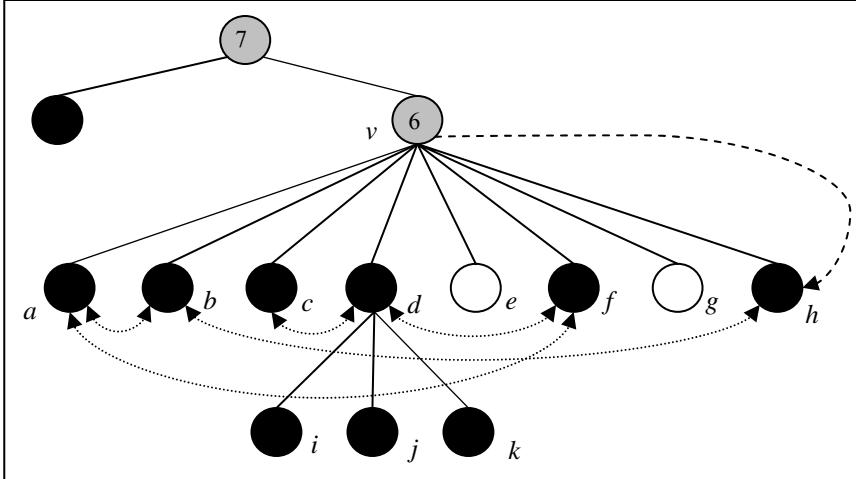


Figure 4.  $OBTAINLOCK(u_1, h)$  -  $h$  is added into the black sibling list

When promotion is required during a  $REMOVELOCK$  action, we must have a way of efficiently resolving which sibling should be promoted (the remaining sibling); a brute force method could traverse all siblings until the remaining black node is found, but this is inefficient and requires  $O(n)$  work where  $n$  is the number of siblings (the maximum branching factor of the tree). Alternatively, we can maintain a back-sibling and forward-sibling reference for each node, linking the black siblings together in a list to maintain a subset of all the siblings; this subset consists of all nodes colored black (e.g.,  $a, b, c, d,$  and  $f$  as shown in Figure 2).

### B. Updating the Sibling List upon $OBTAINLOCK$

When an  $OBTAINLOCK$  request is successfully fulfilled, we have two cases – (1) there was no contention and no demotion and a white node is painted black, or (2) there was contention and this contention is resolved via demotion and by adjusting node coloring.

In the first case (no demotion), a white node must be painted black, and the newly-painted black node must be added into the black sibling list of the grey, parent node. If the document state was as represented by Figure 2, assuming  $h$  was to be painted black, i.e.  $OBTAINLOCK(u_1, h)$  was invoked, then Figure 4 shows the result of painting  $h$  black and adding  $h$  as the head of the sibling list.

In the case where an  $OBTAINLOCK$  operation requires the lock contention be resolved via a demotion of the lock, we must

adjust the black sibling list to reflect the demotion of the lock. Additionally, we must link the two nodes now painted black.

Using Figure 4 for example, if  $OBTAINLOCK(u_2, k)$  was invoked and node  $d$  had previously been locked when  $u_1$  had requested node  $i$  (i.e. node  $d$ 's original request reference is  $i$ ), then the  $u_1$ 's lock on node  $d$  will be demoted to node  $i$ , and then  $u_2$  will acquire a lock on  $k$ . When this occurs, node  $d$  should no longer be in the black sibling list of its parent, node  $v$ . Thus we modify the  $OBTAINLOCK$  algorithm to remove this node whose lock was demoted from the black sibling list by joining the adjacent siblings of the node. Additionally, black sibling links must be established for the two black nodes that result from the demotion (nodes  $i$  and  $k$  in this example). The result for this example would be that node  $c$  and node  $f$  are now joined and node  $i$  and node  $k$  are now joined, as shown in Figure 5.

### C. Removing a Lock

The  $REMOVELOCK$  algorithm works from top-to-bottom via handshake locks to avoid deadlock. As the path from the root to the node to be released is traversed downward, the grey-count for all nodes painted grey is decreased by one until a grey node with a grey-count of one (after decrementing) is encountered; when this occurs, a promotion is needed to ensure that the sibling of the to-be-unlocked node owns the largest sub-tree possible. This is the same behavior as the binary-tree

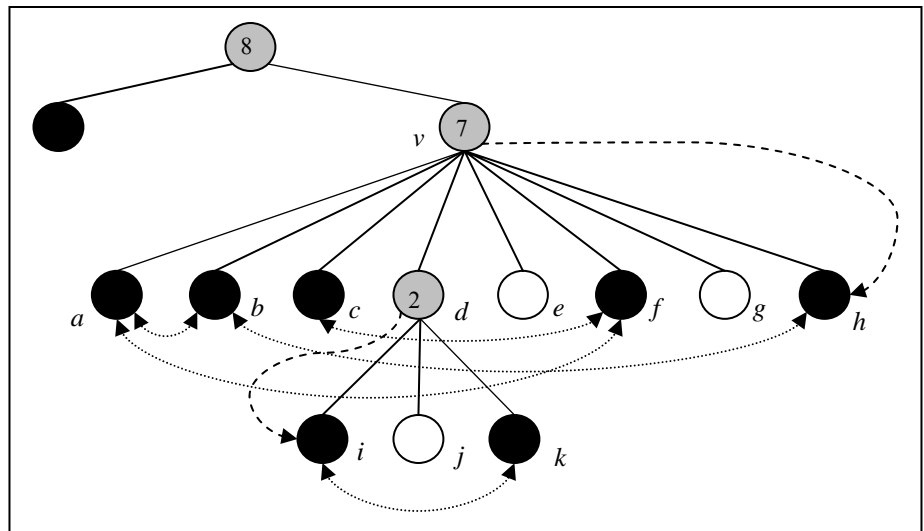
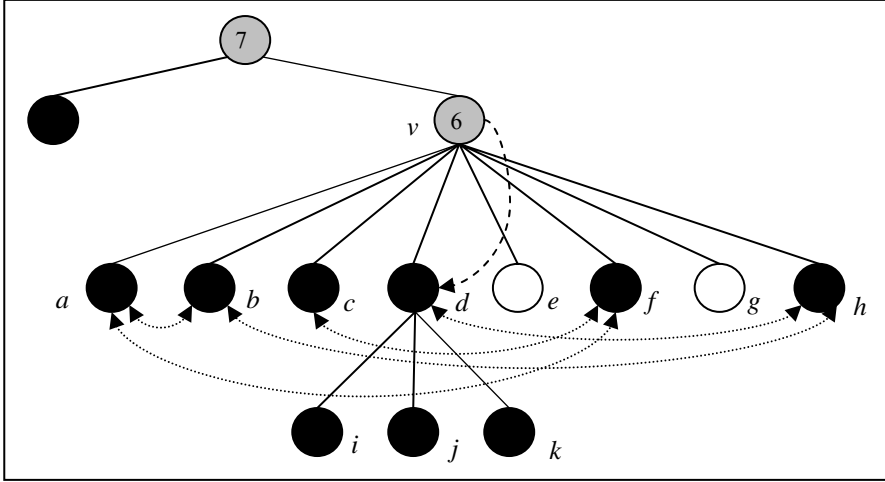


Figure 5.  $OBTAINLOCK(u_2, k)$  - the lock on node  $d$  is demoted to node  $i$



**Figure 6. REMOVELOCK( $u_1, i$ ) -  $u_2$  lock on node  $k$  is promoted to node  $d$**

based REMOVELOCK algorithm.

The only modification that must be made to accommodate an n-ary tree is that when promotion occurs, then the newly-promoted node  $v$  must be added into the black-sibling list of  $v$ 's parent.

#### D. Updating the Sibling List upon REMOVELOCK

When an REMOVELOCK request is fulfilled that necessitates a promotion, the node who's grey count has been reduced to one must be painted black and must be added into the black sibling list of the grey, parent node. Assuming in Figure 5 that the lock on node  $i$  was to be removed (i.e. REMOVELOCK( $u_1, i$ ) was invoked), then Figure 6 shows the result of promoting the lock held on node  $j$  to node  $d$  and adding node  $d$  into sibling list.

The order that the black sibling nodes appear in the list is not significant as we only use this list to maintain adjacent siblings so that we know immediately which sibling to promote. Notice in the example shown in Figure 5, if the lock to be removed is associated with node  $i$ , then we know immediately without incurring a search cost that the lock associated with node  $k$  is the node to promote because node  $i$  and node  $k$  are marked as black siblings. Since promotion will only occur when there are two siblings (one of which no longer requires a lock and the other is associated with the lock to promote) order among the black siblings is not significant within the list. Consequently, when promotion does occur, we can simply place the

node associated with the newly-promoted lock at the front of the black sibling list. Assuming the configuration shown in Figure 4, if a REMOVELOCK( $u_1, i$ ) is invoked, then the lock  $u_2$  has on node  $k$  should be promoted to node  $d$ . Node  $d$  is then added into the front of the black sibling list. The result of this promotion is shown in Figure 6.

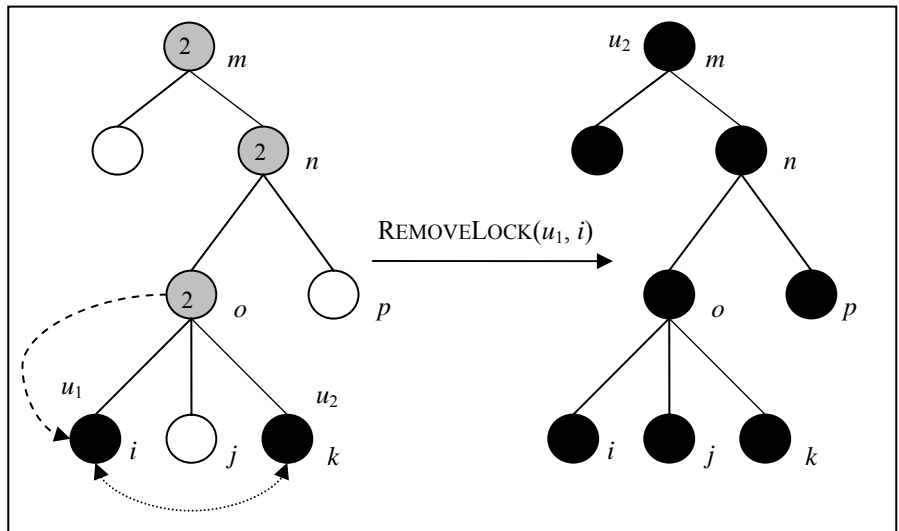
**Multi-level Promotion:** It is possible for a situation to arise in which removal of a lock removes contention and the remaining user should be promoted through multiple levels within the tree. Figure 7 shows such a scenario, and as demonstrated, our algorithm

handles this, promoting the remaining lock to maximize the portion of the document owned by the remaining user.

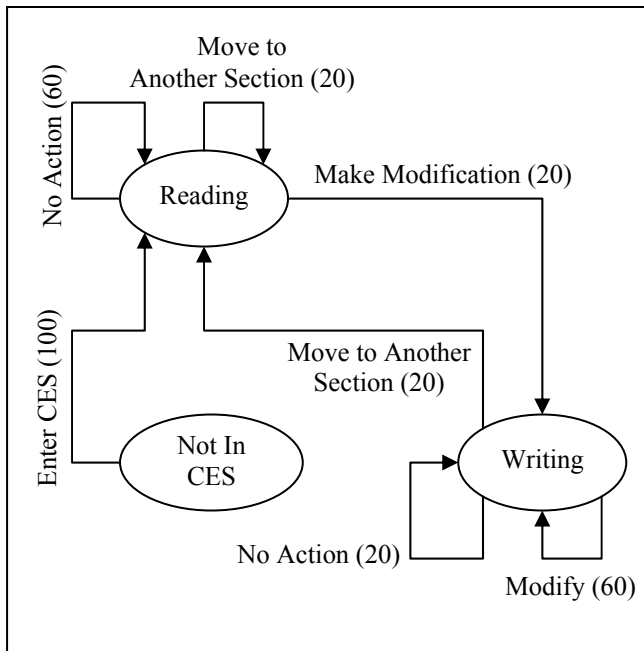
#### IV. COMMUNICATION COST ANALYSIS

Past and present research in CES focus on the computational cost of ensuring the CCI model and assume that distributed views of the shared document are updated at the atomic user action level (i.e. character insertion and deletion); we refer to [7], [4] and [15] as exemplars. These OT-based systems send a network message (packet) upon each edit/write of any user to all other users within the CES (via multicast). In contrast, our system caches changes locally and only distributes these changes when:

1. The writer makes a change and there are readers



**Figure 7. Promotion across multiple levels is permissible**



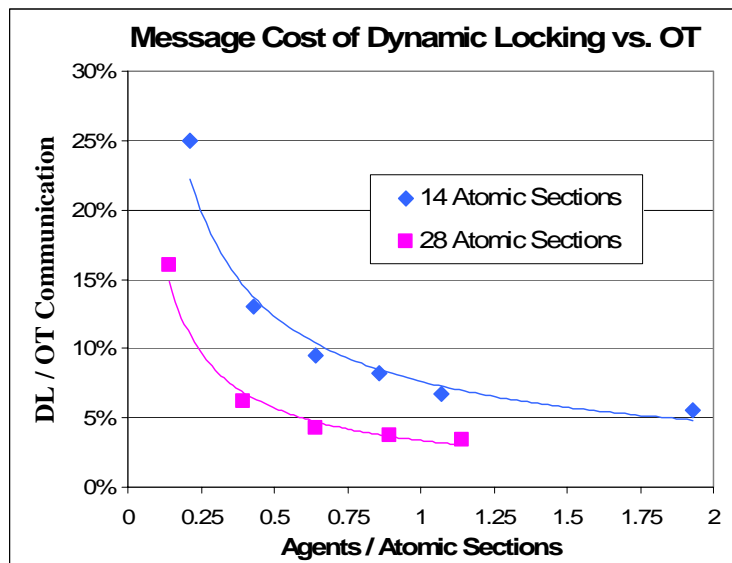
**Figure 8. Agent Behavior States and Actions**

- within the subsection, selectively multicasting to all readers within the subsection
- 2. Another user enters a document section as a reader, sending this cached subsection’s contents to the new reader
- 3. Demotion occurs and the cache on the now un-owned section(s) must be flushed, sending the modified subsection’s contents to the server
- 4. A user changes position within the document or leaves the CES, releasing the lock and sending the subsection’s contents to the server.

In addition to the communication being sent among users as a result of the events 1-4 listed previously, there is also a communication cost incurred to keep the clients aware of which section of the document they own. OBTAINLOCK and RELEASELOCK requests are passed to the server based upon the users’ actions (see Table 1). Because each client tracks which portion of the document that he owns (so as to cache changes within any subsection owned), any client whose lock has been modified by the server (as a result of a promotion or demotion) must be notified of the lock’s modification. Note that race conditions are not possible among the clients’ local lock data because only the server distributes these updates.

To validate the communication effectiveness of our dynamic locking algorithms, we implemented the algorithms and then ran discrete-event simulations which varied the number of users/agents as well as varied the structure of the shared document to capture communication and computation costs. Figure 8 illustrates the agent behavior states and actions modeled; the probability of the action being initiated at each time slice is denoted in parenthesis along each transition. These action probabilities are useful to obtain a mixture of reading and writing events within the simulations. Each configuration of the simulation was run such that each agent generated 1000 actions based upon the state diagram (Figure 8). To more clearly compare communication costs between our dynamic locking approach and an OT approach, within these simulations we do not allow for multiple writers.

The results from these simulations and the comparison to the OT-based communication costs are provided in Table 2 and Figure 9. The communication cost utilizing our approach is significantly less than the communication cost incurred by an OT-based system, and the communication cost improvement increases as the ratio of agents to sections within the document increases (as the collaboration becomes more “dense”). It is also important to note that lock/write failure is possible in the dynamic locking, but for all simulation scenarios in which the number of agents was less than half the number of document sections, no less than 64% of write attempts were successful. Of course, these write failures may be eliminated by incorporating OT at the atomic level within our document tree and using selective multicast among all writers within the shared subsection.



**Figure 9. Significant Communication Cost Improvement of DL over OT as a function of Agent/Section**

**Table 2 – Simulation Configurations and Communication Costs**

Configuration			Communication						
# Agents	# Atomic Sections	Write Events	Dynamic Lock (DL) Messages				OT Messages	DL / OT Messages	DL Write Success Rate
			Client to Server	Server to Client (P/D)	Writer to Readers	TOTAL			
3	14	770	88	242	61	391	1540	25%	74.3%
6	14	1227	122	428	263	813	6135	13%	64.4%
9	14	1760	121	505	708	1334	14080	9.5%	61.6%
12	14	2004	144	615	1050	1809	22044	8.2%	56.7%
15	14	2542	154	731	1509	2394	35588	6.7%	55.8%
27	14	3434	92	856	4115	5063	89284	5.6%	46.0%
4	28	1004	108	326	53	487	3012	16%	73.3%
11	28	2349	253	775	425	1453	23490	6.2%	64.7%
18	28	3526	278	1040	1283	2601	59942	4.3%	62.2%
25	28	4530	289	1257	2430	3976	108720	3.7%	58.3%
32	28	5023	245	1381	3640	5266	155713	3.4%	52.8%

**Client to Server:** transitioning from writer to reader necessitates flushing cached modifications to server

**Server to Client:** P = Promotion; D = Demotion; lock update sent to client (adjust lock position/status)

**Writer to Readers:** incremental changes made by writer selectively multicast to readers within subsection

**OT Messages:** # of write events \* (# agents – 1) (since we multicast to all agents other than the originating writer)

**DL Write Success Rate:** # successful modifications to document accomplished / total modifications attempted (only for the DL simulation since OT write success rate is by definition 100%)

## V. RELATED WORK

Others have examined utilizing varied-granularity locks within a shared document to achieve increased concurrent access and avoid the bottleneck traditionally associated with pessimistic, lock-based concurrency control [2]. In software code, fine-grain locking at a class/function/method level would be advantageous [1]. [1] and [3] 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. Some systems such as Coven [1], DistEdit [6], and COOP/Orm [9] allow the lock to be made at a sub-file level, but these systems' unit of lock remains fixed in size or lock the smallest region possible; also, DistEdit only allows for turn-taking such that only one user may write to the shared document at a time. The POEM and MACE systems utilize hierarchical locks at a sub-file level, but the locks must be defined a priori and are fixed in size [7]. In contrast, our algorithms automatically obtain the largest lock permissible and automatically adjust the size/portion of the document locked.

Duplex [11] improves concurrent access similar to our approach by decomposing the shared documents into hierarchical sections and providing distributed/local cached copies of these sections; Duplex also allows for pessimistic and optimistic concurrency control as specified by the user. But Duplex still relies upon

“direct communication” among users to resolve conflict – either by coordinating who will work on each section or by merging disparate sections manually – whereas our system avoids this costly, user-dependent process through the combination of pessimistic and OT-based concurrency control.

[5] incorporates the idea of varied-granularity into OT-based concurrency control algorithms by allowing the user to specify at what semantic level within the document OT merging should occur; this approach seeks to incorporate flexibility into OT merge by allowing merge resolution to occur at the paragraph, sentence, or word level. This is similar [2] that applies OT algorithms to SGML/HTML/XML. We observe that communication costs may be improved by employing OT at a greater-than-character level due to the fact that this approach would use more of the network packet size than a single character (thus avoiding wasteful single-character payload packets).

## VI. CONCLUSION AND FUTURE WORK

We have demonstrated that a shared document in a synchronous, collaborative editing system may be modeled using an n-ary tree and presented an improved hierarchical locking scheme wherein locks are automatically managed based upon user demand for sections within the document. Lock promotion and demotion occurs dynamically and are guaranteed to be

deadlock free as we employ a top-to-bottom handshake-lock technique when traversing the tree. By maximizing the size of the lock, we are able to efficiently cache changes locally and avoid the expensive communication costs incurred by OT-based approaches. Our simulations results demonstrate a significant reduction of communication among users while ensuring reasonable concurrent access; these results support our assertion that our approach maximizes concurrent access while minimizing communication cost

Now that the lock management system has been implemented and proven to successfully enable concurrent access with minimal communication, we plan to incorporate this system into a larger architecture for enabling synchronous collaborative editing via heterogeneous client editors and heterogeneous server repositories [13]. Additionally, we plan to further study and enhance the simulation of user edit behavior to better model the types of read/write patterns that users in a CES employ; currently, the action probabilities are determined arbitrarily, and modifying these agent behaviors to more accurately reflect real-world users may yield interesting results. This may provide a more authentic measure of the communication cost savings that may be obtained by using our approach to concurrency control.

We recognize that moving from section to section may incur latency as the system communicates the new location and retrieves document state from the server, but we note that there is no latency in updating document state (communicating OT-based changes to other users) since the dynamic locking is employed; thus the system should be responsive as edits are made. When many users are clustered within a small region of the document and/or when the document structure is altered by combining or splitting sections, then the update to the document tree must be propagated among all users. It would be advantageous to distribute the document tree among the users in a peer-to-peer fashion, and such a distributed P2P approach could also be useful in avoiding starvation and bottleneck issues associated with a centralized approach. We are currently examining this as the next step in developing our dynamic-locking approach to CES.

#### REFERENCES

- [1] M. C. Chu-Carroll and S. Sprenkle, "Coven: brewing better collaboration through software configuration management", *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, November 06-10, 2000, San Diego, California, United States, p.88-97.
- [2] A. H. Davis, C. Sun, and J. Lu. "Generalizing Operational Transformation to the Standard General Markup Language", *Proceedings of CSCW 2002*, New Orleans, Louisiana, USA, November 16-20. pp. 58-67.
- [3] 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*, ACM Press, Nov. 1994, pp. 207-217.
- [4] N. Gu, J. Yang, and Q. Zhang, "Consistency Maintenance Based on the Mark & Retrace Technique in Groupware Systems", *GROUP'05*, ACM Press, Sanibel Island, FL, November 6-9 2005, pp. 264-273
- [5] C-L. Ignat and M. C. Norrie, "Flexible Merging of Hierarchical Documents", *7<sup>th</sup> International Workshop on Collaborative Editing Systems*. Sanibel Island, FL, 2005.
- [6] M. J. Knister and A. Prakash, "DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors", *Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, Los Angeles, CA, October, pp. 343 – 355.
- [7] R. Li and D. Li, "A Landmark-Based Transformation Approach to Concurrency Control in Group Editors", *GROUP'05*, ACM Press, Sanibel Island, FL, November 6-9 2005, pp. 284-293.
- [8] Y-J. Lin and S.P. Reiss, "Configuration management with logical structures", *Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society, Berlin, Germany, 1996, pp. 298–307.
- [9] B. Magnusson, "Fine-Grained Version Control in COOP/Orm", *European Conference on Computer Supported Cooperative Work 1995, Workshop on Version Control in CSCW Applications*, Stockholm, Sept. 1995.
- [10] B. Magnusson, U. Asklund, S. Minör, "Fine-Grain Revision Control for Collaborative Development", *Proceedings of ACM SIGSOFT'93*, Los Angeles, CA, December 1993.
- [11] F. Pacull, A. Sandoz, and A. Schiper. "Duplex: A distributed collaborative editing environment in large scale", *Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW '94)*, 1994, pp. 165-173.
- [12] J. A. Preston and S. K. Prasad, "A Deadlock-Free Multi-Granular, Hierarchical Locking Scheme for Real-time Collaborative Editing", *7<sup>th</sup> International Workshop on Collaborative Editing Systems*. Sanibel Island, FL, 2005.
- [13] J. A. Preston and S. K. Prasad, "A Web-Service-based Open-Systems Architecture for Achieving Heterogeneity in Synchronous Collaborative

Editing Systems”, Fourth International Conference on Cooperative Internet Computing, Hong Kong, China, 2006.

- [14] V. N. Rao and V. Kumar, “Concurrent Access of Priority Queues”, *IEEE Transactions on Computers*, vol 37, no 12, 1988, pp. 1657-1665.

- [15] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, “Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems”, *ACM Transactions on Computer-Human Interaction*, 5(1):63-108, Mar. 1998.

```

OBTAINLOCK(w, ui)
  if w.owner ≠ ui
    RECURSEOBTAINLOCK (ROOT, w, ui)

RECURSEOBTAINLOCK(n, w, ui)
  if n.color = white
    then SETLOCK(n, ui, w)
      LINKSIBLINGS(n.parent, n, n.parent.firstBlackChild)
  else if n ISATOMIC
    then RECURSEREMOVELOCK (ROOT, w, ui)
      return failure
  else if n.color = grey
    then n.greyCount = n.greyCount + 1
      RECURSEOBTAINLOCK(NEXTINPATH(n, w), w, ui)
  else b = NEXTINPATH(n, w)
    a = NEXTINPATH(n, n.originalRequest)
    SETLOCK(a, n.owner, n.originalRequest)
    n.color = grey
    n.greyCount = 2
    REMOVEFROMSIBLINGLIST(n)
    if a ≠ b
      then SETLOCK(b, ui, w)
        LINKSIBLINGS(n, a, b)
      else RECURSEOBTAINLOCK(a, w, ui)

```

```

SETLOCK(w, ui, r)
  w.color = black
  w.owner = ui
  w.originalRequest = r

LINKSIBLINGS(n, a, b)
  n.firstBlackChild = a
  a.previousSibling = NIL
  a.nextSibling = b
  b.previousSibling = a

UNSETLOCK(w)
  w.color = white
  w.owner = NIL
  w.originalRequest = NIL

```

```

REMOVEFROMSIBLINGLIST(n)
  n.previousSibling.nextSibling = n.nextSibling
  n.nextSibling.previousSibling = n.previousSibling
  if n.previousSibling ≠ NIL
    then n.parent.firstBlackChild = n.nextSibling
  n.previousSibling = NIL
  n.nextSibling = NIL

```

```

REMOVEDLOCK(w, ui)
  if w.owner = ui
    then RECURSEREMOVEDLOCK(ROOT, w, ui)

RECURSEREMOVEDLOCK(n, w, ui)
  if n.color = black and n.owner = ui
    then UNSETLOCK(n)
      REMOVEFROMSIBLINGLIST(n)
  else if n.color = grey
    then n.greyCount = n.greyCount - 1
      if n.greyCount = 1
        then a = FINDELIGIBLEPROMOTION(n, w)
          SETLOCK(n, a.owner, a.originalRequest)
          LINKSIBLINGS(n.parent, n, n.parent.firstBlackChild)
      else if n.greyCount = 0 // removal occurs before delayed promotion
        then UNSETLOCK(n)
      else RECURSEREMOVEDLOCK(NEXTINPATH(n,w), w, ui)

```

```

FINDELIGIBLEPROMOTION(n, w)
  traverse from n to w until black node (a) is found
  if a.nextSibling.color = black
    then return a.nextSibling
  else if a.previousSibling.color = black
    then return a.previousSibling
  else return a

```

These algorithms are intended to show intent; the actual implementations feature an iterative solution that employs a top-to-bottom, handshake-lock as the paths from the root to the desired nodes are traversed.

**Figure 10. Dynamic Lock Management Algorithms for Generalized n-ary Document Trees**