

Improving Concurrent Access in Collaborative Editing Systems

PhD Proposal

Jon A Preston



Department of Computer Science
Georgia State University
Atlanta, Georgia

Committee Members

Dr. Sushil K. Prasad (Advisor)

Dr. Rajshekhar Sunderraman

Dr. Xiaolin Hu

Dr. Melody Moore Jackson (External)

Table of Contents

TABLE OF CONTENTS	2
ABSTRACT.....	5
1. INTRODUCTION	6
1.1 MOTIVATION.....	6
<i>Software Engineering</i>	6
<i>Collaborative Document Development</i>	7
<i>Computer Aided Design (CAD)</i>	8
1.2 PROBLEM STATEMENT	9
1.3 CONTRIBUTIONS	10
1.4 PROPOSAL ORGANIZATION	11
2 RELATED WORK.....	12
2.1 CONCURRENT ACCESS	12
2.1.1 <i>Responsiveness</i>	13
2.1.2 <i>Concurrency Control – Pessimistic</i>	14
2.1.3 <i>Concurrency Control - Optimistic</i>	17
2.1.3 <i>Dynamic/Multi-level Concurrency Control</i>	19
2.1.5 <i>CCI and Techniques to Ensure Convergence</i>	19
2.1.6 <i>Distributed Configuration Management Systems</i>	22
2.2 ARCHITECTURES.....	25
2.2.1 <i>Computer Supported Collaborative Workspaces</i>	26
2.2.2 <i>Presence/Awareness</i>	27
2.2.3 <i>Single-User to Multi-User Application Architectures</i>	28
2.2.4 <i>Transparency, Awareness, and CES Components</i>	32
2.3 EXISTING APPLICATIONS	39
3. RESEARCH GOALS	40
4. CURRENT RESEARCH SUMMARY	46
4.1 THEORETICAL FRAMEWORK: ALGORITHMS AND DATA STRUCTURES	46
4.1.1 <i>Document Partitioning</i>	46
4.1.2 <i>Deadlock-free, Multi-granular Locking</i>	50
4.2 HETEROGENEOUS ARCHITECTURE	63
4.2.1 <i>Proxy-based Configuration Management</i>	63
4.2.2 <i>Multi-granular Locking Simulation</i>	64
4.3 PROTOTYPE SYSTEM.....	71
4.3.1 <i>Peer-to-Peer Communication Analysis</i>	71
4.3.2 <i>Hooking into Existing Editors</i>	81
5. INTENDED RESEARCH METHODOLOGY	82
5.1 ALGORITHMS AND DATA STRUCTURES	82

5.2 HETEROGENEOUS OPEN-SYSTEM ARCHITECTURE	83
5.3 PROTOTYPE SYSTEM.....	84
6. CONCLUSION	86
7. BIBLIOGRAPHY	87

LIST OF FIGURES

FIGURE 1 - USING AN ORDERED BROADCAST PROTOCOL [1]	14
FIGURE 2 - PESSIMISTIC CONCURRENCY CONTROL	16
FIGURE 3 - OPTIMISITIC CONCURRENCY CONTROL	18
FIGURE 4 - MARK AND RETRACE [43]	21
FIGURE 5 – DISTRIBUTED EVENT SYSTEM MODEL SHOWING ADAPTERS TO EXTERNAL CLIENTS AND SERVERS [6]	26
FIGURE 6 - AWARENESS COMPONENTS INTEGRATED INTO ECLIPSE [64]	28
FIGURE 7 – THE DISTEDIT APPROACH OF ADDING COLLABORATION TO EXISTING APPLICATIONS [27]	29
FIGURE 8 - THE CoWORD APPROACH [18]	30
FIGURE 9 - GENERALIZED COLLABORATIVE ARCHITECTURE FROM [3]	31
FIGURE 10 - DISTRIBUTIONS OF MODELS, VIEWS, AND DISPLAYS	37
FIGURE 11 - A WEB SERVICES-BASED COLLABORATIVE EDITING ARCHITECTURE [96].	38
FIGURE 12 - MAPPING FROM DOCUMENT TO TREE TO BINARY TREE	47
FIGURE 13 - PATH TO UNIQUELY-IDENTIFIED NODE	49
FIGURE 14 - GREY NODE CONFIGURATIONS	50
FIGURE 15 - THE INSERTUSER OPERATION WITHOUT DEMOTION	53
FIGURE 16 - THE INSERTUSER OPERATION WITH DEMOTION	55
FIGURE 17 - INSERTION OF A USER	56
FIGURE 18 - CASE 1 OF THE REMOVEUSER OPERATION	57
FIGURE 19 - CASE 2 OF THE REMOVEUSER OPERATION	58
FIGURE 20 - REMOVAL OF A USER	59
FIGURE 21 -REMOVEUSER (WITH MULTI-LEVEL PROMOTION)	61
FIGURE 22 - OPEN-SYSTEM ARCHITECTURE FOR DISTRIBUTED REPOSITORIES	65
FIGURE 23 - PEER-TO-PEER COMMUNICATION: CHAT	79
FIGURE 24 - PEER-TO-PEER COMMUNICATION COSTS	79
FIGURE 25 – PROTOTYPE IMPLEMENTATION: TECHNOLOGIES TO BE INTEGRATED	85

LIST OF TABLES

TABLE 1 - COMPARING TRANSPARENT AND AWARE COLLABORATIVE SYSTEMS	35
TABLE 2 - ARCHITECTURE SIMULATION RESULTS	68
TABLE 3 - COMMUNICATION AMONG PEERS FOR VARIOUS ACTIVITIES	80

ABSTRACT

Networked computer systems offer much to support collaborative editing of shared documents among users. Software Engineering is one of many fields that benefits from computer-assisted collaboration as a myriad of developers, project managers, testers, and designers work together to develop large, complex systems that consist of a multitude of process and product artifacts. Multi-discipline and geographically-distributed production and research teams collaborate and co-author documents for businesses and universities worldwide. Such collaborations occur asynchronously via access to shared document repositories often assisted by configuration management systems and occur synchronously via shared, real-time collaborative editing systems often assisted by awareness-enhancing technology. Increasing concurrent access to shared documents by allowing multiple users to contribute to and/or track changes to these shared documents is the goal of collaborative editing systems; yet concurrent access is often limited in existing collaborative editing systems, and such systems are often specialized in their functionality and require users to adopt new, unfamiliar software to enable collaboration. Therefore, the purpose of this proposal is to (i) explore algorithms and functionality necessary to maximize concurrent access to shared documents, (ii) develop an architecture that allows for a heterogeneous set of client editing software to connect with a heterogeneous set of server document repositories, and (iii) develop a prototype system of our architecture that is responsive to users' actions and minimizes communication costs.

1. Introduction

Computer Supported Collaborative Work (CSCW) is a discipline of computer science that relies upon foundational research in the fields of database, operating systems, networking, human-computer interaction, and software engineering. Collaborative Editing Systems (CES), also known as group editors, allow multiple users to synchronously collaborate on shared documents. In an age of interconnected computers and high-speed networks, facilitating collaboration among users involves managing concurrency to shared documents, providing awareness to users of the system, and integrating existing systems, such as client editing tools and server document repositories, into interconnected meta-systems [104] that allow users to retain interfaces that are both familiar and allow for maximum productivity [65].

1.1 Motivation

Computer Supported Collaborative Work and more specifically Collaborative Editing Systems have a rich history of research and significant contributions in various fields since the 1980s [11][60][93]. These systems remain collaboration-centric as the computing system merely supports the activity at hand [55]. The following are select example domains in which Collaborative Editing Systems applications that correspond to research questions to be addressed in this work.

Software Engineering

At the heart of software systems development is the coordination of various developers, project managers, documents, and source code [90]. While much work within software engineering involves decomposing large systems into subsystems that can be developed in parallel [1][92][101], much work related to coordination remains a vital part of a

software system development project [90][58]. Managing ever-changing project artifacts such as requirements, plans, test documents, and system models involves coordinating access to either a centralized document repository or a distributed, replicated document repository; with this comes the concomitant consistency management practices [90].

Developers of a software system must be informed of changes not only to the source code but also the foundational project definition documents (requirements, designs, plans, etc.) [38][37]. Awareness of what other users are doing within the system as well as a view of what documents other users are accessing helps avoid conflicting changes and coordinate the development effort [38]. Coordination among developers can be formal or informal and is often driven/defined by the software engineering processes employed with the project [28][91]. Central to the ability to collaborate on documents is the ability to work within a group and coordinate group effort. In a traditional software engineering setting, these activities entail project task scheduling, status reporting (and meetings), and inter-group communication [54][106].

Recently, there has been an increase in commercial interest in the field of integrating collaboration mechanisms into integrated development environments [53][47][48], validating that this area of research has interest in the commercial sector.

Collaborative Document Development

Moving from the specific field of Software Engineering, we can generalize to document sharing and collaborative editing as a joint task among multiple authors either co-located or distributed geographically [54]. Additionally, users may wish to edit the shared document synchronously (at the same time) or asynchronously (at different times) [73]. Collaborative document editing involves a high level of interactivity among users, and

ensuring rapid response time to changes in the document and maintaining a familiar look-and-feel (allowing use of users' favorite, existing editors) are paramount design goals for any collaborative document editing system [1][74]. There has been an increase in recent commercial development of collaborative document management systems in recent years, validating that this area of collaborative editing system research is becoming commercially viable [48][49]. While these systems demonstrate some problems in the field of collaborative document development have been solved, other research problems remain open.

Computer Aided Design (CAD)

Another field that we note would benefit from computer assisted collaboration is design. CAD systems have long supported designers develop schematics, renderings, and other design-related documents. Recent studies in CSCW also support the idea that the design process can benefit from collaborative editing [56]. What is most interesting about this particular field of CES is that modern CAD systems store the documents being edited as objects with layering, so it is believed that the concurrency control employed in CAD systems must manage collections of objects within the document that are not necessarily spatially structured but are rather structured via grouping. For example, all of the electrical wiring (the electrical objects collection/layer) of a building schematic could be locked by one user for editing while all of the flooring (the flooring objects collection/layer) could be locked by another user for editing. We specifically address this domain of CAD because it offers an opportunity to manage concurrent access to collections of objects within a document that are not necessarily spatially related [102],

and our algorithms and models generated for this proposed work easily accommodate this non-spatial organizational structure.

1.2 Problem Statement

Considering that computers have greatly enhanced the ability for people to collaborate (via email, shared information repositories, knowledge management tools, etc) [60][55], it is reasonable to expect that many commercial document editing systems would support synchronous and asynchronous collaboration among multiple authors. Unfortunately, this is not the case; document editors remain isolated as single-user applications, and what progress that has been made in this area focuses on sharing documents and merging different changes made by multiple users (change tracking within Word, for example), but does little to solve the problem of synchronous collaboration. Further, server-side repositories allow only the most primitive access to artifacts at only a file-level, completely ignoring any semantic structure contained within the artifacts being managed [52].

Other CSCW-based approaches to synchronous document editing [10][18][21][27][71][95] either develop foundational/theoretical frameworks that define how synchronous editing of multiple users may properly conform to the CCI-based model (Convergence, Causality-preservation, and Intention-preservation) or integrate into a specific application without regard to the issue of supporting heterogeneous collections of clients and servers [68]. Additionally, distributed repositories are only now beginning to support fine-granular locking mechanisms [70] but most still lack semantic/structural knowledge of the documents they manage, limiting their efficiency in concurrency control. Finally, new CSCW-based architectures supported by Web-services and a generalized view of the

activities and entities associated with collaborative editing systems have been proposed, but none address the issue of heterogeneity among clients and server repositories.

The focus of this thesis proposal is to conduct significant research on the above-listed issues with the specific goal of providing a prototype system that demonstrates reasonable solutions to these open problems and brings together heterogeneous client editing systems and server document repositories to allow users to increase concurrency in their collaboration.

1.3 Contributions

We envision users opening their existing, favored document editor (or software development IDE) and selecting a document that they would like to edit from a distributed repository that resides on other, server computers; these server repositories will be supported by existing, proven configuration management systems; heterogeneity will be supported on both the client and server side of the collaborative editing system. While users synchronously edit the same document, they will each be notified of changes by other users as needed, and their local changes will be distributed to other users as appropriate. Responsiveness in the collaborative editing system will be on par with responsiveness in the single-user editing system, and communication costs between clients and servers will not be prohibitive to the usability of the editing system. Concurrency control will be transparent to the users.

We propose to:

- (i) Explore algorithms and functionality necessary to maximize concurrent access to shared documents.

- (ii) Develop an architecture that allows for a heterogeneous set of client editing software to connect with a heterogeneous set of server document repositories that supports the algorithms developed
And
- (iii) Develop a prototype testbed system of our architecture that is responsive to users' actions and minimizes communication costs, demonstrating the effectiveness of our algorithms and architecture

1.4 Proposal Organization

The rest of proposal is organized as follows: In chapter 2, we discuss related work. We present our research goals in chapter 3. In chapter 4, we summarize the research that has already been completed. We present our plan of intended research and technical milestones for completing our research in Chapter 5.

2 Related Work

This section discusses the related work that others have done in the field of collaborative editing system. We begin with a discussion of concurrency management, the CCI model, techniques to ensure convergence, and distributed configuration management algorithms in section 2.1. Section 2.2 focuses on various architectures employed in CES, and section 2.3 concludes the related work section of this proposal by discussing existing systems that employ CES approaches.

2.1 Concurrent Access

Since a shared set of objects reside at the heart of any collaborative system, some mechanism must be in place to coordinate the activities of the multiple users within the system. Traditionally in collaborative editing, one of two approaches is taken with regard to coordination: pessimistic concurrency control or optimistic concurrency control.

Configuration management systems (and CSCW systems) typically take one of two approaches with regard to locking: optimistic or pessimistic locking. In the optimistic approach, users are free to edit in a more parallel fashion, but conflict occurs at the merge point when two sets of edits must be merged together and changes brought together (to avoid losing work and ensuring that changes in one file have not adversely affected changes in the other file) [85]. In the pessimistic approach, users must obtain a lock on a document before being able to edit it; this can reduce the parallel nature of development since at most one user can edit the document at any time.

Real-time collaborative editing systems avoid the merge problem by immediately broadcasting edits to all other users within the system; in this way, all users' copies of the shared document are kept reasonably up-to-date. The concomitant problem with this

approach is that communication costs are significant. Additionally, since local changes could be made at one user's machine before the changes on another user's machine is received and processed, to ensure that the operation is "replayed" locally correctly, some form of transformation may be necessary.

This section discusses mechanisms to manage concurrent access to shared documents.

2.1.1 Responsiveness

To enable concurrent access in a distributed collaborative system, we must either centralize the storage of the document being edited onto a server and have "thin" clients that merely relay user input/changes, or copy the document being edited onto the clients and coordinate the changes made to the document by all the users (essentially ensuring cache consistency). A centralized approach has proven to be too costly with regard to communication costs and lacks adequate responsiveness typical of an interactive application [42]. Consequently, distributed approaches are typically employed in CES.

Assuming a multi-user system employs replication to allow multiple users access to a shared document, we must ensure that the replicated document state is consistent among the users. If all users are allowed to make local changes to their copies of the document, these changes could be broadcast to the other users and the changes "replayed" on the local copies to ensure consistency. Unfortunately, the ordering of the replayed changes is not preserved, and consequently the replicated copies of the document become unsynchronized. To ensure consistency among the replicas of the document, some form of concurrency must be employed.

Ordered broadcast protocols may be used to ensure proper ordering of changes to the shared document. But this approach requires that all changes be sent to a central

controlling server and local changes cannot be affected until the server responds to the client making the change; consequently, the response time of such systems is typically not appropriate for interactive systems. Additionally, such broadcast protocol approaches require that the changes are operationally-transformed to the client's current document state to preserve user intention [1]. As Figure 1 demonstrates, the state of the document is changed as a result of performing operation A, thus execution of operation B may have unintended results.

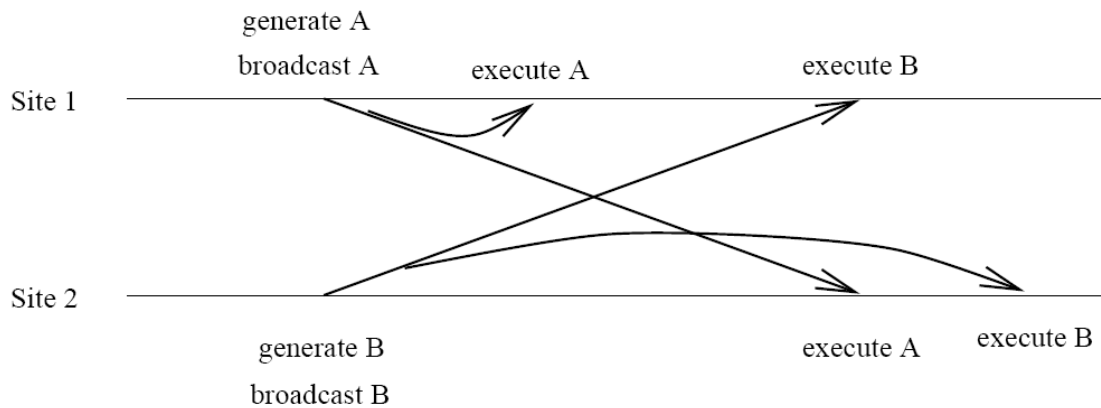


Figure 1 - Using An Ordered Broadcast Protocol [1]

Because of the interactive nature of collaborative editing systems, traditional transaction-based and pessimistic locking schemes typically employed in database systems are often not appropriate. Alternatively, most collaborative editing systems employ some form of optimistic concurrency control in an effort to improve interactive responsiveness.

2.1.2 Concurrency Control – Pessimistic

Pessimistic-lock based SCM systems such as RCS, VSS, and SCCS do not allow for multiple users to concurrently modify the artifact; thus by locking at the file level, these SCM systems can reduce concurrency in developing documents [69].

These systems pessimistically assume that users within the system will desire to edit the same object at the same time and that such edits will be destructive or cause problems. Since this is a shared resource/object, consistency and causality are important. Notice the similarity to causal memory, shared memory, and cache coherency in distributed systems research.

Pessimistic coordination policies are typically implemented using a “check in” and “check out” API. Users may gain access to an unused document by issuing a “check out” request; the document is then locked for that user, and no other user may access the document. When a user has completed any edits to a checked out document, he may issue a “check in” request, returning the document to the repository with any changes made to the local copy.

Since only one user has access to the shared document at any given time, the problem of multiple versions of the same document within the system is avoided. Thus, no two users can have writable copies checked out at the same time. Updates to the repository occur upon a “check in” command, and the old copy of the document is overwritten with the new copy of the document. Often, differentials are saved so that “undo” or “revert to old version” commands are possible. Figure 2 illustrates this.

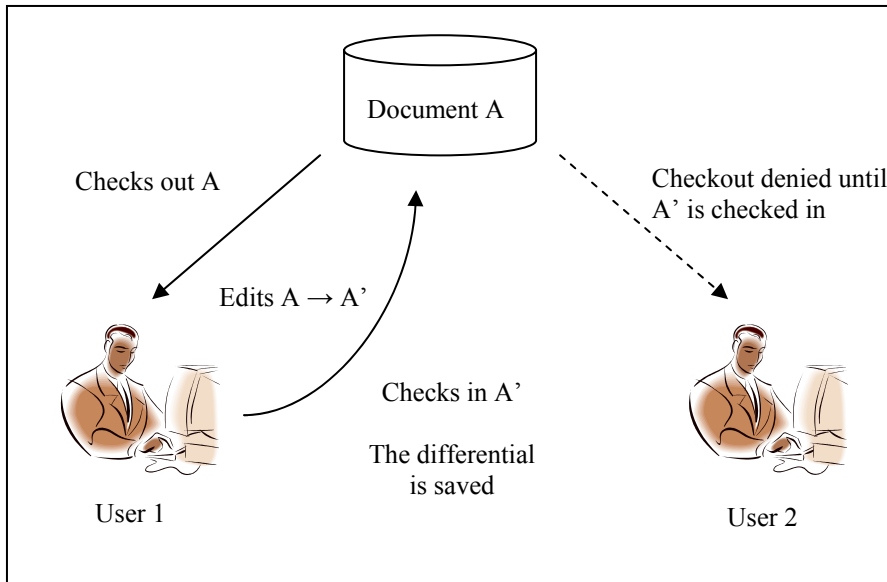


Figure 2 - Pessimistic Concurrency Control

One major limitation of the pessimistic coordination policy is the lack of concurrency in the distributed environment; since only one user can access each shared document at a time, then concurrency of collaboration may be inhibited. A few solutions to this problem exist:

First, one can reduce the size of the code placed into each atomic element within the repository. Since each element (document) within the repository contains less code, the probability of two users requesting the same document may be reduced. This is akin to breaking up a large file into smaller files, each of which may be checked out concurrently without being inhibited by the pessimistic locking policy. Of course, it may not always be possible to create small documents within the repository, and a highly-desired document may inhibit concurrency regardless of its size.

Second, configuration management repositories may allow users to check out “read only” copies of an already-checked-out document. I.e. if one user already owns a document,

other users may view (but not edit) the contents of this document. Such a local copy could be used within local users' workspaces for "what if" editing without corrupting the original, master copy. If such local changes are deemed relevant to the master copy, the user can later check out the master and incorporate these changes.

2.1.3 Concurrency Control - Optimistic

SCM systems such as CVS employ optimistic locking. This coordination policy assumes optimistically that users will not need to access the same resource at the same time frequently [52][76], thus this policy promotes increased concurrency among collaboration at the cost of potential problems in inconsistency in the shared documents and loss of causal access. Such a policy is indicative of and seems to work well in an "agile development" environment where communication and productiveness trump tools, processes, and planning [77].

Optimistic coordination systems are typically implemented using awareness within the system such that users are made aware of each others' activities. Awareness is defined as "an informal understanding of the activity of others that provides a context for monitoring and assessing group and individual activities" [78]. In such a system, synchronous updates occur immediately when an edit occurs (akin to a write through cache policy in distributed shared memory systems). Consequently, all users have a current copy of any shared document and no check-in and check-out is needed because any document a user is editing is by definition checked out (and perhaps checked out simultaneously by many users) [77]. Figure 3 illustrates the optimistic coordination policy.

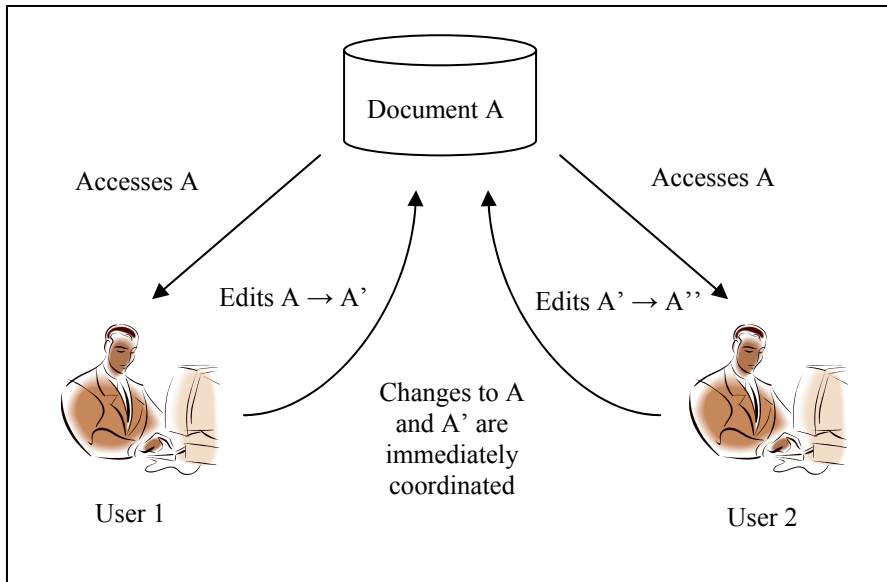


Figure 3 - Optimistic Concurrency Control

Such awareness-based optimistic systems rely upon users to coordinate and avoid collisions in edits to the shared document. According to current CSCW research, this seems to work reasonably well in smaller work groups, but does not scale well to larger collaborations among many users [77]. Two proposed reasons for this include the limited amount of cognitive information users may process simultaneously and the inherent dichotomy of informal coordination and formal, process-driven coordination.

Consequently, optimistic coordination policies work well in smaller collaborative environments with fewer users when self-coordination is accomplished by the users of the system. Alternatively, algorithms to resolve disparate versions of the documents in real-time may be employed if the coordination of changes is to be made automatic; approaches such as operation transformation (OT) [20] as discussed later in this paper can be used to ensure convergence of all copies of the document.

2.1.3 Dynamic/Multi-level Concurrency Control

Many software configuration management (SCM) systems managed locks at the source file level within the repository. Examples include RCS, SCCS, VSS, CVS, and Subversion [45] and view the file as the unit on which to manage locks. But it is often advantageous to allow for finer granular locking to enhance concurrent access, increase reuse through aggregation of artifacts, and easy convergence/merging of disparate versions [50][87]. Given that many edits by users in a software engineering project are localized and only change a small section of the document[13][15], fine-grain locking at a class/function/method level would be advantageous [71]. Some systems such as Coven [71] and COOP/Orm [51] allow the lock to be made at a sub-file level, but these systems' unit of lock remains fixed in size; the lock does not adjust in size dynamically with regard to what other users are doing in the collaboration. Another system (POEM) utilizes the hierarchical nature of software code to lock at a sub-file level, but the methods must be defined a priori, and again the locks remain fixed in size [70].

2.1.5 CCI and Techniques to Ensure Convergence

If mutual exclusion is not guaranteed as the mechanism for ensuring consistency control, then another alternative technique must be adopted to ensure that changes made by concurrent users are preserved.

Sun et al [21] proposed the most widely adopted standard for consistency maintenance in real-time cooperative editing systems when defining the CCI model. This model ensures convergence, causality-preservation, and intention-preservation.

Convergence: when the same set of operations have been executed at all local copies, then the local copies will all have the same content/state.

Causality-preservation: for operations O_1 and O_2 , if $O_1 \rightarrow O_2$ then O_1 precedes (is executed before) O_2 at all local copies.

Intention-preservation: executing an operation O does not change the effects of executing operations $O_1 \dots O_n$ where $O_1 \dots O_n$ are independent of O . Further, the effects of executing O at any local copy is the same as the intention of O (i.e. the intention is the same across all copies).

Wang et al [44] build upon the CCI model and inject the notion of semantic consistency. This work proposes three levels of consistency in their model: operational consistency, content (syntactic or intention) consistency, and semantic consistency. While this model acknowledges that the CCI model ensures consistency control, the new 3-level model addresses the fact that semantic knowledge within the document could allow for different ordering of operations (violating causality-preservation) and allowing for the omission of some operations (violating convergence in that not all operations must be executed) while still maintaining the syntactic and semantic intention of the users.

Alternatively, another approach to ensuring consistency when utilizing a distributed, replicated groupware system is through a “mark and retrace” approach [43]. In this approach, when a new operation from another editor arrives at the local copy of the document, the document’s address space (state) is analyzed relative the efficient/inefficient marked states as shown in Figure 4.

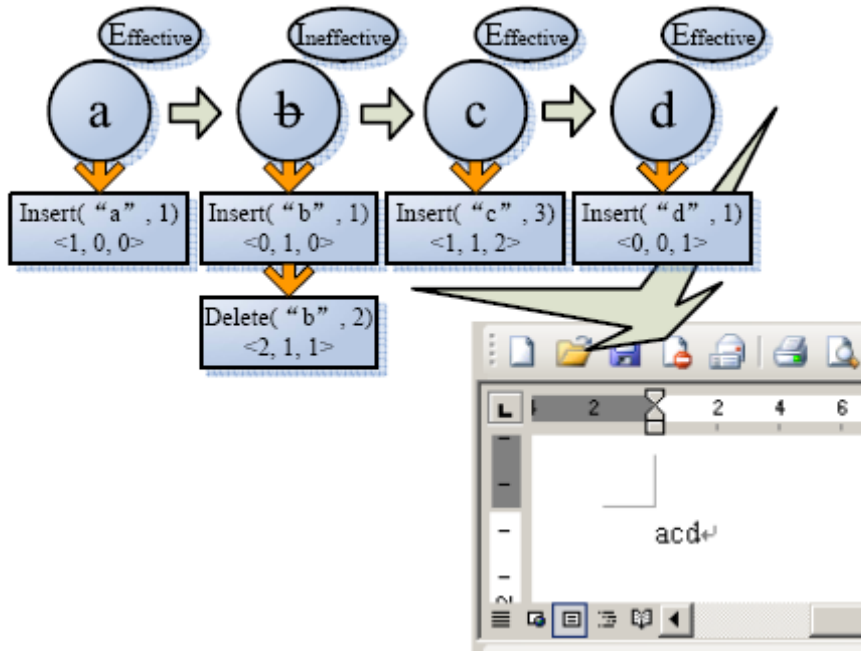


Figure 4 - Mark and Retrace [43]

[62] presents work that allows for the extension of operational transformation techniques to be applied not only to linear text but also to tree-based XML/SGML/HTML documents. The SGML notion of a “grove” of data is utilized and the CCI model is adhered to [21]. Others [86] have applied the techniques of OT to more complex data structures required in word processors. In this approach, termed multi-version, single-display (MVSD), multiple versions of the objects’ states are stored internally and only one version of the object state is displayed to the user; users may then select the correct version desired. The multi-version approach is also employed in [105] within the domain of graphic editing systems, and the challenge of this approach remains achieving semantic consistency rather than syntactic consistency [44].

2.1.6 Distributed Configuration Management Systems

A "Distributed Version Control System" (DVCS) is one in which version control and software configuration control is provided across a distributed network of machines. By distributing configuration management across a network of machines, one should see an improvement in reliability (by replicating the file across multiple machines) and speed (response time). Load balancing can be another benefit of distributed configuration management. Of course, if file replication is employed, then we must implement a policy whereby all copies of the file are always coherent [30].

In order for distributed configuration management to work efficiently, the fact that the files/modules are distributed across multiple computers on the network must be transparent to the developer/user. The user should not be responsible for knowing where to locate the file he/she is seeking. Rather, the system should be able to provide an overall hierarchical, searchable view of the modules present in the system; the user should be able to find their needed module(s) without any notion of where it physically resides on the network [31][32].

Another interesting aspect of distributed configuration management is the idea that the system provides each user with a public and private space for the files [33]. The public space contains all of the files in the collaborative, distributed system. The private space contains minor revisions or "what if" development files that the local user can "toy with" in an exploratory manner; this provides a safe "sandbox" area that each developer can use to explore possible ideas and changes. When a module is ready for publication to others, it is moved from the private space into the public space [30]. The Coven system

also employs this notion of public/private space and utilizes “soft locks” to coordinate concurrent access to the shared documents [71].

Guaranteeing mutual exclusion to the critical section is a classic problem in computing. In the cases of distributed software engineering in a collaborative environment, we need to guarantee that only one user can be editing any section of the collaborative shared space at any given time. In some cases, we might like to allow k users to have simultaneous access to a shared resource (where $k \leq n$, n = total number of users in the system). This section examines the various mutual exclusion algorithms that are relevant within the context of collaborative systems.

Distributed mutual exclusion algorithms fall into one of two primary categories: token-based and permission-based [99]. In a token-based system, a virtual object, the token, provides the permission to enter into the critical section. Only the process that holds the token is allowed into the critical section. Of interest is how the token is acquired and how it is passed across the network; in some models, the token is passed from process to process, and is only retained by a process if it has need for it (i.e. it wants to enter the critical section). Alternatively, the token can reside with a process until it is requested, and the owner of the token makes the decision as to who to give the token to. Of course, finding the token is potentially problematic depending upon the network topology [34].

The other approach to distributed mutual exclusion is the permission-based approach. In the permission-based approach, a process that wants to enter the critical section sends out a request to all other processes in the system asking to enter the critical section. The other processes then provide permission (or a denial) based upon a priority algorithm, and can only provide permission to one process at a time. Once a requesting process

receives enough positive votes, it may enter the critical section. Of interest here is how to decide the priority algorithm and how many votes are necessary for permission [34].

In the case where we would like to allow some subset of users access to the shared resource (or shared data) simultaneously, the work of [36] is of particular interest. Their algorithm achieves k-mutual exclusion with a low delay time to enter the critical section (important to avoid delays within the system) and a low number of messages to coordinate the entry to the critical section. In their model, they use a token-based system where there are k tokens in the system; further, the system is starvation and deadlock free [36].

The k-mutual exclusion algorithm differs from the traditional mutual exclusion algorithm in that in a network of n processes, we allows at most k processes into the critical section (where $1 \leq k \leq n$). The algorithm developed by [35] utilizes a token-based approach that contains k tokens. Tokens are either at a requesting process or a process that is not requesting access to the critical section. In order for this algorithm to work, all non-token requesting processes must be connected to at least one token requesting process. To ensure that tokens do not become "clustered," the algorithm states that if a token is not being used, then it is sent to a processor that is not the processor that granted the token [35].

Distributed token-based, permission-based, and k-mutual exclusion algorithms are all useful in various scenarios. The primary use and impact of distributed mutual exclusion in the context of this research is to manage access to shared documents in the collaborative editing system. This is a vital part of any distributed editing environment with simultaneous users accessing shared documents.

2.2 Architectures

[1] defines a group editor as “a system that allows several users to simultaneously edit a document without the need for physical proximity and allows them to synchronously observe each others’ changes.” Thus an architecture that supports a collaborative editing system must provide for distributed coordination of access to a set of shared artifacts. Providing asynchronous access is easily achieved; and we note that asynchronous access is just a special case of synchronous access wherein there is only one user in the system. Thus most problems and research focus on providing synchronous coordination among multiple users.

[29] defines three fundamental elements of collaborative systems. These fundamental elements are: user management, content management (and version control), and process management. These are defined as :

- “User management is defined as the **administration of user accounts and associated privileges**. This administration should be as simple as possible to avoid wasted time and confused roles.
- Content management is the **process of ensuring the integrity of the data** at the heart of the project. Content management systems often employ **versioning control** that transparently preserves the progression of the project as the associated documents mature and grow.
- A workflow is an abstraction of the process that a task takes through a team of people. During the execution of the workflow, it is often difficult and time-consuming to manage individual processes. Process management **handles the interaction between different levels of project contributors.**”

2.2.1 Computer Supported Collaborative Workspaces

The characteristic of the CES in focus of this work center around distributed cognition and the people, goals, artifacts, structures, and transformation on these structures [100]. [6] advocates the combination of WWW information systems (representation, storage, distribution and visualization of hypermedia information) with group interaction/CSCW systems. Their architecture utilizes existing, external client and server components and connects them to their new, internal components via “plug ins” or adapters, keeping the external components as “black boxes” to their architecture; sensor components within an adapter listen for server communication and actor components within an adapter relay the actions/events into the external components. Additionally, their Distributed Event System (DES) connects internal components. A session manager is responsible for storing session information about which users are active and notification via email, NNTP, IRC, etc. are handled within this system’s notification manager.

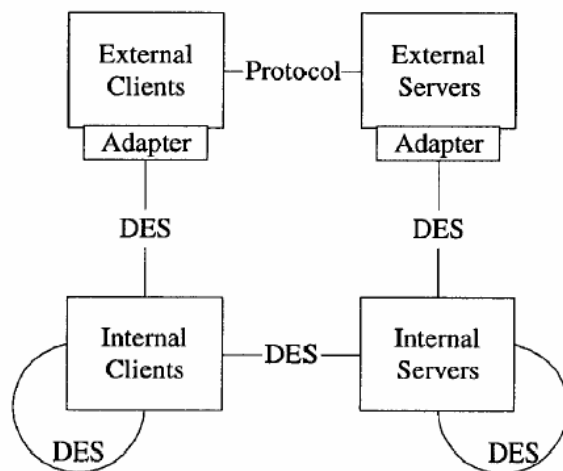


Figure 5 – Distributed Event System Model Showing Adapters to External Clients and Servers [6]

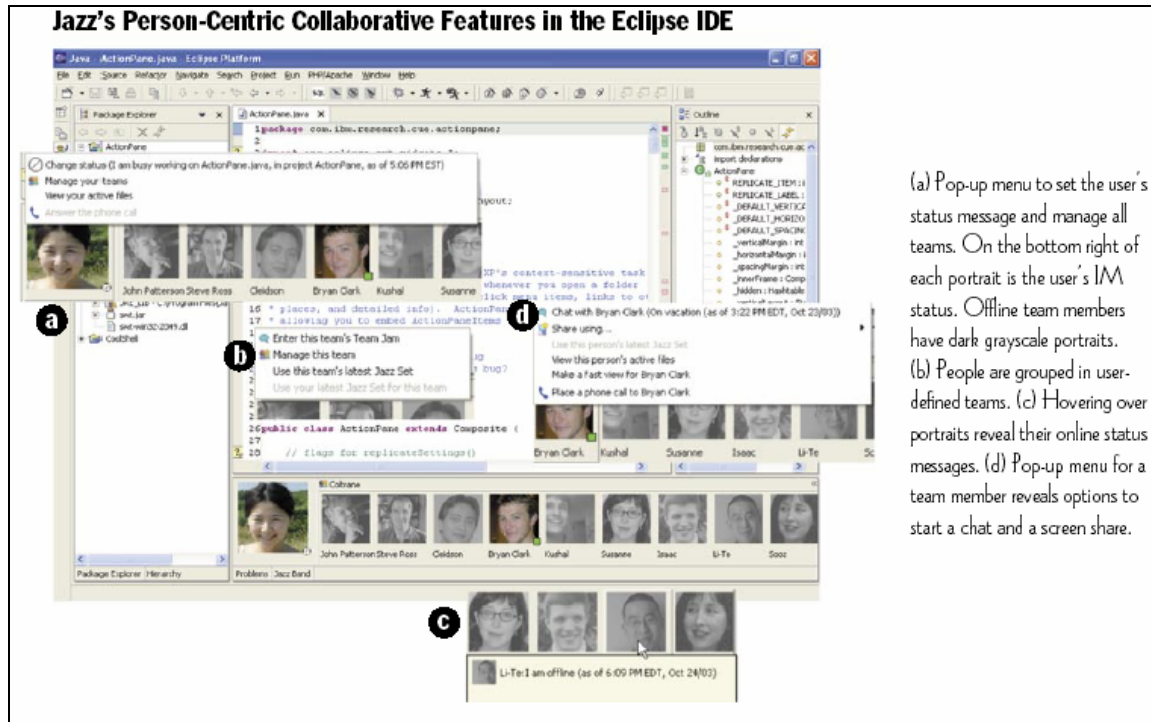
[59] discuss their work in developing an extension of a simple Web server that provides the ability to upload documents, version management (simple pessimistic/locking

concurrency control), and access and group control/administration. CSCW span traditional workstation-based computing and are moving into mobile, PDA-based uses [103], demonstrating that architectures for CSCW must be flexible in supporting various applications and platforms.

2.2.2 Presence/Awareness

[1] enumerates collaboration awareness as one of the critical features of group editors. Collaboration awareness is defined as “sufficient context information so that users are aware of other active participants in the group session... so as to encourage communication and avoid conflicting work.” Such awareness includes knowing what other users are active in the collaborative environment (participant context), synchronizing the relative views of a set of users so that other users can see the document being edited from one user’s point of view (view context), and knowing what activities other users are performing (activity context). When combined, these awareness techniques enable multiple users to better work together in the shared document.

Moving awareness and collaboration tools into the IDE has been a recent research goal of many systems. Given that the Java development IDE “Eclipse” is open-source, it is the focus of many of these studies [64]; see Figure 6 as an example of integrating awareness into the Eclipse IDE.



(a) Pop-up menu to set the user's status message and manage all teams. On the bottom right of each portrait is the user's IM status. Offline team members have dark grayscale portraits. (b) People are grouped in user-defined teams. (c) Hovering over portraits reveals their online status messages. (d) Pop-up menu for a team member reveals options to start a chat and a screen share.

Figure 6 - Awareness Components Integrated into Eclipse [64]

Pulling the collaboration awareness into the IDE is advantageous in that it does not require the user to learn a new system, and it integrates into the tool that developers are most familiar with and use most frequently [72].

2.2.3 Single-User to Multi-User Application Architectures

When attempting to achieve multi-user collaboration, other systems have taken existing single-user applications and modified them such that they can serve as a multi-user editing system. DistEdit [27] is one such system that integrates additional multi-user capabilities into an existing single-user editing system. Others include CoWord [18], CoPowerPoint [95], and CoStarOffice [94].

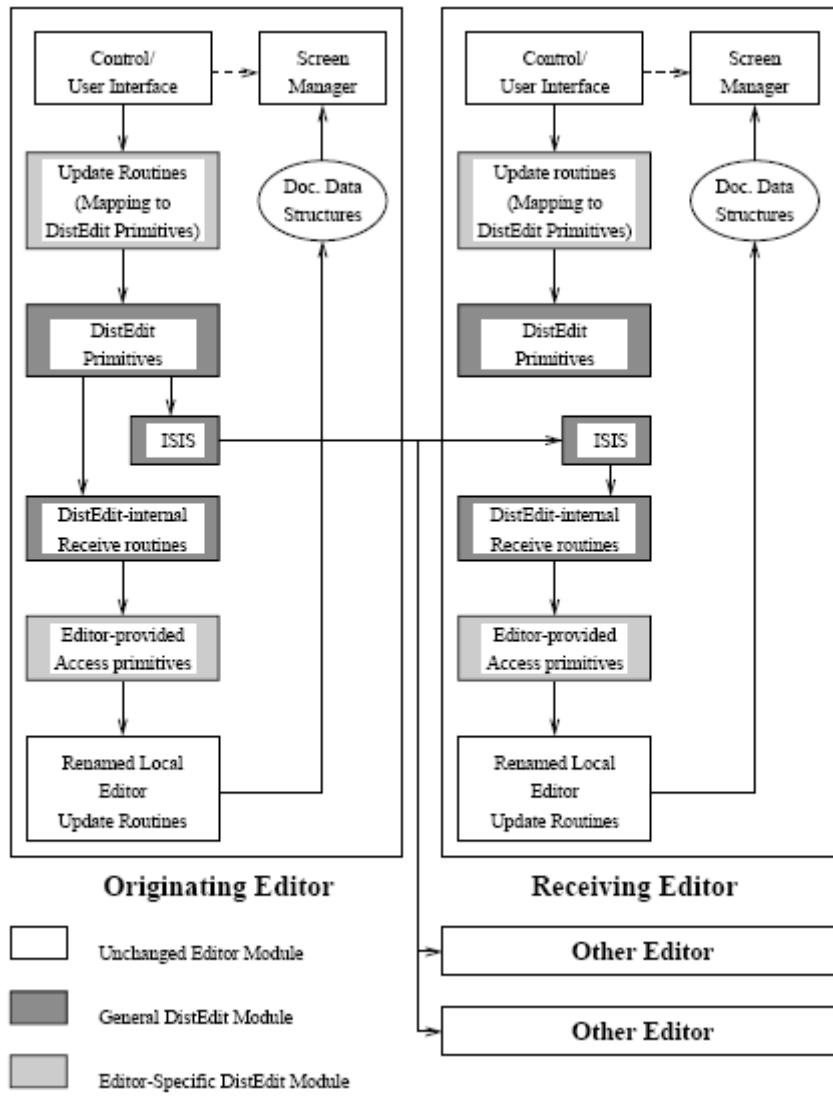


Figure 7 – The DistEdit Approach of Adding Collaboration to Existing Applications [27]

Notice in Figure 7 the original editing application components such as control/user interface, screen manager, and document data structures remain untouched; the update routines are modified to map to primitives that are broadcast to other editors and update the local copy of the document [27].

The CoWord and CoPPT projects are similarly structured in leveraging existing single-user applications with a collaborative adaptor and core collaborative engine hooked into

the existing application to provide for the collaboration functionality [18] as shown in Figure 8.

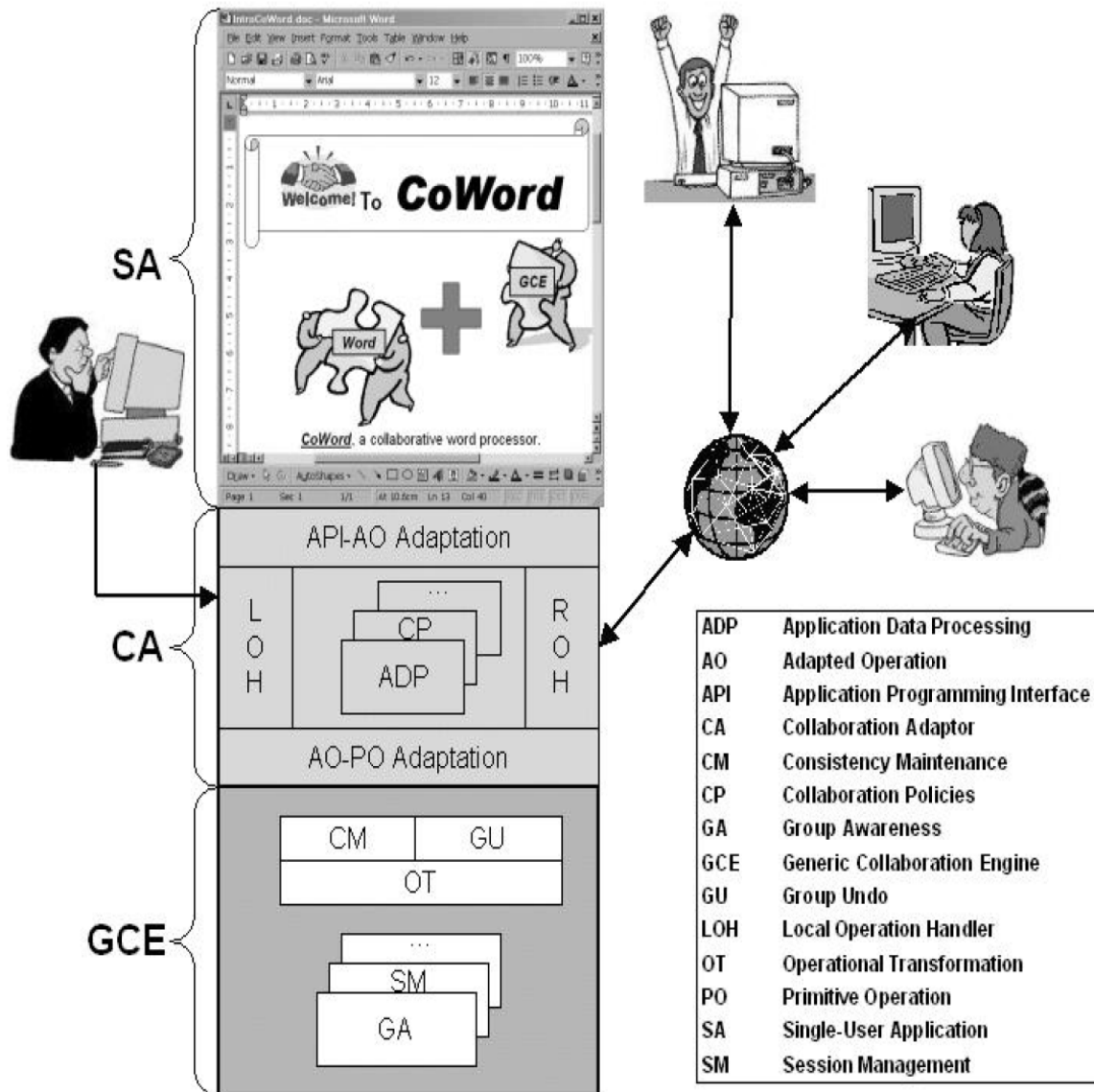


Figure 8 - The CoWord Approach [18]

Additionally, it is advantageous to utilize existing applications that are familiar to users. These applications can be augmented to be utilized in a collaborative environment, allowing users to retain their favored applications while still allowing for collaboration

[45]. Li and Li [45] also allow for heterogeneous collection of client editors in their architecture by providing an event capture-reduction-reproduction mechanism; in this methodology, events are captured and reduced to meta-events, then they are replayed by transforming/reproducing them on the client editor. In this way, multiple users can use a heterogeneous set of editors and still collaborate on a shared document [3]. In this case, as shown in Figure 9, the single application contains single-user semantics and rendering (displaying the state to the user); collaboration can be injected into this single-user application by hooking collaboration semantics that receive a “copy” of the user editing commands. These commands are processed and distributed to other copies of the single-user application.

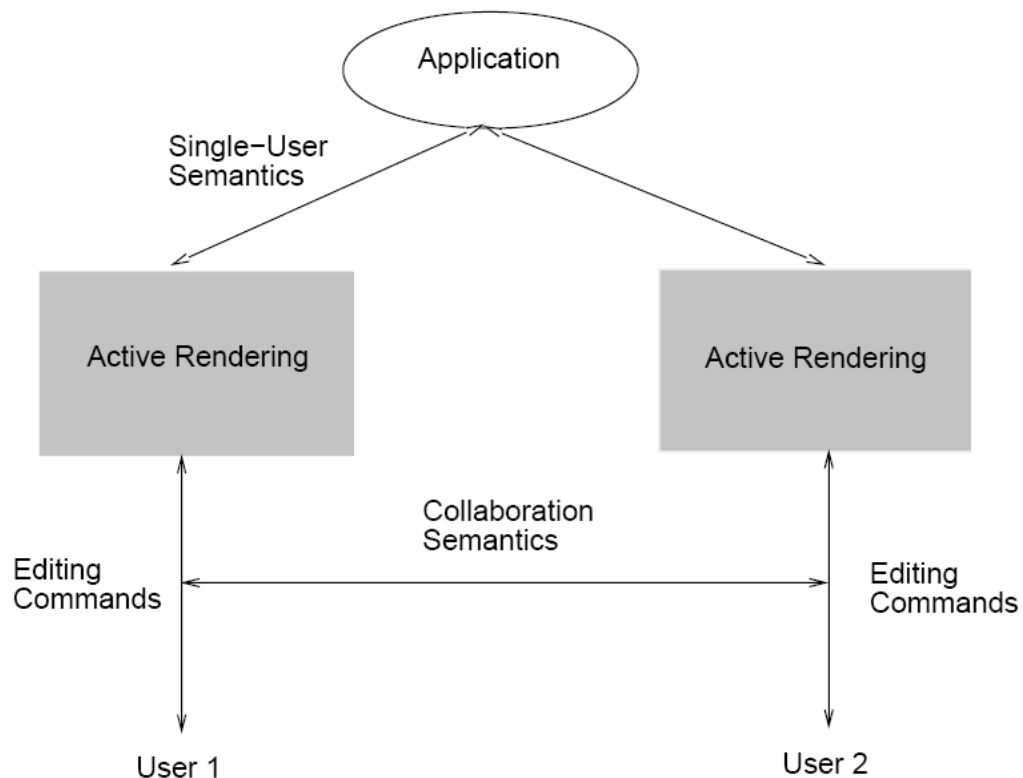


Figure 9 - Generalized Collaborative Architecture from [3]

2.2.4. Transparency, Awareness, and CES Components

Transparent collaborative systems are so named because the applications that are being shared among multiple users have no idea of the collaboration - the collaborative interface acts as an intermediary buffer for the application and receives all users' input and relays these interactions to the application; when the application responds and adjusts its output, the collaborative system/agent relays this information to all users' computers such that all users see the same interface. The advantage of such transparent systems is that they can be integrated into most single-user applications without the need to recompile or edit the original application.

Aware collaborative systems are so named because the collaborative interface is embedded within the application itself and the system's core interface and operations support synchronization and distribution/sharing of the system's content. These systems are defined as aware because the application is "aware" that the content is being shared and the interface of the system enables such sharing. While there are many benefits of embedding the collaboration within the application, the disadvantage is that the source of the application must be available and the collaborative API (synchronization, mutex, etc.) must be tightly coupled within the application. This is often not possible, thus the need for transparent systems.

Application sharing and transparency are two different approaches to collaborative systems. Application sharing involves either centralizing the application's execution and distributing the input and output (display) among user machines or creating a replicated, homogenous architecture in which each user runs the same application across a network; with either model, the user is constrained to use the same application as all other users in

the collaborative environment. Even in heterogeneous application sharing environments, considerable concerns must be overcome in supporting the capture, communication, and replication of users' actions as discussed in the previous section.

In comparison, transparency-based systems allow users to share applications without modifying the original program. Transparencies originally involved screen sharing technologies in which the user would share the entire screen to other users. These systems evolved into sharing only specific windows or applications, rather than the entire screen, and are best represented by the X windows protocol.

Under conventional collaborative transparent system, concurrency is not possible - only one user is able to input to the application at any given time; while this is appropriate for presentations and shared meetings, this is too limiting for collaborative software development. "Floor control" is the term used to define which user has access to the input stream (mutex), and this is needed to ensure that event interleaving is avoided.

One promising concept of being able to merge the best of transparent and aware collaborative systems is the modern object-oriented concept of reflection [81][83]. If a developer wanted to transform a single-user application into a collaborative multiple-user application but did not have access to the source code, then through reflection, the developer could extend the program and add the communication/synchronization API into the system externally via reflection. Unfortunately, this approach does require a high-level knowledge of the internals of the single-user system, and even without access to the original source code, in-depth knowledge of the internals of the system is often required.

An alternative approach would be to design systems that allow users to establish relationships to objects within the system and extend the collaborative software to support such relationships [81]. Of course, the prerequisite of this type of system would be that the collaborative API be built into the current system and that the system supports extension by allowing the user to establish relationships between objects. Li and Patrao's model exhibits such an interface by viewing the elements of the collaborative interaction as objects that support emergent sharing and distributed referential integrity. Such objects inherit common attributes and provide a generalized API for modification such that these modifications (small differentials) can be broadcast to the users of the system and tracked; this avoids the more costly low-level messaging (transparency-based) system wherein all display information is broadcast.

Li and Li [45] discuss current advances in the area of transparencies that should support spontaneous application sharing (i.e. a user can use a single-user application and then later decide to publish/share the application to another user) and support heterogeneous clients and independent views. Additionally, the issue of "late comers" needs to be addressed in modern collaborative environments: how can the system bring new users that were not present at the beginning of the session up to speed quickly; OS hooks such as the Microsoft Windows API provides such capabilities that allow collaborative transparencies to record sessions for replay on future, late arriving clients.

Begole et al [80][88] discuss a synchronous methodology for providing a "transparent" collaboration system that works in coordination with existing applications. This system is different from other existing collaboration transparencies in that it avoids the "conventional" centralized architecture that require that only one person interact with the

system at any given time (single token-based mutex). One difficulty that is avoided in such single-controller transparent collaborative systems is that of interaction interleaving; since only one user can “control” the cursor, then interactions cannot be interleaved incorrectly (i.e. the input is by definition sequential in nature and no undesired overlap is possible).

Four attributes are useful in comparing aware and transparent collaborative systems [80].

Table 1 - Comparing Transparent and Aware Collaborative Systems

	Transparency	Aware
Concurrent Work	Single	Multiple
WYSIWIS	Strict	Relaxed
Group Awareness	Little	Detailed
Network Usage	High	Low

These attributes are defined as:

- **Concurrent work:** Does the system allow for multiple users to provide input simultaneously, or is only one user able to provide input at any given time?
- **WYSIWIS:** All users should see the same state at all times; What You See Is What I see.
- **Group Awareness:** How much detail does the system provide with regard to what other users in the system are doing and what section of the document they are viewing? Some systems simply provide a pointer/cursor showing the current “location” of the other users; other systems provide thumbnails and more detailed views.

- **Network Usage:** How much network bandwidth is consumed and needed by the system? In aware systems, operations are typically all that is communicated (and these messages are small), whereas in transparent systems typically rely upon centralized server architectures and broadcast display change information (quite large).

Aware collaborative systems consume less bandwidth, allow for concurrent work, more easily provide flexible WYSIWIS interfaces, and allow for more inherently robust group awareness. Transparency-based collaborative systems are useful in situations where the developer needs to create a collaborative system based upon a single-user application but does not have access to the underlying code base of the single-user system; transparency-based systems often consume more system resources and require a centralized server model, but they are often the only option in some circumstances.

Another model to define CSCW systems is Patterson's [75] that defines groupware into four levels: **display** (renders the application to the user), **view** (contains the application's logical presentation), **model** (the application's state and internal information), and **file** (the persistent information of the application). Based upon these four levels, three different variations can be described. The **shared model** is one in which the different users each have their own displays and views, but the model and file levels are combined in a centralized server. The **shared view** is one in which each user has a separate file, model, view, and display, but the models and views utilize communication mechanisms to ensure consistency. The **hybrid model** is one in which the file and model are centralized and shared on a server, but the system allows for different views and displays

(and views are coordinated via communication to ensure consistency). These configurations are displayed in Figure 10.

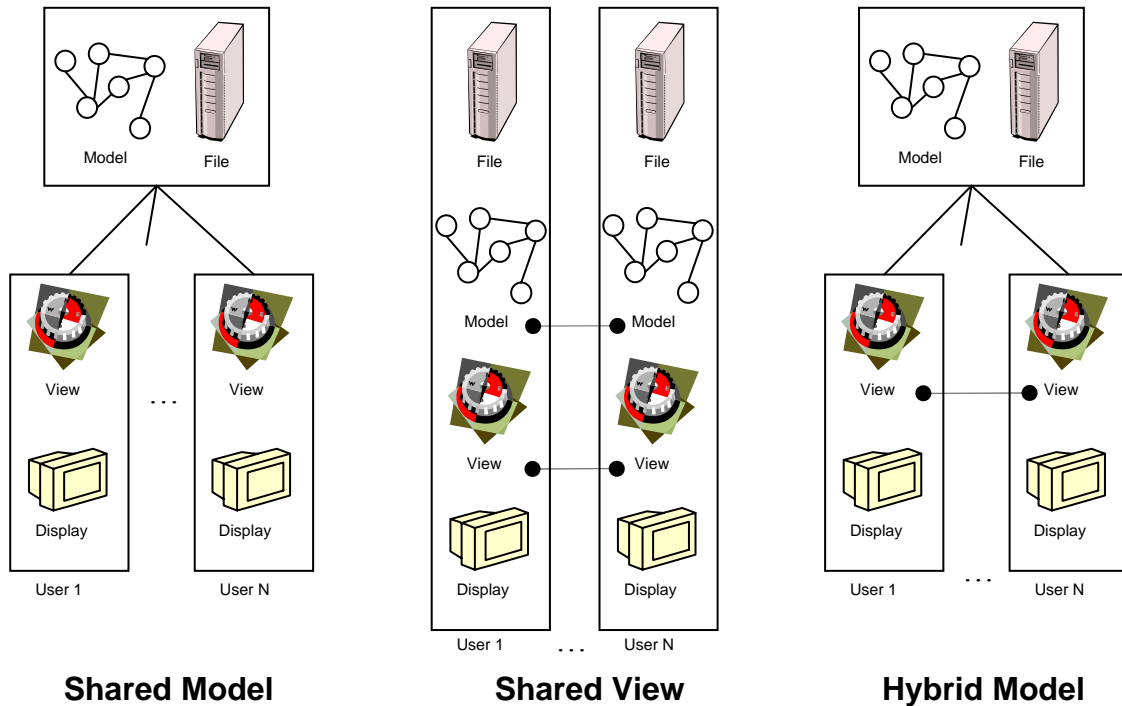


Figure 10 - Distributions of Models, Views, and Displays

Other modern models include the window system and coordination agent/subsystem that communication to the presentation and functional core aspects of the model. Based upon this view, the system can be central (contain server that maintains all state), direct communication (a peer-to-peer system), hybrid (combination of server and peer-to-peer), asymmetrical (in which the server resides on a user's machine), and multiple servers (in which there is a hierarchy of servers and communication layers) [75]. Of course, other permutations of the placement of these CES components are possible, and a goal of modern CSCW architectures is to accommodate modular components that can accommodate a wide range of computation, data management, communication, and

application components [84]. To increase reuse of CES components, Geyer et al [87] advocate aggregating components in an object-centric architecture and allowing each CES component to control access, rights, etc. This model is similar to a Web-services approach, and coordination among such objects is critical to achieve successful utilization of the components. Mehra et al [96] propose such a Web Services-based architecture as shown in Figure 11.

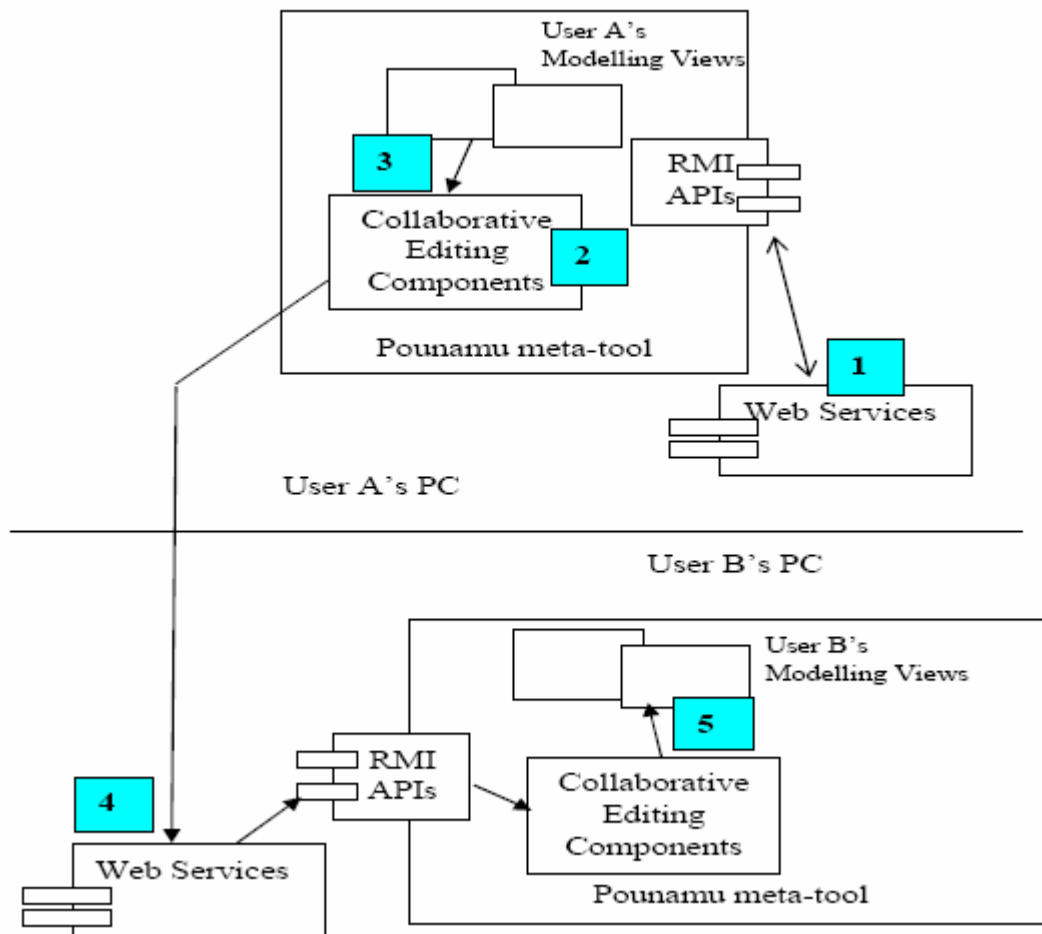


Figure 11 - A Web Services-based Collaborative Editing Architecture [96]

2.3 Existing Applications

IRIS is a project that supports CSCW and CES through the use of optimistic concurrency control and multicast for communication; this project supports synchronous and asynchronous collaboration but does not offer specific conflict resolution algorithms (instead, resolution is left up to the users as in CVS and RCS). Private local edits can be made and selectively published (with conflict resolution possibly needed), but no algorithms to handle such events are presented [61].

Concerning notification mechanisms, others have examined how to ensure that users of the system are kept up-to-date with respect to asynchronous editing (not real-time, concurrency management). Work such as [63] and [66] present customizable notification mechanisms by which users may be notified when a document is changed through a variety of interfaces.

Existing IDEs such as Eclipse and Visual Studio provide the ability to extend the IDE and add new functionality. Jazz is one such project that adds the capability of CES into the Eclipse IDE. Jazz supports awareness, communication (via chat and annotations) and coordination (informal via communication – not through concurrency control mechanisms) [64].

Existing applications such as CoWord [18], CoPowerPoint [95], and CoStarOffice [94] all allow multiple users to coordinate shared authoring of a document, but each of these systems employ an architecture that only allows a homogeneous collection of client applications. Further, CoStarOffice requires explicit, token-based turn taking for coordination.

3. Research Goals

We propose the following three research goals to address the outstanding issues identified in the preceding sections.

1. Develop a set of algorithms and services that maximize concurrent access to shared documents while minimizing communication costs. The outcomes of this goal will provide a solid theoretical foundation and framework for the development of the architecture and prototype (goals 2 and 3).
2. Develop an architecture wherein existing client editing systems can connect to existing server document repositories. This architecture will allow users to synchronously and asynchronously edit documents, subscribe for notification upon document change through varied communication mechanisms (email, IM, etc.), and reflect an open-systems approach whereby future server and client tools can be integrated easily into the architecture (flexibility in design). Of key importance is to examine the proper placement of the algorithms and data structures developed in goal 1 to maximize the efficiency of the algorithms and communication.
3. Develop a prototype system that demonstrates the effectiveness of the architecture developed in goal 2. This prototype will connect a heterogeneous set of client editors residing on the user machine to a heterogeneous set of document repositories residing on distributed server machines and will allow for performance analysis of our architecture and algorithms.

Scope of Research Goals

Maximizing concurrent access and minimizing communication: We believe there exists an optimal concurrency model that will allow the maximum number of clients to share a document while not incurring an excessive amount of communication across the network (consequently decreasing the response time on the users' machines). The research goal of minimizing communication costs while maximizing access will be achieved by developing a set of core algorithms and data structures that are independent of any specific client or server technology and ensure optimal access to shared documents. We expect to leverage well-established cache policies within our algorithms to minimize communication among clients and the server. The outcome of this research goal will be achieved by analyzing various combinations of our concurrency algorithms with client cache policies and finding a balance between how much information is stored on the clients and how much is centralized on the servers. Preliminary work in leveraging semantic knowledge of the document to create a tree-based view of the shared document shows promise in improving access, and we will continue to pursue the algorithms and data structures to achieve semantic-aware, tree-based concurrent access coordination. Our currently-developed algorithms work on binary trees, and we intend to improve these algorithms to support all trees.

While many operational transformation (OT), pessimistic (lock-based) concurrency, and optimistic (merge-based) concurrency policies are well established, we will examine how a combination of these approaches with the integration of dynamic, multi-granular locking can achieve superior concurrent access and avoid costly communication policies like multicast updating [98]. We expect that the solution to maximizing concurrent

access and minimizing costs will entail a combination of lock-based (dynamic) and optimistic merge/OT solutions, thus examining how to best combine these approaches is a key goal.

Given that our system will allow heterogeneous client and server technologies, we must develop transformations from specific client and server tools into a “meta language” that captures the events and allows these events to be propagated among all users’ tools. Consequently, we will research and develop general-purpose algorithms that facilitate these transformations.

Theoretical goals are:

1. Extend our document to tree algorithm to support general trees (not limited to binary trees)
2. Extend our deadlock-free lock management algorithms to support general trees (not limited to binary trees)
3. Research how to best combine locking and OT solutions to maximize concurrent access and minimizing communication costs
4. Generate data structures that support the efficient storage of ownership information in a distributed fashion (not necessarily at a central server) to minimize communication cost when updating this information
5. Generate technology-independent event transformation algorithms that may be applied to specific client and server technology transformations

Heterogeneous Architecture: Other research has investigated and designed various architectures to support collaborative editing [10][18][21][27][71] [95]; the major

contributions of our proposed architecture is the ability to connect heterogeneous clients and servers and the integration of the algorithms to achieve maximum concurrency through dynamic locking. Our preliminary work in this area has centered on an open-system, Web-services approach, and we expect our resultant architecture to have a similar structure wherein common components will work with all client and server technology combinations with the insertion of application-specific hooks that communicate with the existing heterogeneous client and server tools. Our architecture will allow clients from different operating systems and different editing tools to communicate with distributed servers running various operating systems and configuration management suites. Any number of clients and servers may be supported simultaneously. Our architectural components will integrate with existing editing and server repository tools with no modifications to the original tools. An additional outcome of this research goal will be to establish a necessary and sufficient application programmer's interface (API) for coordinating collaborative editing among clients and servers; this API may then be implemented as a functioning system (see next goal). Of equal importance in the development of the architecture is to examine and properly place the algorithms and data structures developed in goal 1. For example, should a central server store all lock and position information, or should this be distributed among all clients within the system?

Architecture goals are:

1. Define an architecture such that technology-specific events are propagated via technology-independent components to heterogeneous client and server technologies.

2. Determine the proper location of the algorithms and data structures defined in the theoretical goals to minimize communication cost among the components of the architecture.

Prototype System: The architecture we develop will be validated by implementing client-side and server-side components that integrate into existing client editing software and existing server repositories. We have previously written peer-to-peer software to replicate changes by clients as well as written hooks into Microsoft Office products. It is important to allow users to retain use of their favored editing software – and we assume the editing software is proprietary; as a result we will connect into existing editors without modifying them. It is our intention to use Web services to develop a server-side API to which our clients may connect. The Web service will then connect to various server repositories such as Visual Source Safe (VSS) and Concurrent Versions System (CVS) and check-out and check-in documents via proxy; consequently, no changes will be made to the server repositories. We intend to measure response times and communication costs for various cache policies and verify the algorithms, data structures, and architecture have reasonable performance.

Prototype goals are:

1. Implement the components defined in the architecture goals; Web services will be utilized to pass client-side events to the coordination server. This coordination server will then pass events to and from server-side repositories
2. Implement two client-side technology-specific event transformations (MS Word and a text editor) and pass these events to the coordination server

3. Implement two server-side technology-specific event transformations (MS Visual Source Safe and CVS)
4. Analyze the communication and computation effectiveness of the system

4. Current Research Summary

This section summarizes the current state of our research. Section 4.1 discusses the algorithms and data structures we have developed for managing multi-granular locks and optimizing concurrent document access. Section 4.2 discusses the preliminary work we have achieved in combining the best practices of other architectures and integrating them in supporting heterogeneous client and server technologies. Section 4.3 demonstrates a prototype peer-to-peer update system that employed varied locking policies on the shared document and our current work in hooking into existing client editing tools.

4.1 Theoretical Framework: Algorithms and Data Structures

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.

4.1.1 Document Partitioning

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 12 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.

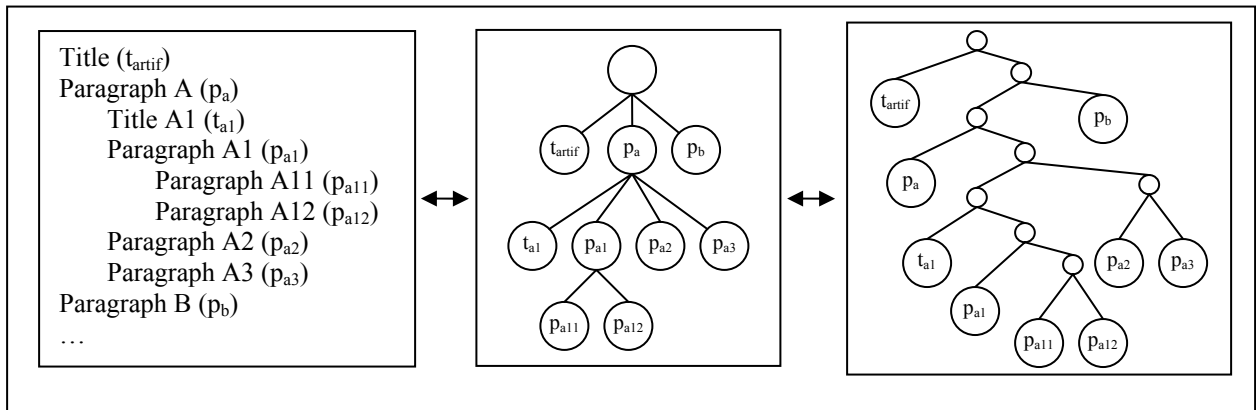


Figure 12 - Mapping from document to tree to binary tree

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 denotes move left, and a 1 denotes move right). Thus leaf node identified with "1011" would be right, left, right, right (Figure 13). 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.

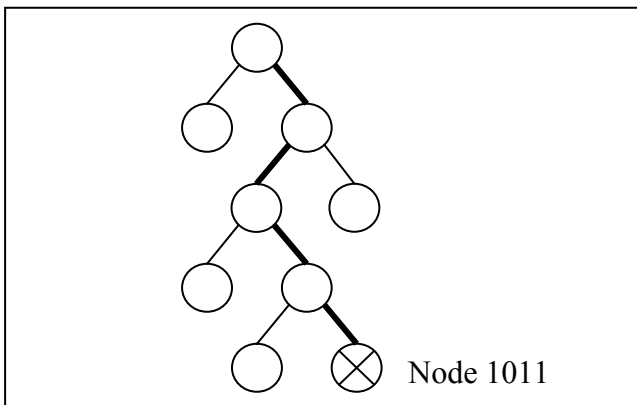


Figure 13 - Path to uniquely-identified node

4.1.2 Deadlock-free, Multi-granular Locking

Data Structure

Each node in the tree representing the document is colored in such a 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 Figure 14. Note that we do not show symmetric equivalents.

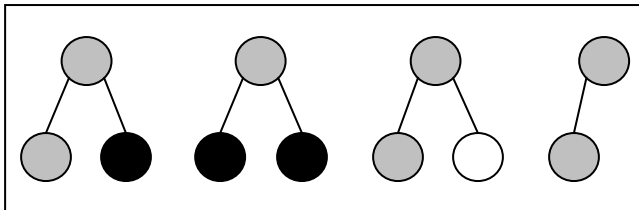


Figure 14 - Grey Node Configurations

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)

Algorithms

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).

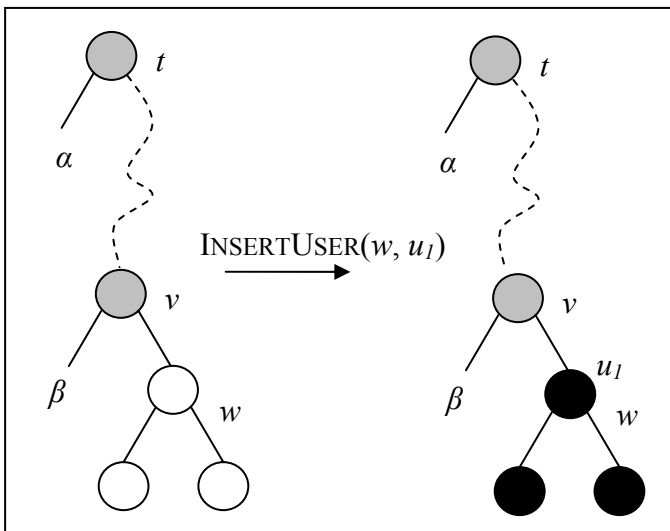


Figure 15 - The INSERTUSER operation without demotion

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 Figure 15). 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 16, u_2 wanted x and u_1 wanted w , so the conflict was resolved, but as you can see in the algorithm in Figure 17, 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 Figure 15 and Figure 16 – 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.

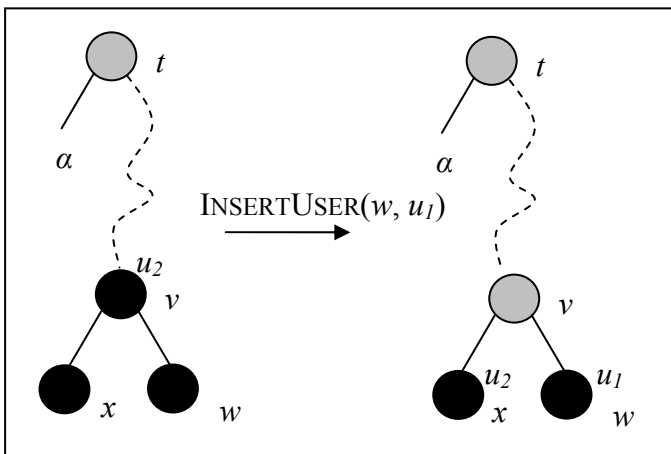


Figure 16 - The `INSERTUSER` operation with demotion

```

INSERTUSER( $w, u_i$ )
    if  $w.owner \neq u_i$ 
        RECURSEINSERT(ROOT,  $w, u_i$ )

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

SETOWNER( $w, u_i, r$ )
     $w.color = \text{black}$ 
     $w.owner = u_i$ 
     $w.originalRequest = r$ 

```

Figure 17 - 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 18. 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 19. 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 (Figure 18 and Figure 19) 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.

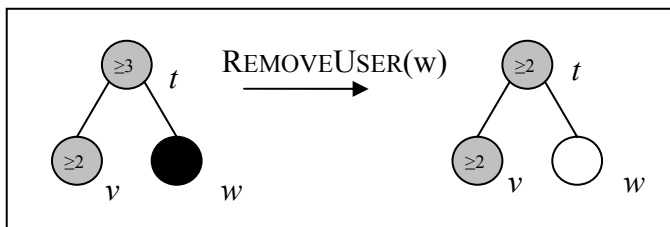


Figure 18 - Case 1 of the REMOVEUSER operation

In the above case (Figure 18), 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.

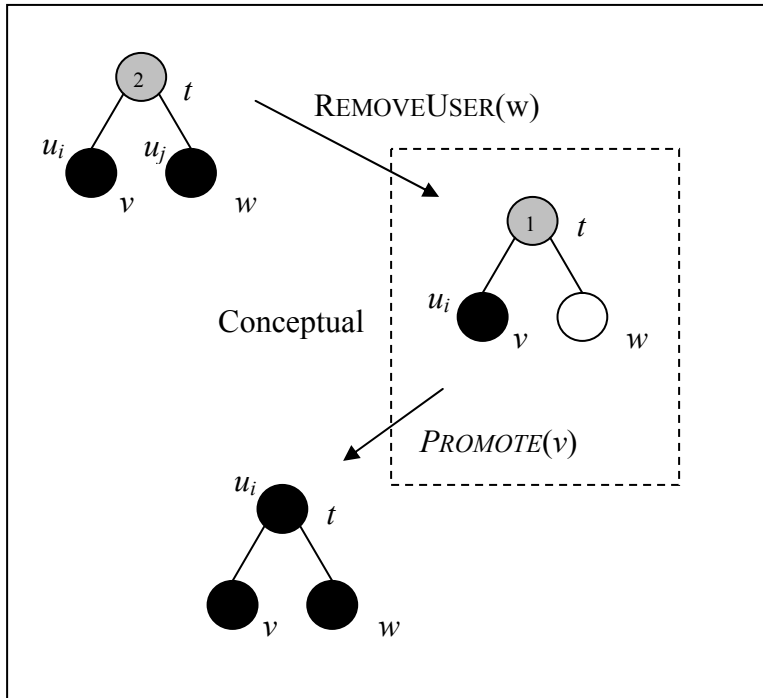


Figure 19 - Case 2 of the REMOVEUSER operation

Note that in the above case (Figure 19), 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 20.

```

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

RECURSEREMOVE( $n, w, u_i$ )
  if  $n.color = \text{black}$  and  $n.owner = u_i$ 
    then RELEASEOWNER( $n$ )
  else if  $n.color = \text{grey}$ 
    then  $n.greyCount = n.greyCount - 1$ 
    if  $n.greyCount = 1$  and  $w.sibling.color = \text{black}$ 
      then SETOWNER( $n, w.sibling.owner, w.sibling$ )
    else if  $n.greyCount = 1$  and  $w.color = \text{black}$ 
      then SETOWNER( $n, w.owner, w$ )
    else if  $n.greyCount = 0$ 
      then RELEASEOWNER( $n$ )
    else RECURSEREMOVE(NEXTINPATH( $n, w$ ),  $w, u_i$ )

RELEASEOWNER( $w$ )
   $w.color = \text{white}$ 
   $w.owner = \text{NIL}$ 
   $w.originalRequest = \text{NIL}$ 

```

Figure 20 - 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 its 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 20.

The following (Figure 21) shows the execution of $\text{REMOVEUSER}(w, u_1)$ 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.

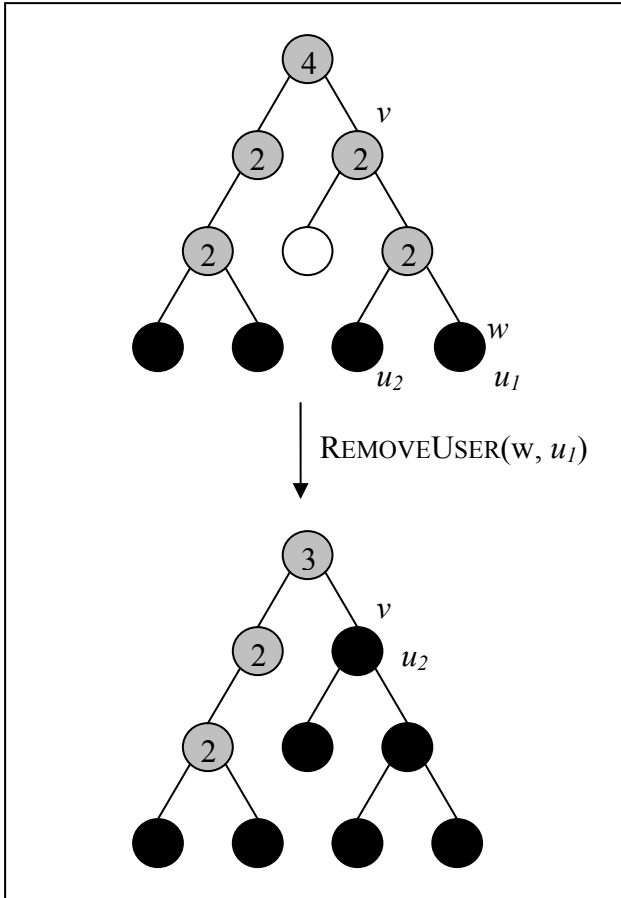


Figure 21 -REMOVEUSER (with multi-level promotion)

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 $O(1)$.

4.2 Heterogeneous Architecture

Motivated by the need to support concurrent, collaborative access to shared documents, we have designed and validated an architecture that integrates existing and familiar systems for client software and document repository. Through Web-services, we achieve an open system wherein numerous clients can use varied editing tools to fit their preferences and access documents distributed on a heterogeneous collection of document repository systems (configuration management systems, or CMS). Our simulation results on numerous client/server configurations validate our architecture and demonstrate an increase in concurrent access to shared documents; by adding a Lock Manager to the server, our system achieves a 67% reduction in check-out failures.

4.2.1 Proxy-based Configuration Management

The addition of the fine-grain lock manager proxy to the server machine allows for the addition of fine-grain check in and check out of artifacts. This proxy intercepts messages from the network and processes them accordingly. The proxy maintains a set of artifacts that have been checked out from the server; this stored database of artifacts also contains information about subsections within the artifacts. This subsection management allows a client to check out only a subsection of an artifact and allows other clients to check out other subsections. Consequently, the middleware proxy will only check in an artifact if there are no clients accessing the artifact. Assuming pessimistic locking, a check out

request is only passed to the server from the proxy if there are no other clients currently accessing the subsection being requested.

The result of this additional proxy is that each artifact may be checked out simultaneously by different clients so long as the clients are accessing disjoint subsections of the artifact. Notice in this scheme, no change is required to the existing CMS system; the addition of multi-granular locking is transparent to the existing CMS system.

4.2.2 Multi-granular Locking Simulation

As seen in Figure 22, our architecture consists of four components that connect with each other and to existing client applications and server repositories.

First, the **Client Application Listener** component connects to existing client applications such as MS Word and IDEs like JavaBeans so that users may use their current, preferred methods of editing. The role of this component is to listen to change events that occur within the application (edits to the document) and cache (if desired) and send on these changes to the server coordinating the collaborative editing among other users. This component also receives update notifications from the server and sends the changes to the client application, thus maintaining consistency among all users collaborating together.

Second, the **Web Service** component provides an API for traditional CMS systems (check-in and check-out, etc.) as well as an API for managing changes among the users that are collaborating together (insert, delete, move, etc.). This component also provides an API by which users can subscribe to receive synchronous and asynchronous notification when a document has been changed.

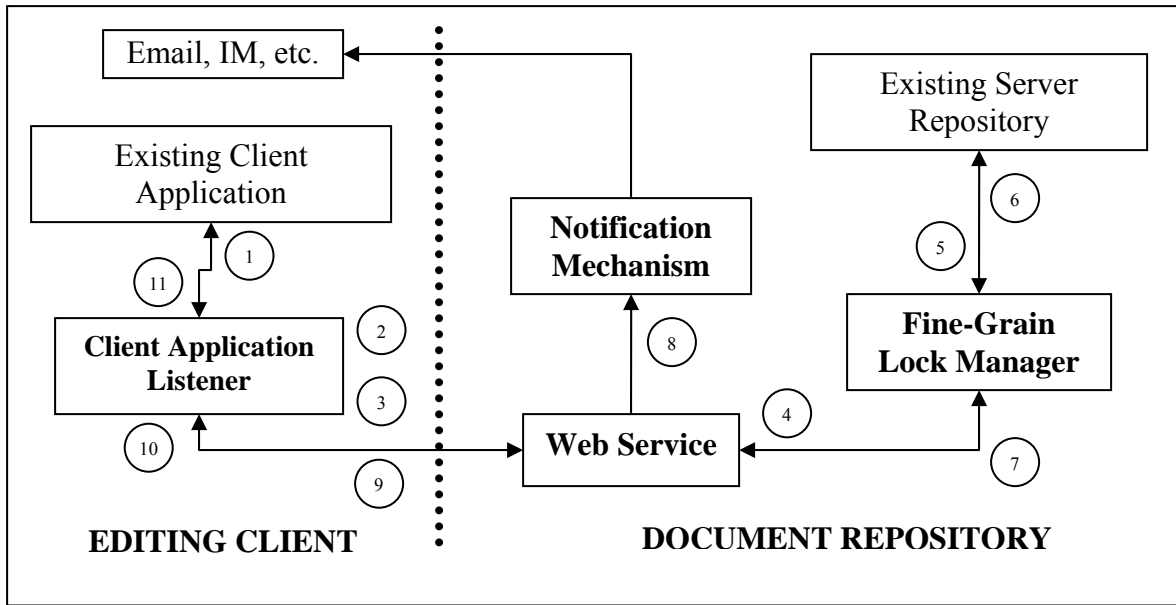


Figure 22 - Open-system architecture for distributed repositories

Third, the **Fine-Grain Lock Manager** component acts as a proxy that checks-out and checks-in documents from the existing server repository (such as CVS, VSS, etc.). This component receives check-in and check-out events from the Web Service component and processes and executes these requests via the existing server repository. This component provides the ability to manage artifacts at a finer granularity (viewing an artifact as a collection of sub-artifacts); as an example, a user can edit page one of a shared artifact at the same time another user is editing page two. This component tracks who is currently working on each artifact in the server repository and is thus able to “push” these changes to the necessary clients. A more detailed description of this component is provided in Section 3.2.

Fourth, the **Notification Mechanism** component is responsible for passing on any events that the user has requested notification of (document change, check-out, etc.) to the users’ preferred email, IM, etc. This component receives the event from the Web Service component and sends the notification to the client. Clients may subscribe for notification

when changes are made (even if they are not currently editing the document); thus the system supports synchronous and asynchronous collaboration.

In summary, heterogeneous editors are able to coordinate by sending messages to the server via an established API. Since the server provides the common API, any client IDE can connect if it utilizes this API. The server propagates changes to other users and maintains consistency among all users' copies of the artifact as needed. The system tracks who is currently working on each artifact in the server repository and is thus able to "push" these changes to the necessary clients.

Event Flow

The following 11 events are illustrated in Figure 22. When a change event occurs in the client's document editing application, a state update message (user edit of artifact) is sent (1) to the Client Application Listener. The Client Listener receives the update message and caches the change (2). When the cache must be flushed (when the cache is full or when another user enters the document as a reader), changes are sent (3) to the Web Service on the server via the network. The Web Service receives the updates and sends (4) them to the Fine-Grain Lock Manager to be processed. Upon receipt of a check-out or check-in message, the Fine-Grain Lock Manager updates its data store of users that must be notified of the change and may also send (5) the check-out or check-in message to the existing Server Repository. The Server Repository (an existing CMS or document server) processes the check-in or check-out and confirms (6) update of the artifact to the Fine-Grain Lock Manager. The Fine-Grain Lock Manager notifies (7) the Web Service component that the change has been committed (the check-in or check-out has succeeded). For each client subscribed for notification concerning this document being

changed, the Web Service component sends (8) a message to the Notification Mechanism (which will notify the client). Additionally, the Web Service component selectively broadcasts (9) via the network change notifications to each client interested in the change (and client currently reading the document being modified). The Client Application Listener will receive the update notification (10) and cache it if the user is not currently viewing the updated section. When the client views the changed section of the document, the Client Application Listener flushes the update cache to the Client Application (11); this maintains consistency as the user views the content of the shared document.

To validate our architecture and experimentally determine whether this middleware proxy approach could improve concurrency, we simulated two configurations of our architecture – one in which the middleware was absent (as in a traditional distributed repository) and one in which the proxy was present (as serving to implement fine-granular locking). We utilized the DEVS Java simulation framework for this study [11]. Both simulations connected numerous clients to a set of servers hosting CMS (document repositories) through a network. Clients would simulate users requesting documents, editing a document once owned, and returning the document to the repository when the edits were completed (checking the document back in).

The second simulation configuration was identical to the first except that this system added a middleware component to the server that intercepted document requests from clients and processed these requests as a proxy to the server. Note that if the middleware was not present, the Web Services API would communicate directly with the document repository (CMS).

There are three types of clients in the simulation: random, clustered, and hybrid.

- The random client has a high probability (90%) of selecting a new random artifact from the repositories from the full range of all of the documents.
- The clustered client is programmed to exhibit a localization policy in that it remains within a close proximity to a single document. We achieve this by sequentially numbering the documents, so this client would check out documents numerically close to its currently preferred document.
- The hybrid client is programmed as a mixture of the clustered and random client behaviors – behaving like each of them 50% of the time.

The simulation was run in nine configurations for each of the two versions of the simulation (for a total of 18 runs).

Table 2 - Architecture Simulation Results

Test	Iterations	Client Distribution (# per type)			Repository Distribution (# Artifacts at each Server)								
		Random	Clustered	Hybrid	S1	S2	S3	S4	S5	S6	S7	S8	S9
1	500	1	1	1	1	2	1						
2	500	3	0	0	1	2	1						
3	500	0	3	0	1	2	1						
4	500	0	0	3	1	2	1						
5	500*	1	1	1	10	20	10						
6	500	10	10	10	30	50	80	30	30	40	40	100	100
7	5000	10	10	10	30	50	80	30	30	40	40	100	100
8	2500	10	10	10	15	25	40	15	15	20	20	50	50
9	5000	1	1	1	1	2	1						

* Test 5 for the fine-grain version was run to 5000 iterations to obtain lock failures

Table 2 demonstrates the various configurations. The number of iterations is defined by the number of iterations for which the simulation was run (all time advances). The client distributions denote how many of each type of client (random, clustered, and hybrid) were in the system when the simulation was run; for example, for test 1, there was one client of each of the three types. The repository distributions denote how many artifacts existed at each server and how many servers existed in the system; for example, in tests 1-4, there was one artifact at server 1, two artifacts at server 2, and one artifact at server 3. Check-out fail rates for the simulation configuration without the fine-grain locking ranged from 2% (test 5) up to 33% (test 1). Check-out fail rates for the simulation configuration with the fine-grain locking ranged from 0.75% (test 1) up to 12% (test 2). In all configurations, the version of the simulation that contained the fine-grain locking proxy significantly outperformed the other version (without the middleware) in reducing the number of check-out failures (collisions). The minimum improvement when adding the fine-grain locking in reducing check-out failures occurred in test 2 (50% improvement), and the maximum improvement occurred in test 1 (78% improvement). The average improvement in reducing check-out failure as a result of adding the fine-grain locking was 67%.

This work has shown that the hypothesis behind adding middleware to existing repository management systems is sound and that fine-grain management of artifacts via proxy does improve the reduction of failed check-outs (collisions) among multiple users in a distributed collaborative system.

In all test scenarios, dramatically fewer check-out failures occurred in the fine-grain locking version of the simulation as compared to the initial version of the simulation

without fine-grain locking. This is as expected as the middleware, fine-grain version of the simulation effectively increases the number of artifacts (via subsections of the artifacts) that clients are able to simultaneously check out; this is due to the fact that checking out a subsection of an artifact does not preclude another client from checking out a different subsection of the same artifact.

Additionally, this work shows that the number of failed check-outs is related to the relative density of clients when compared with artifacts; note that test 1 and 5 differ only in the number of artifacts stored in the server machines (by a factor of 10). The check-out fail rate decreases dramatically as the number of artifacts is increased. This is as expected since the clients have a wider range of artifacts from which they may select.

The results also indicate that the improvement in moving from the initial simulation to the fine-grain enabled simulation is comparable regardless of the number of iterations to which the simulation is run. This claim is supported by examining the comparable improvements between test 6 and test 7 (in which only the iterations was changed).

Additionally, the results indicate that the concurrency is maximized at some number of artifacts relative to the number of clients. Examining the difference between test 7 and test 8, the decrease in the number of artifacts by 50% does not show any appreciable difference in the improvement rate. Consequently, we may infer that both of these tests had a sufficiently large set of artifacts from which the clients could make use that the check-out failure rate was not affected by the reduction in the number of artifacts. It is interesting to note that the improvement rate is still significant when the proxy middleware is added, even though the number of artifacts is large enough to handle the client requests well in both simulation configurations.

4.3 Prototype System

We have completed preliminary work on various aspects that can be incorporated into the prototype testbed system. We developed a peer-to-peer collaborative editing system that allows users to lock documents on a line-level (to avoid edit collisions), and changes to the shared document are sent in real-time; since we are using a pessimistic concurrency control mechanism in this prototype, we avoid the need to merge or perform OT. Communication analysis and performance are presented. Second, we have examined how to create hooks into existing applications so that our completed prototype can accommodate a heterogeneous set of existing client and server technologies.

4.3.1 Peer-to-Peer Communication Analysis

This section discusses a research study in the development of a collaborative system that provides peer-to-peer interaction on a shared document. Central to the study is ensuring consistency among users' copies of the document being edited and maintaining low network communication overhead during the collaboration. Various locking/mutex granularities are explored. The paper concludes with open questions and future research areas that are evidenced by the current version of the system.

This study was motivated by the exploration of synchronous collaboration for software development and document development. The generation of software code and documents in general share much in common, thus this study will focus on general document editing and how a synchronous collaboration tool can benefit such interactions. Further, past collaborative systems typically provide mutual exclusion and locking at a file level only. This study is also motivated by the premise that various locking

granularities deserve examination; how does locking granularity affect the ability for users to collaborate effectively?

When designing the collaborative system for this study, a few critical considerations are essential.

First, the system should allow for peer-to-peer interaction as a client-server model provides for a central bottleneck of communication and failure; also, since collaborations are often spontaneous, the system should support quick establishment of communication, and the peer-to-peer model affords this.

Second, the system should allow for editing a shared document and allow users to be added and removed at any time. Additionally, while a host peer may initiate the collaborative session, the host should be able to leave and the session remain (i.e. allow for host migration to another peer in the system).

Third, communication via text messages (chat) and audio/voice should be included to facilitate non-document-based messages and information to be shared. Ideally, visual/video information could also be incorporated. In essence, the system should provide an interaction environment that as closely as possible mimics “face to face” interactions.

Finally, the system should ensure that all users see the same content at any time; it is critical that the system provide consistency among all users such that the “are you looking at what I’m looking at?” problems do not interfere with the collaborative interaction.

It is believed that if these elements are achieved, then the users will have a positive, efficient and effective interaction and will be able to collaborate in real time (synchronously) even though they may be distributed geographically.

Synchronizing Upon Entry

When a peer enters the system, there is inconsistency among the shared collaborative windows. All peers that existed in the system prior to the new peer's entry should be synchronized, but the new peer does not contain any data in its shared space. Thus the new peer must initiate a synchronize request; this request is fulfilled (answered) by the host (the first peer of the system). The entire content of the shared space is transferred to the newly arrived peer. While this communication can be costly, this ensures that the new peer receive all the current data in the shared space.

While not implemented in this system, alternatively, each peer could maintain a cache copy of the last known state of the shared data and only update actions could be sent to the peer. Once this set of actions was received by the newly arriving peer, the new peer could "replay" the actions locally to achieve consistency.

Regardless of which method is employed, once the initial synchronization and consistency is achieved only small change notifications need be sent among peers to maintain consistency.

Ensuring Consistency

The cornerstone of this collaborative system is to ensure that all peers share the same state of the shared collaborative document. This prototype takes the approach that all updates should be immediately broadcast to all peers – in $O(p)$ time, where p is the

number of peers in the system. Since these update messages are small, the communication cost of maintaining consistency is also small; and since this is a user-input dependent system, the delay in communication is reasonable.

Tests were run on a simulated 33.6kbps modem connection with 2% packet loss, and update/consistency times were less than a second. While it would be best if such updates were instantaneous, sub-second delays in an interactive system may be tolerated if the users are informed of such delays a priori.

Other more complicated mechanisms exist when ensuring consistency. For example, if a user is viewing (and editing) a section of a document that is not related to the section another user is editing, why must these updates be broadcast immediately? Delaying the change notification until the user's view entered the modified section of the document would reduce the need for immediacy in the update notification; also, such delay of notification could afford batch updates and reducing the number of network packets needed. While such a model is plausible, the system developed in this study utilizes the immediate update model.

Various Locking Granularities

Rather than lock the entire file, why not allow various users to edit the document synchronously? The system designed in this study allows for various locking policies.

Initially this study was to explore four levels of locking: no locking, word-level locking, line-level locking, and paragraph-level locking. Only two of these four locking granularities were implemented: no locking and line-level locking.

The first implementation did not ensure locking at all; any user could edit any portion of the shared collaborative window at any time. It was assumed in this model that the

probability of edit collisions would be minimal, thus why lock users from editing any portion of the document. The probability of edit collisions increases as the size of the document decreases, and the probability of edit collisions increases as the number of users increase. Thus edit collision is inversely proportional to document size and proportional to number of users.

The second implementation locked at a line level. Thus only one user can “own” any given line in the system, and a first-come first-served policy is enforced. The current implementation does not pass ownership from one peer to another when a user changes lines in the shared window; thus if peer A owns line X, and peers B and C request line X, when peer A moves to another line, line X is not owned by any peer. Line X will be owned by the next peer to request the line (i.e. no queue is maintained for any line).

Communication Analysis

Communication dominates this application as computation/runtime time is reliant upon user input – which is quite slow relative to the computation speed of the processor. Consequently, the analysis of this system focuses on the communication overhead of ensuring consistency among peers and ensuring that the state of the peers is maintained.

When a new peer joins an existing collaboration space, DirectX 9 generates 24 packets of data for a total of 1770 bytes transmitted. This does not establish consistency among the shared content window but is merely the overhead associated with the peer joining. Consequently, a peer join activity generates a quite reasonable load on the network.

Though other messages are sent and managed by the DirectX 9 code for establishing connections and joining, there are only five types of data packets that are sent in this system:

- A peer has joined and requests a SYNCH
- The host responds to the SYNCH request
- A peer as placed content into the chat window
- A peer has updated its position (line owned) in the shared content window
- A peer has modified the shared content window

Let us examine each of these scenarios in turn with respect to communication cost.

Peer SYNCH Request and Host SYNCH Response

When a peer requests a SYNCH request, the peer is asking for the host to respond with the current state of the shared content window (we do not worry about synchronizing the chat window in this application as is). The current implementation sends this request to all peers as the system does not maintain the host ID locally, thus a broadcast message for update is submitted to all peers. This request involved sending a packet to all peers from the requesting peer; the host will then respond with sufficient packets to transmit the content of the shared content window to the peer that requested the SYNCH.

In test results, running two peers generated 3 packets and a total of 56 bytes transmitted as overhead (when the shared space was empty). Running three peers generated 5 packets and a total of 93 bytes transmitted.

When 68 bytes of data occupied the shared space, a SYNCH request (among three peers) generated 5 packets for a total of 159 bytes transmitted. This is nearly exactly the overhead (93 bytes) plus the payload (68 bytes).

Thus the request communication is $O(p)$ where p is the number of peers in the system; this could be improved to $O(1)$ if each peer were to maintain the ID of the host. The response communication is $O(n)$ where n is the size of the shared collaborative window.

Chat

When a chat message is generated by a peer, the text that is typed by the user is sent to all other peers (including the host). In test results with three peers, when a 14 byte message is transmitted, 4 packets for a total of 94 bytes are generated. With three peers, when a 1 byte message is transmitted, 4 packets for a total of 68 bytes are generated.

Thus the transmission of a chat message is $O(np)$ where p is the number of peers in the system and n is the size of the chat message.

Position Update

When a user changes the line on which the text cursor rests in the shared collaborative window, all other peers must be updated. The system employs a lookup table at each peer to track which line is “owned” by each peer, thus when a peer changes its current line, a message must be broadcast to all other peers in the system. The content of the message is quite small – represented by an integer – as we are sending the new position of the peer whose position has changed.

In tests with three peers, an update of position message generated 4 packets for a total of 68 bytes.

Consequently, a change in position by a peer in the shared collaborative window generates $O(p)$ communication, where p is the number of peers in the system.

Modified Shared Content

Similarly, when a peer changes the content of the shared collaborative window, all peers must be notified of the change. The message that is sent to the other peers is quite small and consists of

- an integer denoting the start position of the selection that was changed
- an integer denoting the length of the selection that was changed
- the keystroke of the change

This system allows the insertion of a single character into the shared space and the deletion of a single character (or multiple characters). Notice that the overall data payload of the message is at most 9 bytes (or as little as 5 depending upon the size of an integer in the system). Thus the communication overhead is small.

In tests with three peers, a modification message generated 4 packets for a total of 76 bytes.

Thus a modification by a peer to the shared collaborative window generates $O(p)$ communication, where p is the number of peers in the system.

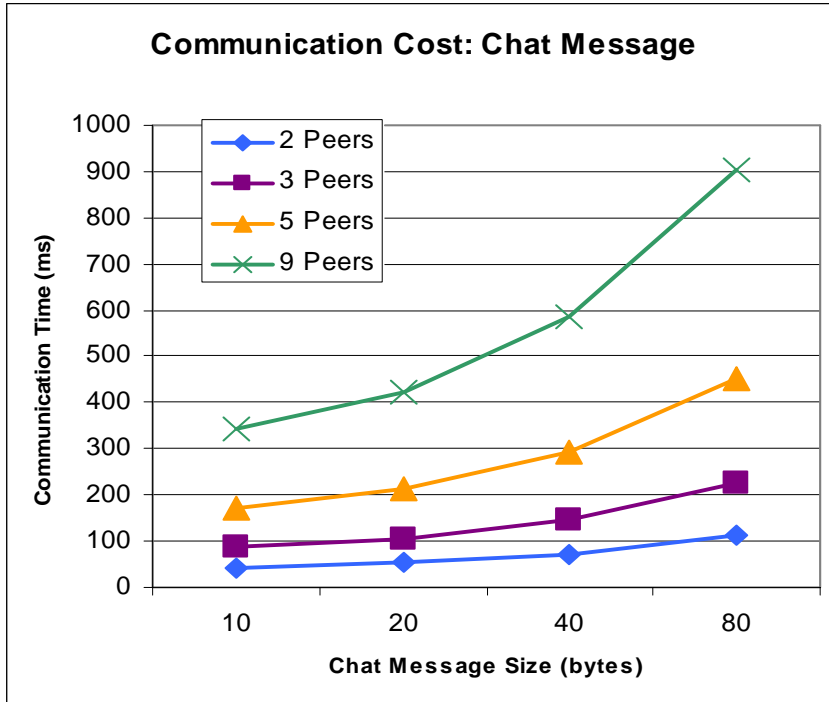


Figure 23 - Peer-to-Peer Communication: Chat

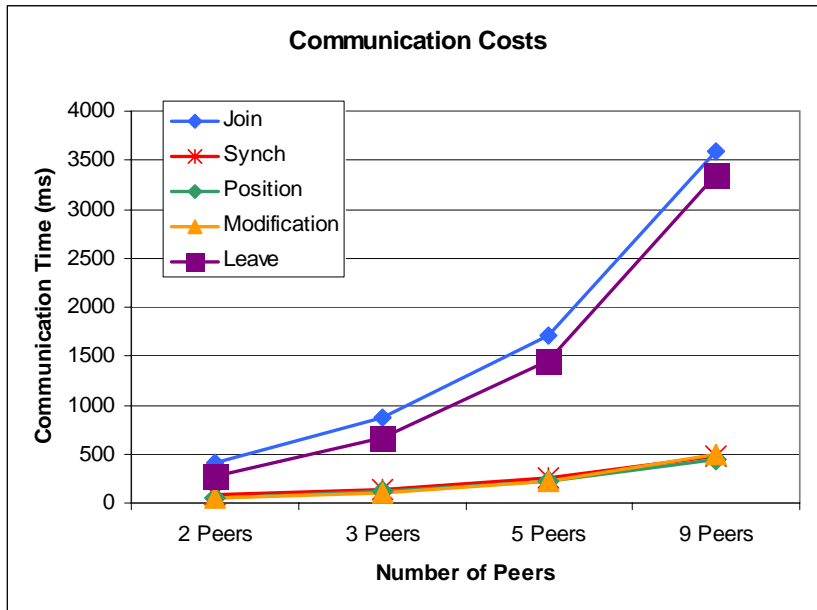


Figure 24 - Peer-to-Peer Communication Costs

Communication Summary

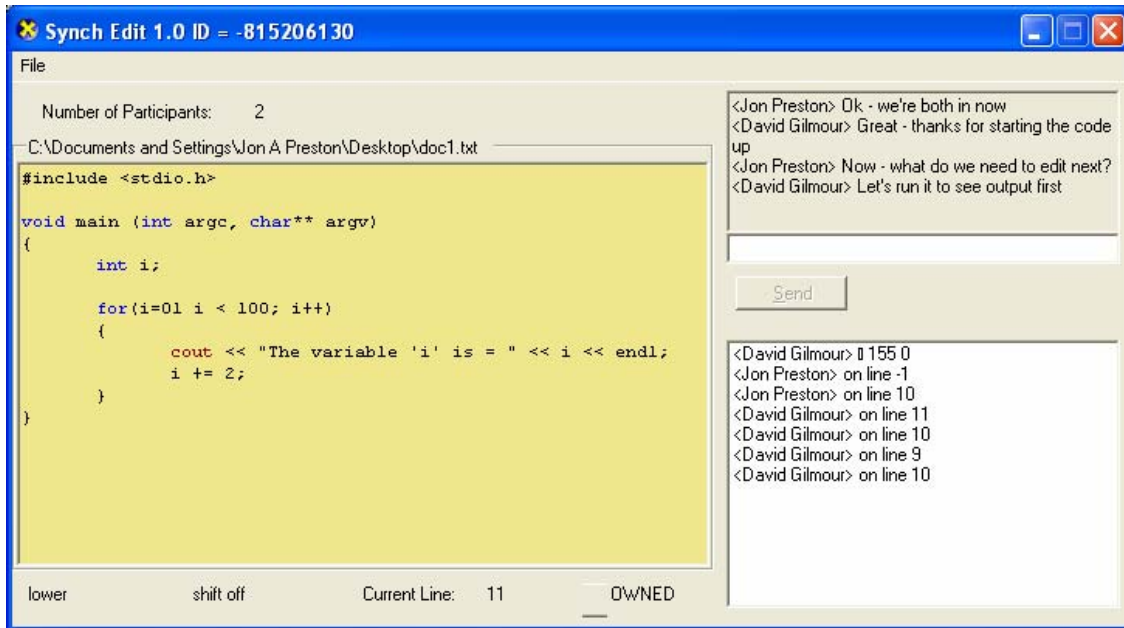
Table 3 summarizes the communication overhead of the collaborative system. The packets and bytes generated are for tests with three peers.

Table 3 - Communication among peers for various activities

Activity	Packets	Bytes	Expectation
Peer joins (DirectX overhead)	24	1770	$O(1)$
Peer SYNCH and response (68 bytes of content)	5	159	$O(p) + O(n)$
Chat input (14 bytes of content)	4	94	$O(np)$
Position update	4	68	$O(p)$
Modification of shared content	4	76	$O(p)$

Technologies Employed

For this prototype, we implemented the code in Microsoft C#. This language provided the rapid application development (RAD) required for the short duration of the study and interfaced quickly with DirectX 9. DirectX 9 was utilized to establish and maintain the peer-to-peer communication of the software. The overhead cost of employing DirectX 9 was minimal.



4.3.2 Hooking into Existing Editors

Similar to [45] our system will integrate with a set of client editing tools. To accomplish this, hook/listener components must be written that will intercept user input and convert these actions into operations that can be replayed on any client editor that our system supports. For example, if a user selects a sentence in an MS Word document and presses the Delete button, these actions/inputs should be mapped to the “Delete Sentence” meta-operation (that could then be translated into the commands native to StarOffice, OpenOffice, etc.).

Preliminary research in this area indicates that the Eclipse and MS Visual Studio 2005 IDEs both provide APIs for developing hooks/extensions into the IDEs; additionally, OpenOffice is an open-source technology and Microsoft Office provides the macro/VBA interface/API, so both of these technologies are extensible. At a lower level, the operating system events could be captured and translated into meta operations through the use of technologies such as WinSpy/Spy++.

5. Intended Research Methodology

This section defines how we will achieve our goals previously expressed in section 3. Section 5.1 discusses our approach in developing the theoretical framework for our work. Section 5.2 discusses our intended approach in defining necessary components to integrate heterogeneous client and server technologies through Web-services. Section 5.3 defines the prototype system we intend to develop and which existing technologies will be incorporated into the prototype.

5.1 Algorithms and Data Structures

Based upon our previous work in this area, we must define the foundational framework on which our architecture and prototype will be built. Managing the complexity of concurrent access to the shared documents, utilizing cache techniques to minimize communication costs and developing notification policies are essential to achieve this goal.

As a result, we intend to:

- i) More rigorously prove the correctness of our existing multi-granular locking algorithm
- ii) Research existing distributed cache techniques and determine which are most appropriate for collaborative editing systems
- iii) Define efficient data structures to store the document state and ownership information in the distributed system
- iv) Analyze the efficiency of the algorithms and data structures to ensure real-time responsiveness for users

5.2 Heterogeneous Open-system Architecture

Building upon the theoretical framework of goal 1, the architecture must support heterogeneity among client editing tools and server document repositories. Notification mechanism and concurrent access APIs must be defined to ensure varied clients may connect with any set of servers that implement the API.

Concurrent Access API: just as a traditional configuration management system must allow for check-out and check-in actions, our architecture must support such actions. Additionally, the API for our system must provide interfaces to update users' active viewspace (aka cursor position within text editors). When change notifications are received, the server will distribute these changes to other users who's viewspaces coincide with the position of the change.

Client Hooks: Since we are not modifying the existing client editing tools, our approach will be to provide hook components that listen to user edit events and capture the intention of such events. For example, if the user selects a set of text and presses the "delete" key, the client hook component must capture this "delete set of text" event so that later (based upon the cache update policies enacted) this change can be incorporated into the server-side copy of the document as well as notify other synchronous users of the change.

Server Proxy: to enable multi-granular locking into existing configuration management systems that currently do not support such locking, we must establish a set of hooks into these existing servers that will check-out and check-in documents via proxy on behalf of the users of the collaborative editing system. These server proxies will then manage the fine-grain locks distributed to multiple client writers to the same document and resolve

lock conflict dynamically based upon our previously-established algorithms to maximize concurrent access.

As a result, we intend to:

- i) Define a necessary and sufficient set of interfaces (API) that cover the core functionality of a semantic-independent collaborative editing system; this will also involve improving our existing version of the heterogeneous architecture
- ii) Establish a protocol by which client components may hook/listen into existing client editing tools, capture edit and viewspace event changes
- iii) Establish a protocol by which server components may hook into existing server document repository tools, sending check-in and check-out events via proxy to these tools

5.3 Prototype System

Once the theoretical foundation and architecture have been established, we will build a prototype that demonstrates the utility of our approach in improving concurrent access for collaborative editing. Such a prototype must integrate into existing editing tools and server repositories, and we will demonstrate the heterogeneity of our system by connecting at least two different editors synchronously to documents distributed within at least two different configuration management systems. Notification policies are not addressed in this prototype as we are not addressing a research question to this field. The following figure demonstrates our intended prototype implementation:

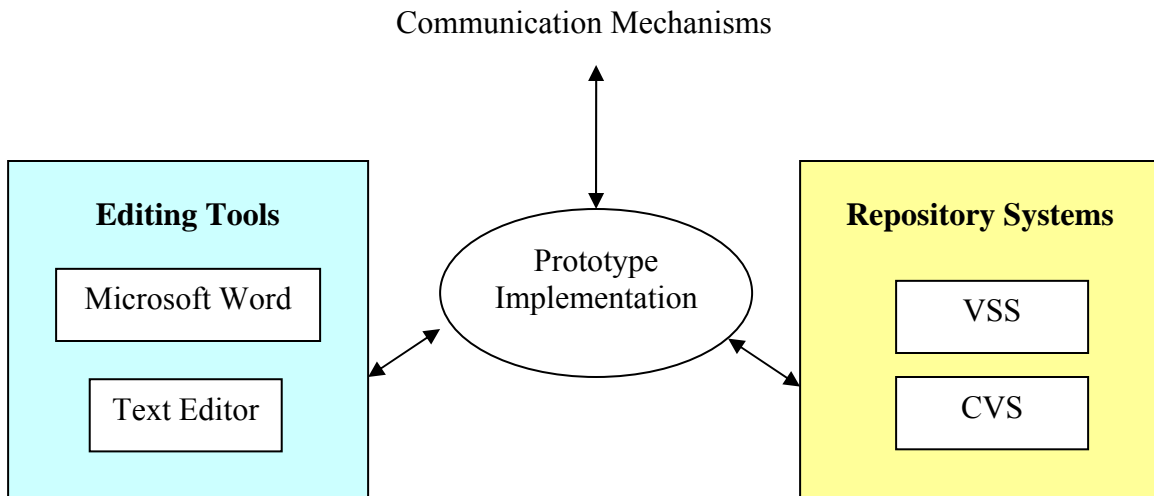


Figure 25 – Prototype Implementation: Technologies to be Integrated

As a result, we intend to:

- i) Implement the architecture defined in goal two
- ii) Implement client hooks into at least two existing client editing technologies
- iii) Implement server proxies into at least two existing server repository systems

6. Conclusion

Computing is increasingly supportive of collaboration among distributed users, and document editing systems are increasingly looking for solutions to allowing multiple authors to create documents synchronously and asynchronously. To date, much work has been done in this field, but there exists an opportunity to improve concurrent access while still maintaining reasonable communication costs – resulting in reasonable real-time responsiveness for users. Concomitantly, we propose to investigate and develop solutions to this open problem. First, we will establish the theoretic foundational algorithms and data structures to maximize concurrent access to shared documents while minimizing communication cost. Initial work in developing a deadlock-free dynamic multi-granular algorithm and tree structure has been presented. Second, we will articulate an architecture that supports heterogeneous client editing tools, heterogeneous server document repositories, and the ability for subscription and notification mechanisms upon document change. Our preliminary architecture supports all three of these, but this architecture will be refined to better accommodate reuse among the client and server components. Third, a prototype that validates the architecture will be developed, and communication and responsiveness data will be measured to demonstrate the viability of our approach. Finally, we have presented our intended scope and methodology to realize our research goals.

7. Bibliography

- [1] Atul Prakash “Group Editors”, Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 103-133.
- [2] Saul Greenberg and Mark Roseman, “Groupware Toolkits for Synchronous Work”, Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 135-168.
- [3] Prasun Dewan, “Architectures for Collaborative Applications”, Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 169-193.
- [4] Paul Dourish, “Software Infrastructures”, Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 195-219.
- [5] G. Henri Ter Hofte, Working Apart Together: Foundations for Component Groupware, Telematica Instituut, The Netherlands, ISBN 90-75176-14-7, 1998.
- [6] Peter Manhart and DaimlerChrysler AG. “A System Architecture for the Extension of Structured Information Spaces by Coordinated CSCW Services”, Proceedings of GROUP 1999, Phoenix Arizona USA, pgs 346-355.
- [7] Kevin L. Mills, “Introduction to the Electronic Symposium on Computer-Supported Cooperative Work”, ACM Computing Surveys, Vol. 31, No. 2, June 1999.
- [8] Arregui, D., Pacull, F., Willamowski, J.: “Yaka: Document notification and delivery across heterogeneous document repositories”. In: Proc. of CRIWG'01, Darmstadt, Germany (2001)
- [9] Anita Sarma and André van der Hoek, “A Conflict Detected Earlier is a Conflict Resolved Easier”, Proceedings of the 4th Workshop on Open Source Software Engineering, Edinburgh, United Kingdom, May 2004.
- [10] Anita Sarma , Zahra Noroozi and André van der Hoek , Palantír: Raising Awareness among Configuration Management Workspaces . In Proceedings of Twenty-Fifth International Conference on Software Engineering, pp 444-454, May 2003, Portland, Oregon.
- [11] Anita Sarma, “A Survey of Collaborative Tools in Software Development”, UCI, ISR Technical Report, UCI-ISR-05-3, March 2005.
- [12] André van der Hoek , David Redmiles , Paul Dourish , Anita Sarma , Roberto Silva Filho , and Cleidson de Souza, “Continuous Coordination: A New Paradigm for Collaborative Software Engineering Tools”, In Proceedings of the Workshop on Directions in Software Engineering Environments, pp 29-36,Edinburgh, United Kingdom, May 2004.
- [13] Dewayne E. Perry and Harvey P. Siy and Lawrence G. Votta, “Parallel Changes in Large Scale Software Development: An Observational Case Study”, International Conference on Software Engineering 1998, pgs 251-260.
- [14] Goopeel Chung and Prasun Dewan, “Towards Dynamic Collaboration Architectures”, Proceedings of the 2004 ACM conference on Computer supported cooperative work, November 6-10, 2004, Chicago, Illinois, USA. pgs 1-10.
- [15] Perry, D.E., H.P. Siy, and L.G. Votta, Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Transactions on Software Engineering and Methodology, 2001. 10(3): p. 308-337.

- [16] Mens, T., A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 2002. 28(5): p. 449-462.
- [17] Ronald van der Lingen and André van der Hoek, "Dissecting Configuration Management Policies", *Proc. of the International Conference on Software Engineering Workshops: Software Configuration Management 2001*.
- [18] Steven Xia, David Sun, Chengzheng Sun, David Chen and Haifeng Shen, "Leveraging Single-user Applications for Multi-user Collaboration: the CoWord Approach", *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, Chicago, Illinois, USA, 2004. pgs 162-171.
- [19] J. Estublier. Defining and Supporting Concurrent Engineering Policies in SCM. *Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.
- [20] Shengzheng Sun and Clarence Ellis, "Operational Transformation in Real-Time Group Editor: Issues, Algorithms, and Achievements", *Proceedings of 1998 ACM Conference on Computer Supported Cooperative Work*, Seattle USA, Nov 14-18, pgs 59-68.
- [21] C. Sun, X. Jia, Y. Zhang, Y. Yang, D. Chen: "Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction*, Vol.5, No.1, March, 1998, pp.63-108.
- [22] C. Sun, X. Jia, Y. Zhang, Y. Yang: "REDUCE: a prototypical cooperative editing system," *Proceedings of the 7th International Conference on Human-Computer Interaction*, pp.89-92, San Francisco, USA, Aug. 24-30, 1997.
- [23] C. Sun, Y. Zhang, Y. Yang, D. Chen: "Distributed concurrency control in real-time cooperative editing systems," *Proc. of the 1996 Asian Computing Science Conference*, Lecture Notes in Computer Science, #1179, Springer-Verlag, Singapore, pp.84-95, Dec. 1996.
- [24] C. Sun, Y. Yang, Y. Zhang, and D. Chen: "A consistency model and supporting schemes in real-time cooperative editing systems," *Proc. of the 19th Australian Computer Science Conference*, Melbourne, pp.582-591, Jan. 1996.
- [25] C. Sun and R. Sasic: "Optional Locking Integrated with Operational Transformation in Distributed Real-Time Group Editors," *In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*. pp.43-52, Atlanta, GA, USA, May 4-6, 1999.
- [26] Muhammad Younas and Rahat Iqbal, "Developing Collaborative Editing Applications using Web Services", *The Fifth International Workshop on Collaborative Editing, ECSCW 2003*, Helsinki, Finland, September 15, 2003
- [27] Knister, M. and Prakash, A., *DistEdit: A distributed toolkit for supporting multiple group editors*. In *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, pages 343–355, Los Angeles, California, October 1990.
- [28] A. van der Hoek and et.al. Continuous coordination: A new paradigm for collaborative software engineering tools. In *Proceedings of Workshop on WoDISEE*, Scotland, 2004.

- [29] M. Locasto et al. CLAY: Synchronous Collaborative Interactive Environment. *The Journal of Computing in Small Colleges*, vol. 17, issue 6, pp. 278-281, May 2002.
- [30] Korel, B. et al. Version Management in a Distributed Network Environment. In *Proceedings of the 3rd International Workshop on Software Configuration Management*, pp. 161-166, May 1991.
- [31] Magnusson, Boris, Asklund, Ulf. Collaborative Editing – distribution and replication of shared versioned objects. *European Conference on Object Oriented Programming 1995*, in *Workshop on Mobility and Replication*, Aarhus, August 1995.
- [32] Magnusson, Boris, and Guerraoui, Rachid. Support for Collaborative Object-Oriented Development. *International Symposium on Parallel and Distributed Computing Systems (PDCS'96)*, Dijon, France, September 1996.
- [33] de Souza, Cleidson R. B., Redmiles, David, and Dourish, Paul, "Breaking the code", moving between private and public work in collaborative software development, *Proceedings of the 2003 international ACM SIGGROUP conference on Supporting group work*, November 09-12, 2003, Sanibel Island, Florida, USA.
- [34] Velazquez M. *A Survey of Distributed Mutual Exclusion Algorithms*. Colorado State University Department of Computer Science Technical Report CS-93-116, September 1993.
- [35] Walter, J. et al. A K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks. *Principles of Mobile Computing '01*. Newport, Rhode Island USA. 2001.
- [36] Bulgannawar, S. and Vaidya, N. A Distributed K-mutual Exclusion Algorithm. *International Conference on Distributed Computing Systems*, pp. 153-160, 1995.
- [37] Shari Lawrench Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall, New Jersey, NJ, 1998, pgs 1-34.
- [38] Roger S. Pressman. *Software Engineering, A Practitioner's Approach: Sixth Edition*. McGraw Hill, Boston, MA, 2005, pgs 596-613 and 739-765.
- [39] Jon A Preston, Exploring Communication Overheads and Locking Policies in a Peer-to-Peer Synchronous Collaborative Editing System. *ACMSE 2005*, Kennesaw State University, GA, USA. Poster presentation.
- [40] Jon A Preston and Sushil K Prasad. A Web-Service-based Open-Systems Architecture for Collaborative Editing Systems. *Proceedings of CollabTech 2006*, Ibaraki, Japan, 2006, pending acceptance.
- [41] Jon A Preston and Sushil K Prasad. A Deadlock-Free Multi-Granular, Hierarchical Locking Scheme for Real-time Collaborative Editing. *Proceedings of the 7th International Workshop on Collaborative Editing Systems*. Sanibel Island, FL, 2005.
- [42] Greenberg, S. and Marwood, D. Real time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM conference on Computer-Supported Cooperative Systems*, November, 1994, 207-217.

- [43] Gu, N., Yang, J., and Zhang Q. Consistency Maintenance Based on the Mark & Retrace Technique in Groupware Systems. Proceedings of GROUP 2005, November 6-9, 2005, Sanibel Island, Florida, USA. pgs 264-273.
- [44] Wang, X., Bu, J., and Chen C. A New Consistency Model in Collaborative Editing Systems. Proceedings of the 4th International Workshop on Collaborative Editing. New Orleans, Louisiana, USA, 2002.
- [45] Li D. and Li R. *Transparent Sharing and Interoperation of Heterogeneous Single-User Applications*. In Proceedings of CSCW'02, New Orleans LA, pp. 246-255, November 2002.
- [46] Tigris.org. Subversion.
- [47] Borland.com/jbuilder. Borland JBuilder.
- [48] Microsoft.com/windowsserver2003/technologies/sharepoint. Microsoft SharePoint Services
- [49] Groove.net. Groove Networks.
- [50] Chu-Carroll, Mark C., Wright, James, and Shields, David. Supporting Aggregation in Fine Grain Software Configuration Management. SIGSOFT 2002/FSE-10, pp. 99-108. November 18-22, 2002, Charleston, SC, USA.
- [51] 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.
- [52] Magnusson B., Asklund U., and Minör S. *Fine-Grained Revision Control for Collaborative Software Development*. In Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering, vol. 18, issue 5, pp. 33-41, December 1993.
- [53] Cheng L. et al. *Building Collaboration into IDEs*. ACM Queue. December/January 2003-2004. 40-50.
- [54] James D. Herbsleb , Audris Mockus , Thomas A. Finholt , Rebecca E. Grinter, An empirical study of global software development: distance and speed, Proceedings of the 23rd International Conference on Software Engineering, p.81-90, May 12-19, 2001, Toronto, Ontario, Canada
- [55] Donald Norman. Collaborative Computing: Collaboration First, Computing Second. Communications of the ACM. Vol 34, No. 12. December 1991. pgs. 88-90.
- [56] Katherine M. Everitt, Scott R. Klemmer, Robert Lee, James A. Landay. Two Worlds Apart: Bridging the Gap Between Physical and Virtual Media for Distributed Design Collaboration. Proceedings of CHI 2003, April 5–10, 2003, Ft. Lauderdale, Florida, USA. pgs 553-560.
- [57] Borghoff U. and Teege G. *Application of Collaborative Editing to Software-Engineering Projects*. ACM SIGSOFT, 18(3), pp. 56-64, July 1993.
- [58] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. IEEE Software, pages 63–70, 1999.
- [59] Horstmann T. and Bentley R. *Distributed Authoring on the Web with the BSCW Shared Workspace System*. StandardView, vol. 5, no. 1, pp. 9-16, March 1997.
- [60] Grudin J. *CSCW Introduction*. Communications of the ACM, vol. 34, no. 12, pp. 30-34, December 1991.

- [61] Kock M. *The Collaborative Multi-User Editor Project IRIS*, Technical Report TUM-I9524, University of Munich, Aug. 1995.
- [62] Aguido Horatio Davis, Chengzheng Sun, Junwei Lu. Generalizing Operational Transformation to the Standard General Markup Language. Proceedings of CSCW 2002, New Orleans, Louisiana, USA. November 16-20. pgs. 58-67.
- [63] Shen H. and Sun C. *Flexible Notification for Collaborative Systems*. In Proceedings of CSCW'02, New Orleans Louisiana, pp. 77-86, November 2002.
- [64] Cheng L. et al. *Jazz: A Collaborative Application Development Environment*. In Proceedings of OOPSLA'03, Anaheim CA, 102-103, 2003.
- [65] Korel B. et al. *Version Management in Distributed Network Environment*. In Proceedings of the 3rd International Workshop on Software Configuration Management, pp. 161-166, May 1991.
- [66] Glance N. et al, *Collaborative Document Monitoring*. In Proceedings of GROUP'01, Boulder CO, pp. 171-178, September 2001.
- [67] Vidot N. et al. *Copies convergence in a distributed real-time collaborative environment*. In Proceedings of CSCW'00, Philadelphia PA, pp. 171-180, December 2000.
- [68] Hao M. C., Karp A. H, and Garfinkel D. *Collaborative Computing: A Multi-Client Multi-Server Environment*. In Proceedings of COOCS'95, Milpitas CA, pp. 206-213, August 1995.
- [69] Conradi R. and Westfechtel B. *Version Models for Software Configuration Management*. ACM Computing Surveys, vol. 30, no. 2, pp. 232-282, June 1998.
- [70] Y-J. Lin and S.P. Reiss. Configuration management with logical structures. In Proceedings of the 18th international conference on Software engineering, pages 298–307, Berlin, Germany, 1996. IEEE Computer Society.
- [71] Chu-Carroll, Mark C., and Sprenkle, Sara. 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, p.88-97, November 06-10, 2000, San Diego, California, United States.
- [72] Uri Dekel and Steven Ross. Eclipse as a Platform for Research on Interruption Management in Software Development. OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada. Copyright 2004 ACM. pgs. 12-16.
- [73] Teege G. and Borghoff U. W. *Combining Asynchronous and Synchronous Collaborative Systems*. In Proceedings of the 5th International conference on Human-Computer Interaction, Amsterdam Netherlands, pp. 516-521, 1993.
- [74] Nickson R. C. *A Taxonomy of Collaborative Applications*. <http://hsb.baylor.edu/ramsower/ais.ac.97/papers/nickers.htm>.
- [75] Roth J. and Unger C. *An extensible classification model for distribution architectures of synchronous groupware*. 4th International Conference on Cooperative Systems. 2000.
- [76] O'Reilly, C., P. Morrow, and D. Bustard. Improving Conflict Detection in Optimistic Concurrency Control Models. In Proceedings of the Eleventh International Workshop on Software Configuration Management. 2003. Portland, Oregon. pgs 191-205.

- [77] Ciaran O'Reilly: A Weakly Constrained Approach to Software Change Coordination. ICSE 2004: 66-68
- [78] van der Hoek A., Heimbigner D., and Wolf A. L. *A Generic, Peer-to-Peer Repository for Distributed Configuration Management*. Proceedings of the 18th international conference on Software Engineering, pp. 308-317, May 1996.
- [79] Sarma A., Noroozi Z., and van der Hoek A. *Palantir: Raising Awareness among Configuration Management Workspaces*. Proceedings of the 25th international conference on Software engineering, Portland OR, pp. 444-454, May 2003.
- [80] Begole J., Rosson M. B., and Shaffer C. A. Flexible Collaboration Transparency: Supporting Worker Independence in Replicated Application-Sharing Systems. ACM Transactions on Computer-Human Interactions, vol. 6, no. 2, pp. 95-132, June 1999.
- [81] Li D. and Patrao J. Demonstrational Customization of a Shared Whiteboard to Support User-Defined Semantic Relationships among Objects. In Proceedings of GROUP'01, Boulder CO, pp. 97-106, October 2001.
- [82] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the ACM Conference on the Management of Data 1989*, pages 399-407, Portland Oregon, May 1989. ACM.
- [83] Roth, J. and Unger C. *Developing synchronous collaborative applications with TeamComponents*. 4th International Conference on Cooperative Systems. 2000.
- [84] Sunderam V. et al. *CCF: Collaborative Computing Frameworks*. SC'98: High Performance Networking and Computing Conference (Orlando, Florida USA). IEEE. 1998.
- [85] T. Mens. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering, 28(5):449-462, 2002.
- [86] David Sun, Steven Xia, Chengzheng Sun, and David Chen: "Operational transformation for collaborative word processing," *Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work*, Nov 6-10, Chicago, IL USA.
- [87] Geyer W., Vogel J., Cheng L., and Muller M. *Supporting Activity-centric Collaboration through Peer-to-Peer Shared Objects*. In Proceedings of GROUP'03, Sanibel Island FL, pp. 115-124, November 2003.
- [88] Begole J., Rosson M. B., and Shaffer C. A. Supporting Worker Independence in Collaboration Transparency. In Proceedings of UIST'98, San Francisco CA, pp. 133-142, 1998.
- [89] Grinter, R. E. Recomposition: Putting It All Back Together Again. In Proceedings of CSCW'98, Seattle, Washington, USA, 1998. pgs 393-402.
- [90] L. Osterweil. Software processes are software too. In Proceedings of the 9th International Conference on Software Engineering, pages 2-13, Monterey, CA, 1987.
- [91] R.E. Grinter. Using a configuration management tool to coordinate software development. In Conference on Organizational Computing Systems, pages 168-177, 1995.
- [92] D.L. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053-1058, 1972.
- [93] J. Grudin. Csw: History and focus. IEEE Computer, 27(5):19-27, 1994.

- [94] Haifeng Shen and Chun Ti Cheong. CoStarOffice: Towards a Flexible Platform-independent Collaborative Office System. 6th International Workshop on Collaborative Editing Systems. Chicago, IL, USA, November 6, 2004.
- [95] Steven Xia, David Sun, Chengzheng Sun, David Chen, Yanxin Shi. Supporting Interactive Presentations with CoPowerPoint. 6th International Workshop on Collaborative Editing Systems. Chicago, IL, USA, November 6, 2004.
- [96] Mehra, A. et al. Supporting Collaborative Software Design with a Plug-in, Web Services-based Architecture. Workshop on Directions in Software Engineering Environments. ICSE 2004. IEEE. Edinburgh, Scotland, UK. May 23-28.
- [97] Tam, J., and Greenberg, S. (In Press - Accepted May 2005) A Framework for Asynchronous Change Awareness in Collaborative Documents and Workspaces. International Journal of Human Computer Studies, Elsevier
- [98] Chawathe Y., McCanne S., and Brewer E. *RMX: Reliable Multicast in Heterogeneous Networks*. In Proc. IEEE INFOCOM, March 2000.
- [99] Fu S., Tzeng N., and Li Z. *Empirical Evaluation of Distributed Mutual Exclusion Algorithms*. International Parallel Processing Symposium '97. 1997.
- [100] Drury J. *Developing Heuristics for Synchronous Collaborative Systems*. In Proceedings of CHI'2001, pp. 447-448, March/April 2001.
- [101] Walpole J. et al. *A Unifying Model for Consistent Distributed Software Development Environments*. In Proceedings of the third ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, pp. 183-190, January 1989.
- [102] Wu D. and Sarma R. *Dynamic Segmentation and Incremental Editing of Boundary Representations in a Collaborative Design Environment*. Proceedings of the sixth ACM symposium on Solid Modeling and Applications, Ann Arbor Michigan, pp. 289-300, May 2001.
- [103] Buszko, D., Lee W., and Helal A. *Decentralized Ad-Hoc Groupware API and Framework for Mobile Collaboration*. In Proceedings of ACM 2001 International Conference on Supporting Group Work, Boulder, Colorado, pages 5-14, 2001.
- [104] Harrison W. H., Ossher H., and Sweeney P. F. *Coordinating Concurrent Development*. In Proceedings of CSCW'90, 157-168, October 1990.
- [105] C. Sun and D. Chen: "Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems," *ACM Transactions on Computer-Human Interaction*, vol 9, no 1, March 2002. pgs 1-41.
- [106] P.H. Feiler. Configuration management models in commercial environments. Technical Report SEI-91-TR-07, Software Engineering Institute, Carnegie Mellon University, 1991.