6-28-2007

Rethinking Consistency Management in Real-time Collaborative Editing Systems

Jon Anderson Preston

RETHINKING CONSISTENCY MANAGEMENT IN REAL-TIME COLLABORATIVE EDITING SYSTEMS

by

JON A PRESTON

Under the Direction of Sushil K Prasad

ABSTRACT

Networked computer systems offer much to support collaborative editing of shared documents among users. 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 real-time collaborative editing systems (RTCES); yet concurrent access is either limited in existing systems that employ exclusive locking or concurrency control algorithms such as operational transformation (OT) may be employed to enable concurrent access. Unfortunately, such OT based schemes are costly with respect to communication and computation. Further, existing systems are often specialized in their functionality and require users to adopt new, unfamiliar software to enable collaboration.

This research discusses our work in improving consistency management in RTCES. We have developed a set of deadlock-free multi-granular dynamic locking algorithms and data structures that maximize concurrent access to shared documents while minimizing communication cost. These algorithms provide a high level of service for concurrent access to the shared document and integrate merge-based or OT-based consistency maintenance policies locally among a subset of the users within a subsection of the document – thus reducing the communication costs in maintaining consistency. Additionally, we have developed client-server and P2P implementations of our hierarchical document management algorithms. Simulations results indicate that our approach achieves significant communication and computation cost savings. We have also developed a hierarchical reduction algorithm that can minimize the space required of RTCES, and this algorithm may be pipelined through our document tree. Further, we have developed an architecture that allows for a heterogeneous set of client editing software to connect with a heterogeneous set of server document repositories via Web services. This architecture supports our algorithms and does not require client or server technologies to be modified – thus it is able to accommodate existing, favored editing and repository tools. Finally, we have developed a prototype benchmark system of our architecture that is responsive to users' actions and minimizes communication costs.

INDEX WORDS: Real-time Collaborative Editing, Dynamic Hierarchical Locking, Heterogeneous Architecture, Collaboration, Consistency, Distributed System, Peer-to-peer

RETHINKING CONSISTENCY MANAGEMENT IN REAL-TIME COLLABORATIVE EDITING SYSTEMS

by

JON A PRESTON

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2007

Rethinking Consistency Management in Real-time Collaborative Editing Systems

by

Jon A Preston

To my wife, Jennifer; children: Joshua, Micah, Lillian, and Eric;
parents; and family – you are my joy.

Acknowledgements

A task as monumental a doctoral degree may only come to fruition as a result of consistent encouragement and support of many people. First and foremost, I thank my family and, most notably, my wife Jennifer for travailing with me through this arduous task – you're the best! It wouldn't have happened without your love and constant support. Certainly, my parents deserve much praise and thanks for always encouraging me and training me in the way I should go. Russ Shackelford and Melody Moore Jackson were instrumental in supporting me in my Bachelors and Masters studies, and I appreciate their support and leadership in guiding me to continue my research in computing. I also thank the Computer Science faculty at Georgia State and especially my committee members – Raj Sunderraman, Xiaolin Hu, Melody Moore Jackson, and my advisor and mentor Sushil Prasad; I am a better researcher and student of computing because of you. I thank my fellow students within the program at Georgia State and the DiMoS research group for their feedback and insightful questions; my work is better for your influence. And special thanks to Jeff Chastine – through it all, your friendship supported and motivated me to finish this work. I would also like to thank Jan Towslee for encouraging me to begin my PhD work; she was an amazing mentor and a lady of the finest caliber. I also thank the administration and my colleagues at Clayton State University. Most sincerely, I thank God for gifting me with my talents; I continuously refine them to Your glory.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application Programmer’s Interface
CCI	Convergence, Causality preservation, and Intention preservation
CES	Collaborative Editing System
CMS	Configuration Management System
COT	Context-based Operational Transformation
CSCW	Computer Supported Cooperative Work
CVS	Concurrent Versioning System
DEVS	Discrete Event Simulation
DN	Document Narrative
DOM	Document Object Model
EIC	External Internal Coupling
EOC	External Output Coupling
HB	History Buffer
HCI	Human-Computer Interaction
IC	Internal Coupling
IDE	Integrated Development Environment
IM	Instant Message
MVSD	Multi-Version, Single Display
OP	Operation
OT	Operational Transformation
P2P	Peer-to-Peer

RCS	Revision Control System
RST	Rhetorical Structure Theory
RTCES	Real-time Collaborative Editing System
SCCS	Source Code Control System
SCM	Software Configuration Management
TTF	Tombstone Transformation Function
VSS	Visual Source Safe

CHAPTER 1

INTRODUCTION

Imagine a scenario in which a geographically distributed team can work together, sharing ideas, collaboratively editing a shared document in real-time, and interacting as closely and productively as a team of workers within the same room. This is one of the goals of the field of Computer Supported Collaborative Work (CSCW) and in particular the subfield of Collaborative Editing Systems (CES). CES may be synchronous (real-time) or asynchronous in coordinating the collaboration among users; in either case, managing a repository of the shared documents, maintaining consistency among replicas of the documents, and resolving concurrent and potentially conflicting changes to the shared documents is of central concern.

Enhancing communication and collaboration is one of the increasingly popular uses of modern computing technology; we observe that computing technologies are ever more user-centric and allow multiple users to work collaboratively to solve modern, interdisciplinary and complex problems facing the world today. We note that productivity software tools (document authoring, email, Web site management, etc.) increasingly focus on supporting collaboration among multiple users – a welcome addition to their core functionality.

However, the current state of CES research uses ever increasingly complex algorithms to achieve convergence, causal preservation, and intention preservation (see [66], [90], and [131] as examples) and still have limited capacity in achieving intention preservation. Additionally, these systems that are replica-based in supporting concurrency control are costly with respect to communication and computation.

Therefore there exists an opportunity to view Real-time Collaborative Editing Systems (RTCES) systematically – moving beyond OT algorithms and focusing in how viewing the system as a whole may uncover new opportunities for optimizations and new approaches to solving the problem of CCI.

This research explores areas of RTCES that can be improved to be more scalable in supporting larger collaborations (as measured by the size of the documents being shared as well as the number of users within the collaboration). Our research revisits the idea of using locking and intelligently cache operations when possible to reduce communication and computation costs. First, we developed an open systems approach that supports existing client and server technologies. Next, we formally developed our theoretical work in hierarchical locking algorithms and data structures to support caching operations and managing concurrency among the users in client-server and P2P scenarios. Third, we integrate current best practices in Operational Transformation (OT) research into our theoretical work. Finally, we extend our simulation results indicating the viability of our approach into prototypes of client and server technologies to support our approach into RTCES.

This chapter presents the motivation of our research, the current state of the art and its limitations, and then we present our problem statement, goals, and contributions of this dissertation. We conclude this chapter with a discussion of the organization of the remainder of the dissertation.

1.1. Motivation

CSCW and specifically RTCES and CES have a rich history of research and significant contributions in various fields since the 1980s [43][44] [119]. These systems remain collaboration-centric as the computing system merely supports the activity at hand [87]. The following are select example domains in which our research in RTCES 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 [88]. While much work within software engineering involves decomposing large systems into subsystems that can be developed in parallel [96][100][154], much work related to coordination remains a vital part of a software system development project [53][92]. 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 [92]. 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.) [99][101]. 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 [101]. Coordination among developers can be formal or informal and is often driven/defined by the software engineering processes employed with the project [40][147]. 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 [33][52]. Recently, there has been an increase in commercial interest in the field of integrating collaboration mechanisms into integrated development environments [7][14][81], 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 [52]. Additionally, users may wish to edit the shared document synchronously (at the same time) or asynchronously (at different times) [145]. 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 [86][100]. As an example of the need for such collaboration, consider a large research proposal authored by faculty from many different universities. 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 [42][81]. 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 [32]. 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 [157], and our algorithms and models generated in this work easily accommodate this non-spatial organizational structure.

1.2. Current State of the Art

Real-time collaborative editing systems allow multiple users to synchronously edit a shared document in a geographically-distributed environment. In such an environment, there are two approaches in managing the document state as shown in Figure 1. The shared document is either centralized at one location within the collaboration or a distributed replica/copy model may be used wherein each user maintains a local copy of the shared document. Current RTCES research utilizes the distributed replica approach in order to maintain high local responsiveness.

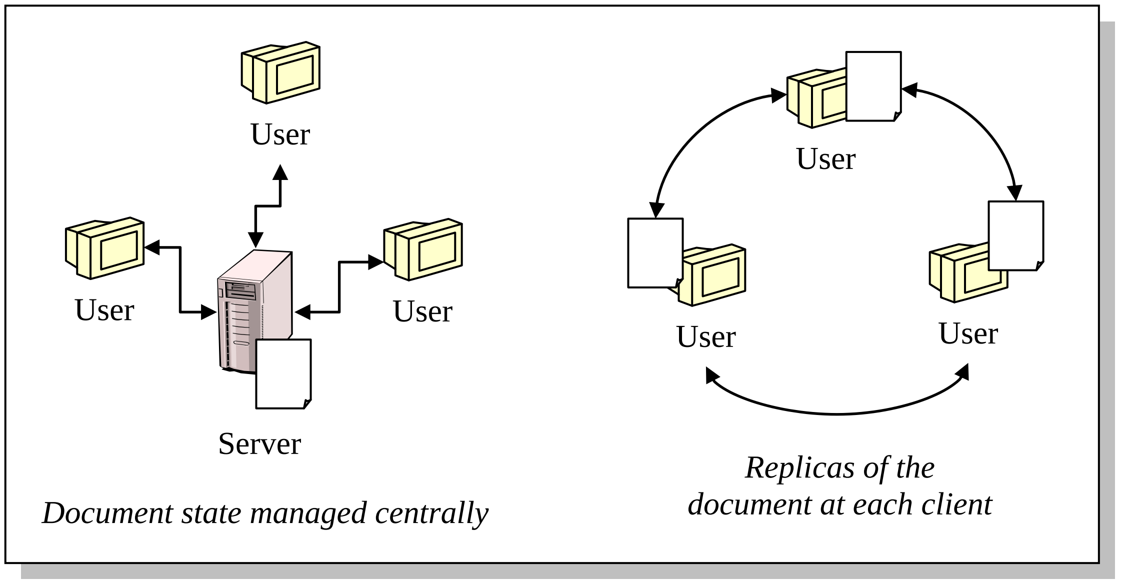

User
User
User
Server
Document state managed centrally
User
User
User
Replicas of the document at each client

Figure 1: Centralized and Replica Document State Management Approaches

Because the current approach in RTCES research is to utilize a replicated architecture, concurrent changes are possible among the users; as a result, concurrency control algorithms must be adopted to ensure the document replicas remain consistent. CCI – convergence, causality preservation, and intention preservation (defined in detail in Section 2.4) – is the current benchmark standard by which RTCES are judged to be correct; thus if a RTCES achieves CCI, then it is said to be correct. Operational transformation (defined in detail in Section 2.5) is the most prevalently researched way to achieve CCI. Briefly, OT involves transforming operations that are created by a remote user that are to be replayed on a local copy of the document; once transformed, the operation may then be enacted on the local replica to achieve the intended result on the document. Without OT, the remote operation, when replayed locally, may not have the same effect as when it was enacted on the remote copy of the document.

1.3. Limitations of Current Technology

This section discusses the limitations of current RTCES architectures and concurrency management techniques.

RTCES Architectures: while the focus of RTCES research has traditionally been on algorithms to better achieve CCI via OT, some research has developed architectural support for RTCES. The client editing and server repository technologies and the connecting network of the collaborative system are for the most part assumed and little work has been done to investigate how these technologies work together to support RTCES. The work of Li and Li [68] focus on supporting heterogeneous client technologies to work together by transforming operations into client technology-neutral "meta" operations that can be incorporated into varied client editing technologies. But this heterogeneous approach has not been extended to server technologies necessary for managing document repositories. Additionally, there has been work to differentiate aware and transparent sharing of documents and workspaces/desktops [2][3], and even some commercial products have emerged from this research [80]. Unfortunately, these architectures employ interaction interleaving, only allowing one user to "control" the cursor and concurrency is not supported. [12] performed an evaluation of RTCES technologies currently developed and being developed (both by academia, industry, and hobbyists), but this work did not perform an analysis of the architectural structure of these systems; it would be fruitful to compare each of these systems to see what architectural components support the collaboration.

Concurrency Management: whether the collaborative system employs a centralized or replication-based approach to managing document state, concurrent access to the shared document must be managed. As mentioned in the previous section, Operation Transformation (OT) is the most popular way to ensure consistency among copies of a shared document in RTCES that employ replication of document state, but OT is costly with regard to computation and communication. Whenever an operation is generated by a user, this operation is broadcast to all other users within the collaboration and replayed locally after being transformed by the other users. Since almost all existing OT solutions view operations at the keystroke level (i.e., the user inserts or deletes a character), the number of messages and the processing of these messages in the RTCES can grow quickly. [57] allows for operations to occur semantically higher than simple characters, but their approach fixes the depth of the document tree – imposing rigid constraints on what operations may be performed – and all operations are still broadcast to all users. Additionally, a history of operations must be maintained at each user's copy requiring storage space for all operations that have been performed in the collaboration; this history of operations is called a "history buffer."

Alternatively, in a centralized approach to document state management, locking may be employed to avoid concurrency problems of the shared document, but such locking techniques as round-robin, token-based, and exclusive locking all reduce concurrent access to the document because only one user may edit the document at any given time. Some systems such as Coven [16] and COOP/Orm [73] attempt to increase concurrent access by reducing the size of the lock (to the sub-file level), but the lock does not adjust in size dynamically with regard to what other users are doing in the collaboration. POEM [71] utilizes the hierarchical nature of software code to lock at a sub-file level, but the methods must be defined a priori by the user (contextually-costly overhead), and again the locks remain fixed in size.

Further, while there has been some preliminary work in examining how semantic structure contained within the shared document can be used [56], no work has been done to investigate how history buffers may be consolidated (reduced) at opportune or predefined times; nor has any research examined how operations stored at one level within the hierarchy of the document may be transformed and combined into operations operational transformation applied within

1.4. Problem Statement and Research Goals

In this dissertation we have focused on the following goals in an effort to solve some of the limitations addressed in the previous section:

1. Investigate how an open systems RTCES architecture may support existing client technologies that connect with existing server technologies with an emphasis on extending legacy server/repository technologies and supporting clients' preferred editing technologies.
2. Revisit the feasibility of utilizing locking to support concurrency management such that communication and computation costs may be reduced when compared to current replication and non-locking approaches.
3. Examine opportunities to leverage semantic knowledge of a document's structure to better achieve intention preservation, apply operations more intelligently at semantically-aware levels within the document, and reduce the

size of the history buffers needed to manage operations within sections of the shared document.

4. Study how the natural structure of RTCES may be supported via a peer-to-peer (P2P) approach that may increase reliability and avoid performance bottlenecks at a single server.
5. Develop prototype implementations of the client and the server technologies we develop that validate our theoretical approach is viable and easily supported in actual, usable tools.

1.5. Contributions and Significance

We have made the following contributions to the field of RTCES in this dissertation work:

1. *An open systems architecture*: we have developed an architecture that allows existing client technologies to connect via Web services API to existing server technologies. Our architecture enables clients to continue to use their preferred editing tools with hooks that capture events and translate them into recognizable messages for others within the collaboration to respond to. Further, our architecture allows existing server repositories of documents to host collaborative editing sessions and manage clients' connections.
2. *Theoretical algorithms and data structures to support dynamic locking*: we have developed a set of algorithms and data structures to support dynamic, hierarchical locking that maximizes the space owned by a user to increase caching and reduce communication costs in a RTCES. We developed client-

server and P2P versions of these algorithms and data structures that are validated empirically via simulation.

3. *Integration of OT best practices and improved CCI*: further, we have integrated best practices of OT techniques into our dynamic locking approach such that concurrent editing of a shared document is supported while minimizing the costs relative to an OT-only approach. Additionally, our approach is semantically aware, so we are able to apply operations intelligently and achieve better intention preservation within a RTCES.
4. *Prototype client and server technologies*: finally, we have developed a functional client editor that connects to a functional Web service API server. These technologies implement our theoretical developments and show that our approach is easily integrated into usable tools for clients to use.

1.6. Organization of the Thesis

The remainder of this dissertation is organized as follows.

Chapter 2 introduces the reader to the background for the research including collaborative editing systems, various architectural approaches to supporting collaboration, locking policies, the CCI model, operational transformation, and existing systems within the field of RTCES.

Chapter 3 introduces the open systems architectural approach we developed to support a heterogeneous collection of client and server technologies. We present our architectural components and the research that validates this approach to real-time collaborative editing systems.

Chapter 4 presents the algorithms and data structures we developed to support relaxed/lazy consistency via hierarchical, dynamic locking on a document tree. We discuss how documents may be modeled as trees, why it is advantageous to maximize the space a user locks within a document, and then present the lock request and lock release algorithms. We discuss our initial simulation results demonstrating that such an approach may reduce communication costs associated with a RTCES, present the correctness and efficiency of these algorithms, and conclude with a discussion of related work.

Chapter 5 extends the research developed in Chapter 4 by showing how our relaxed consistency approach may integrate existing OT algorithms to support concurrent writers and better achieve CCI. We present the improved versions of our approach, and simulation results validating this approach are also presented.

Chapter 6 extends the client-server algorithms of Chapters 4 and 5 into P2P algorithms and data structures. Results of the simulation presented in this chapter demonstrate that this P2P approach is effective in load balancing work among peers and avoiding a single point of failure and bottleneck in processing user actions. We also present a discussion of the correctness and efficiency of our algorithms.

Chapter 7 presents our work in reducing history buffers hierarchically at various depths within the document tree. As a result of this reduction approach, we are able to explore opportunities for better intention preservation. We present simulation results that show how the history buffers are distributed among the peers managing the document tree.

Chapter 8 presents our work in developing prototypes of client and server technologies and the simulation design approach we utilized. These implementations are based upon our previous theoretical work and demonstrate the viability of our approach. The process of moving from models of both the client and the server to fully implemented versions of the client and server technologies is also presented.

Finally, Chapter 9 presents conclusions of this dissertation work and discusses our future research direction.

CHAPTER 2

BACKGROUND

In Chapter 1, RTCES was identified as an active area of research and important field in the future of collaborative and distributed computing. Consequently, the goals of this research focus on viewing RTCES in a systematic way, addressing opportunities for improving architectural structures that support RTCES and reducing communication and computation costs associated with RTCES by addressing fundamental, theoretical algorithms in achieving CCI. To establish a basis by which to evaluate our contributions, we begin by discussing the past work within the field of RTCES research. This chapter presents an overview of collaborative editing systems with an emphasis on real-time collaborative editing systems; we then present the existing architectural approaches to support RTCES and concurrency control policies used in these architectures; next, we define CCI and OT and present current OT approaches; finally, we conclude with a discussion of existing systems – both prototype and commercial.

2.1. Collaborative Editing Systems

Collaborative editing systems may be asynchronous or synchronous (real-time). In an asynchronous collaborative editing system, users collaborate at different times on shared documents. Real-time collaborative editing systems allow users to concurrently share a common document, make changes to this shared document, and have their changes distributed to other users within the system.

Because responsiveness and usability are key components to a real-time collaborative editing system, researchers in RTCES have adopted a replicated approach to RTCES architectures; under this approach, the document is copied to each user's machine, and the users interact with their local copy of the document. When a change (operation) is made to the document, this operation is broadcast to all other users within the collaboration, and the operation is enacted on each user's local copy.

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 [39]. 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 [100]. As Figure 2 demonstrates, the state of the document only converges when concurrent changes are broadcast and ordered in the same total ordering on all clients or else executing A then B on Site 1 and B then A on Site 2 would result in a different state at the different sites and may have unintended results.

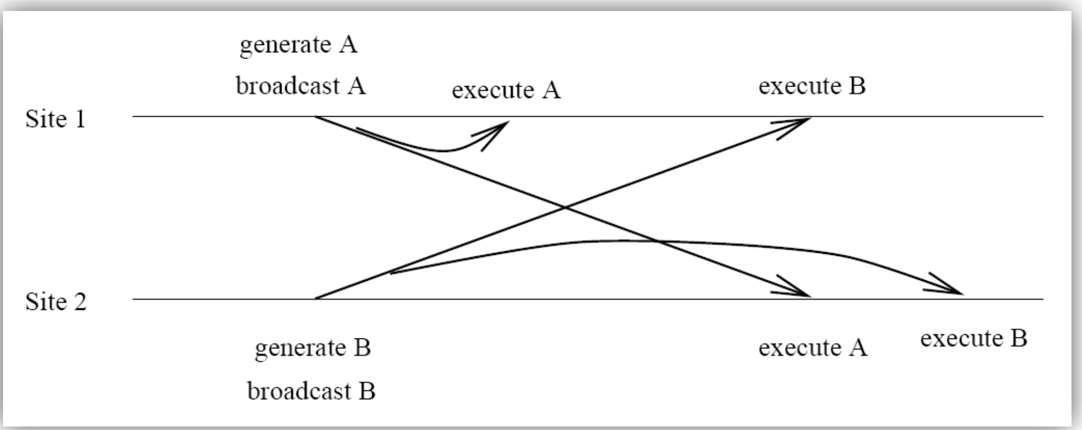


Figure 2: Ordered Broadcast Ensures Convergence

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 as they are best employed in a batch environment where rollbacks are permissible. Alternatively, most collaborative editing systems employ some form of optimistic concurrency control in an effort to improve interactive responsiveness.

2.2. Architectures Supporting Collaborative Editing Systems

[82] performed one of the earliest studies on design for combining synchronous and asynchronous group editing and discovering components of both types of systems. Therein, a model of cooperative work as applied to the task of collaborative writing suggests that mechanisms to support communication among participants and the sharing of a common artifact/document are critical for the success of the CES. While there has been other research to focus on the HCI side of CES (such as communication, awareness, and presence), because this work is focused on systems-level research regarding RTCES such as communication and computation costs savings and improving consistency within a RTCES, this section will focus on such systems-level issues within the scope of RTCES architectures.

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 [69][115]. 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 [69]. 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 [68] 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 [2][3] 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 [3] as shown in Table 1.

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 [116] 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 3.

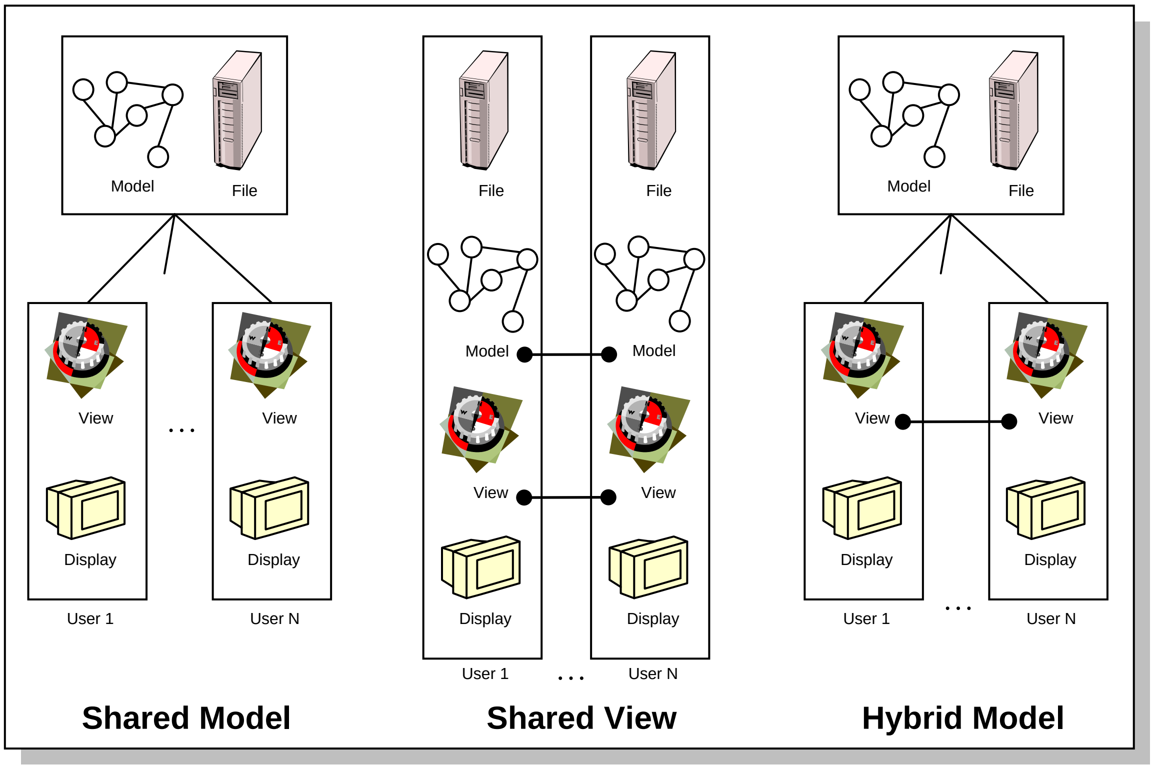


Figure 3: Distributions of Models, Views, and Displays

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) [116]. 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 [142]. To increase reuse of CES components, Geyer et al [35] 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 [79] propose such a Web Services-based architecture as shown in Figure 4.

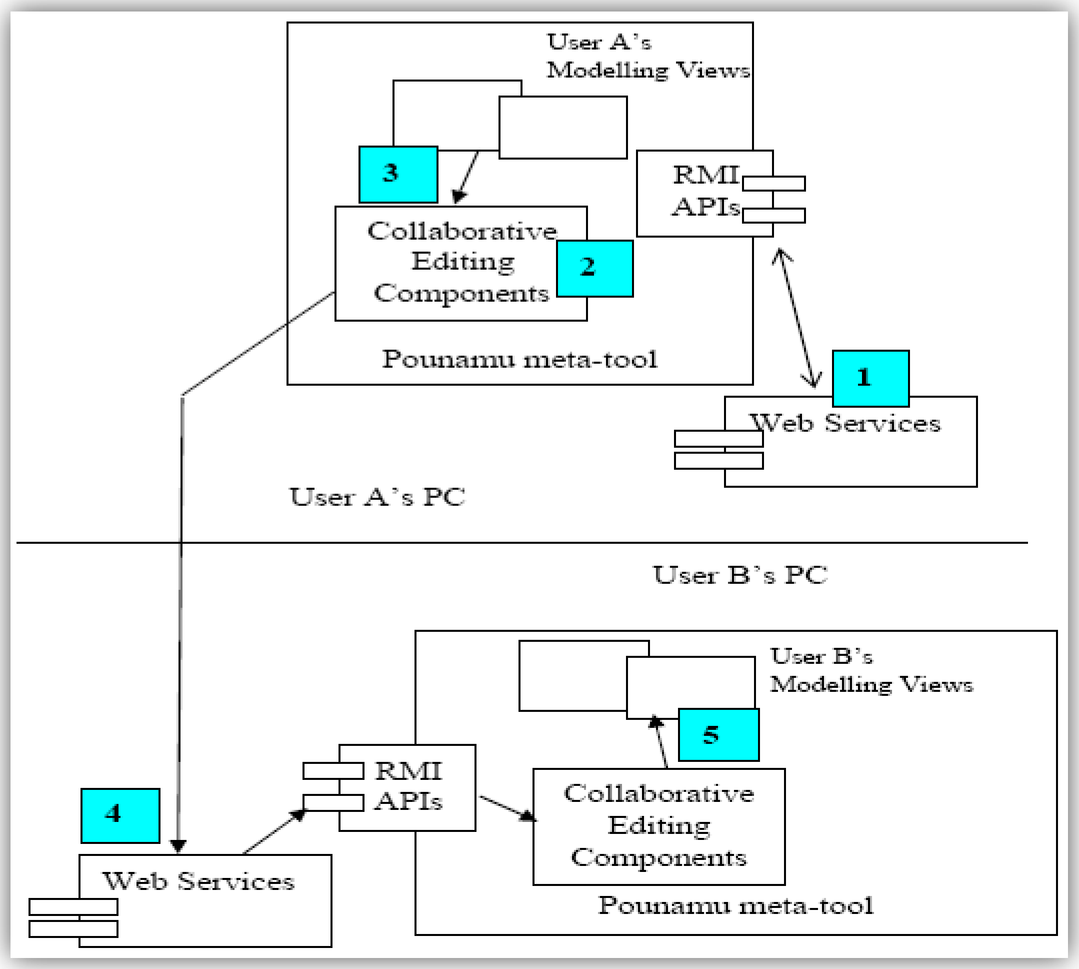


Figure 4: A Web Services-based Collaborative Editing Architecture

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 [64].

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 [73][74].

2.3. Concurrency Control Policies

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) [78]. 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 including pessimistic locking, optimistic locking, and sub-file level locking.

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 [19].

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 5 illustrates this.

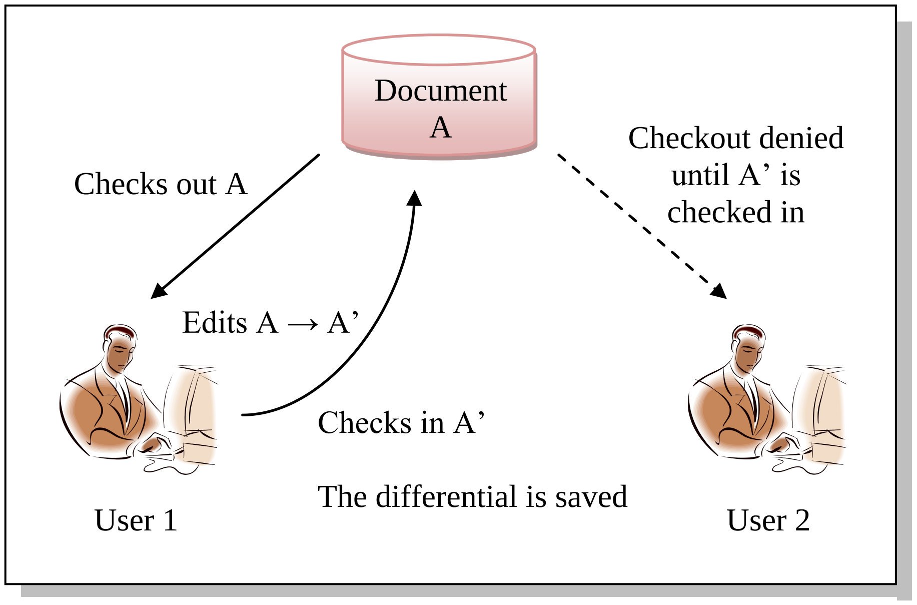

Document A
Checks out A
Checkout denied until A’ is checked in
Edits A → A’
Checks in A’
The differential is saved
User 1
User 2

Figure 5: 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.

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 [76][89], 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 [88].

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" [146]. 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) [88]. Figure 6 illustrates the optimistic coordination policy.

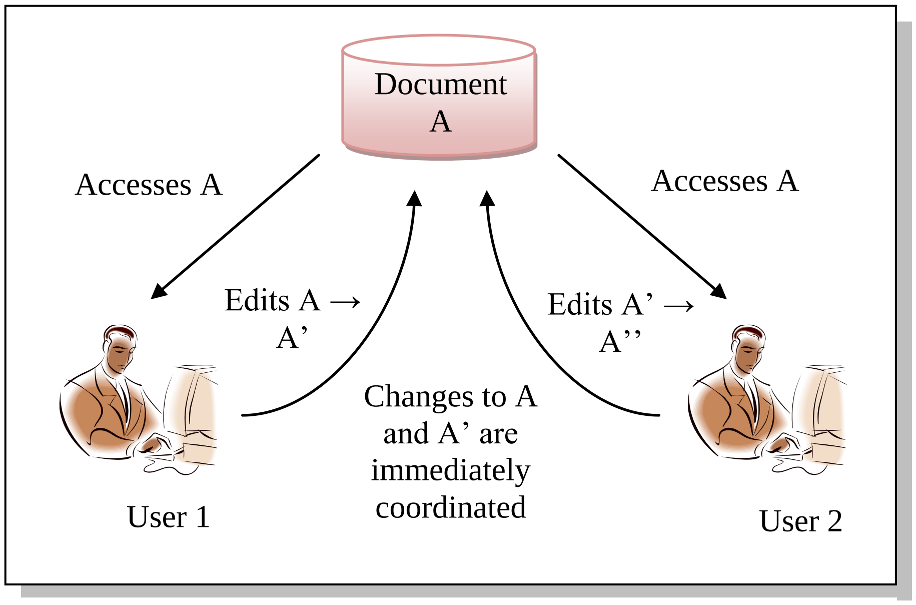


Figure 6: Optimistic Concurrency Control

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 [88]. 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) [132] as discussed later in this chapter can be used to ensure convergence of all copies of the document.

Many software configuration management (SCM) systems managed locks at the source file level within the repository. Examples include RCS, SCCS, VSS, CVS, and Subversion [Subversion] 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 [17][35]. Given that many edits by users in a software engineering project are localized and only change a small section of the document [97][98], fine-grain locking at a class/function/method level would be advantageous [16]. Some systems such as Coven [16] and COOP/Orm [75] 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 [71].

2.4. Convergence, Causality-preservation, and Intention-preservation

If mutual exclusion (locking) 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 [132] 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 $O_1 \ldots O_n$ where $O_1 \ldots O_n$ 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 [156] 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.

Currently, the CCI model is the standard by which to measure the correctness of a RTCES. The first two requirements (convergence and causality-preservation) have been achieved, but intention-preservation is still an open problem.

2.5. Operational Transformation

Operational transformation (OT) is a mechanism which seeks to achieve CCI. This section presents an overview of the approach and focuses on how causality-preservation and convergence are achieved via OT. We also present relevant concepts such as integration algorithms, transformation functions, and transformation properties.

Since a RTCES is a distributed system in which various sites are performing operations, either a centralized or a replicated state approach must be adopted to share the document being edited, and if a replicated approach is adopted, we must have some way to ensure CCI. When an operation occurs at a client's copy (site), four events occur [90]:

1. The operation is performed locally
2. The operation is broadcast to all other sites
3. The other sites receive the incoming operation
4. The other sites execute/replay the received operation

In a distributed system such as one adopting a replica based approach to RTCES, all operations have either causal relation (order) or concurrent relation with any other operation [65]. Vector timestamps can be used to establish correct causal ordering for causally related operations, but convergence is not so easily achieved among concurrent operations since the state of different sites changes when operations are performed and “replaying” an incoming remote operation may no longer be valid. OT is an approach to overcome this problem and achieve convergence based upon transforming incoming operations to the locally modified state. Figure 7 demonstrates the need to transform operations to ensure convergence among all sites within the collaboration. Two concurrent operations can be executed in a different order on two different sites’ copies. As a result, when an operation is received, the state of shared object at the receiving site may be changed relative to the state where the operation had been created. Thus, executing this operation in its original form on a receiving site does not ensure the copies converge.

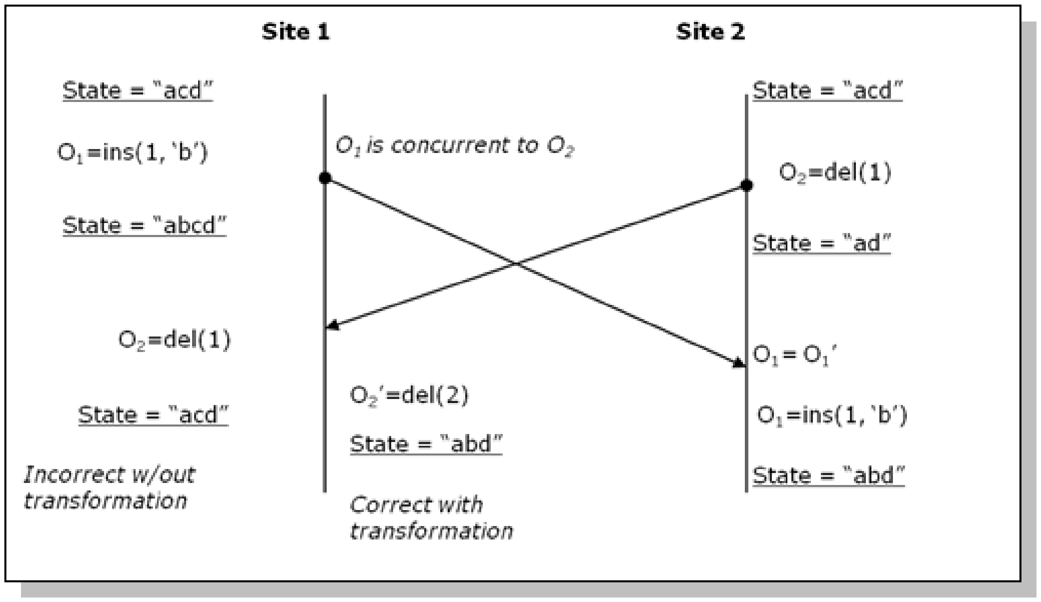


Figure 7: The Need for Operation Transformation – State Convergence

Causality preservation can be achieved by using a state vector that is generated when the operation is created [112][114] as follows. Assume that *n* is the number of sites, and sites are identified by integers 1 to *n*. Each site *n* maintains an *n*-tuple state vector SV_n. Initially $SV_n[i] = 0$, for $1 \leq i \leq n$. After site n executes an operation created at site i, the site timestamps its sequence number is increased by one such that $SV_n[i] = SV_n[i] + 1$. Further, let O be an operation generated at site k and let SV_o be the last timestamped state vector, which is transferred to other sites with O. We can say that O is causally ready to be executed at site l ($k \neq l$) with a state vector SV_l if the following conditions are true:

(1) $SV_o[k] := SV_l[k] + 1$

(2) $SV_o[i] \leq SV_l[i]$, for $1 \leq i \leq n$ and $i \neq k$.

To preserve causality, if an operation is not causally ready, then it must be delayed until both of the above conditions are true. Holding on to these non-ready operations necessitates a queue of waiting operations. Further, since operations may need to be undone at a future time, a history buffer must also be maintained.

Having discussed causality-preservation, we now turn our attention to convergence. To achieve convergence among all replicated states of the shared document OT defines two main components: the OT integration algorithm and the OT transformation function.

The OT integration algorithm is responsible for receiving the incoming operations from remote sites, distributing locally-generated operations to remote sites, and executing the operations on the site's document state. This component is essentially a distribution/communication and execution engine, and it invokes the transformation function as needed.

The OT transformation function makes up the bulk of active OT research. [29] defined a transformation function T to be a function that takes as parameters two concurrent operations, op_1 and op_2 where op_1 and op_2 must be defined on a same state S. The function T returns a new operation T(op_1, op_2) that is equivalent to op1 (has the same effects) but is defined on the state S', where S' is the state resulting when performing op_2 on state S.

[113] further refined the requirements of correctness of a RTCES in achieving CCI and demonstrated the sufficiency of TP_1 and TP_2, two transformation properties that must be met in order to preserve causality and achieve convergence in replicas within a RTCES. These properties are defined as:

TP_1 *For every pair of concurrent operations op1 and op2 defined on the same state, the transformation function T satisfies TP_1 property if and only if:*

$$op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

where $op_i \circ op_j$ denotes the sequence of operations containing op_i followed by op_j; and where $\equiv$ denotes equivalence of the two sequences of operations

TP_2 *For every three concurrent operations op_1, op_2 and op_3 defined on the same state, the transformation function T satisfies TP_2 property if and only if:*

$$T\big(op_3, op_1 \circ T(op_2, op_1)\big) = T\big(op_3, op_2 \circ T(op_1, op_2)\big)$$

TP_1 guarantees that the state generated at one site performing op_1 and then op_2 (after op_2 has been transformed relative to op_1's resultant state) will be the same as the state generated at another site performing op_2 and then op_1 (after op_1 has been transformed relative to op_2's resultant state). TP_2 guarantees equality of the states at different sites if op_3 is performed after an equivalent transformation; this property ensures that once two sites achieve equivalence (after TP_1), they will remain equivalent and cannot affect the resultant state after transformation on a future operation (op_3).

2.6. Discussion and Existing Systems

This section discusses an overview of the field of RTCES systems that have been developed since the field's inception in 1989 and some of the most current systems that support modern RTCES techniques. As shown in Table 2 [12], there have been numerous RTCES systems developed since 1989 when the field of RTCES research began. Most of these systems have been developed in the United States and half have been developed as a result of academic research.

Table 2: RTCES Developed by Year [Chen 2006]

RTCES	Year	RTCES	Year	RTCES	Year
GROVE	1989	GroupGraphics	1995	CoPowerPoint	2004
Aspects	1990	JointEmacs	1996	CoWord	2004
DistEdit	1990	LICRA	1997	DocSynch	2004
MultimETH	1990	REDUCE	1997	JotSpot Live	2004
CoMedia	1991	Col.AutoCad	1998	Tendax	2004
GroupIE	1991	Flex JAMM	1998	ACE	2005
MACE	1991	CoDiagram	2000	Gobby	2005
Ensemble	1992	GRACE	2000	InstaColl	2005
GroupDesign	1992	Presence-AR	2000	Java Studio	2005
GroupDraw	1992	CollabCAD	2001	Moonedit	2005
SEPIA	1992	ICT	2002	Scratchpad	2005
CoDraft	1993	Groove	2002	Writely	2005
ConversionBoard	1993	LeoN	2003	Sigsoft	2006
Iris	1993	LiveDrive	2003	SynchroEdit	2006
SASSE	1993	Subethaedit	2003	Syntext	2006
ShrEdit	1993	Chalks	2004	G.SpreadSheet	2006

There has been a steady increase in the number of RTCES developed since 1989's introduction of the GROVE system [29]. As Figure 9 shows, there is a consistent interest in the field of developing RTCES, and this interest is supported by our experiences when talking with colleagues about such collaborative tools – the question is almost universally raised: "Where can I get something like this to support my group in collaborating together?"

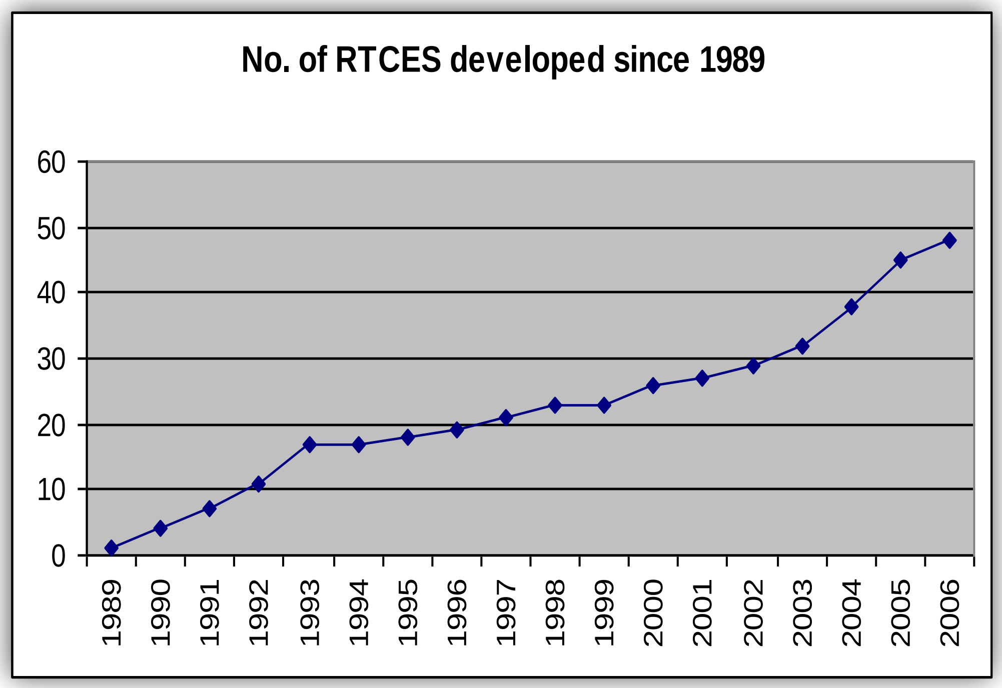


Figure 8: RTCES Development Growth: 1989-2006 [12]

While there was an initial surge of RTCES development in the early 1990s, the pace of development cooled from the mid 90s until its resurgence in the early 2000s – with the rise of Web-based systems.

Additionally, as shown in Figure 9, the document technologies supported in RTCES research since 1989 have been: text documents (no structure), rich text documents (with formatting such as fonts and graphics – this also includes presentation and spreadsheet document types), vector graphics documents, and structured documents (plain text documents with embedded bitmap graphics images). There has been a clear rise in the interest of rich text documents since 2001, and plain text document RTCES continue to be popular as this document type is most prevalent in consistency maintenance and OT algorithmic research.

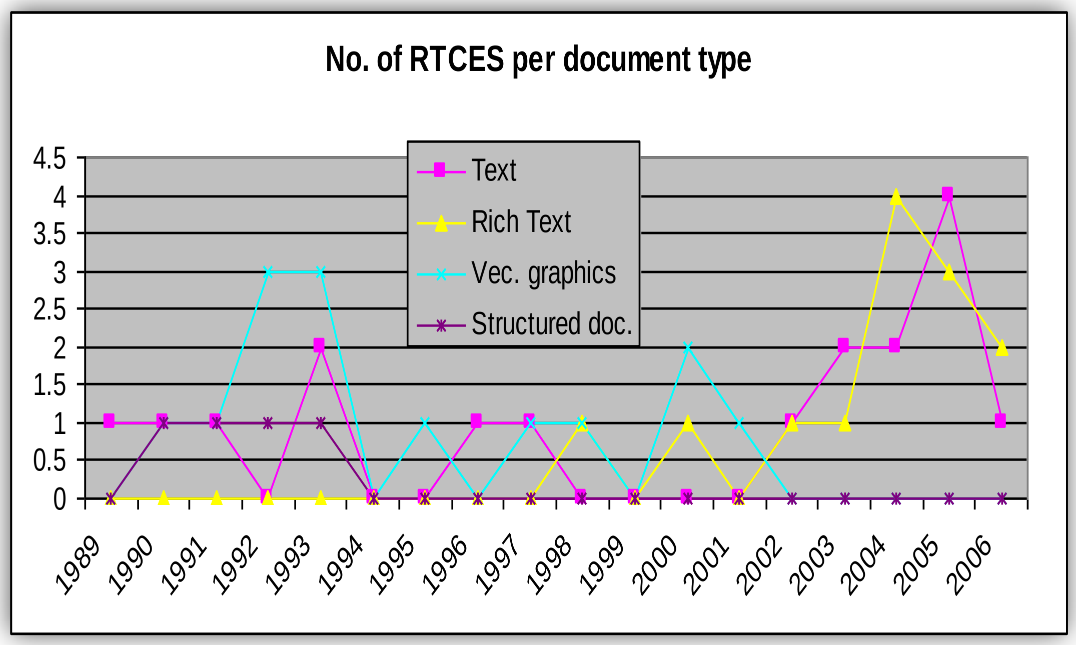


Figure 9: RTCES Document Types Supported: 1989-2006 [12]

According to [12], there have been only five Web-based RTCES developed in the past 3 years (2004-2006); these systems focus on supporting rich-text editing and utilize the new Asynchronous JavaScript and XML (AJAX) technology for their implementations.

While it is common that these Web-based RTCES are associated with Wikis given the collaborative nature of Wikis, it is important to note that Wiki technology utilizes version control and differentials that support asynchronous editing – allowing users to modify a shared document and "check in" their changes once their edits have been completed [26] and not in real-time.

A recent approach to ensuring consistency when utilizing a distributed, replicated groupware system is through a "mark and retrace" approach [45]. 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 to the efficient/inefficient marked states as shown in Figure 10. Mark and retrace is similar to the tombstone function approach of [90].

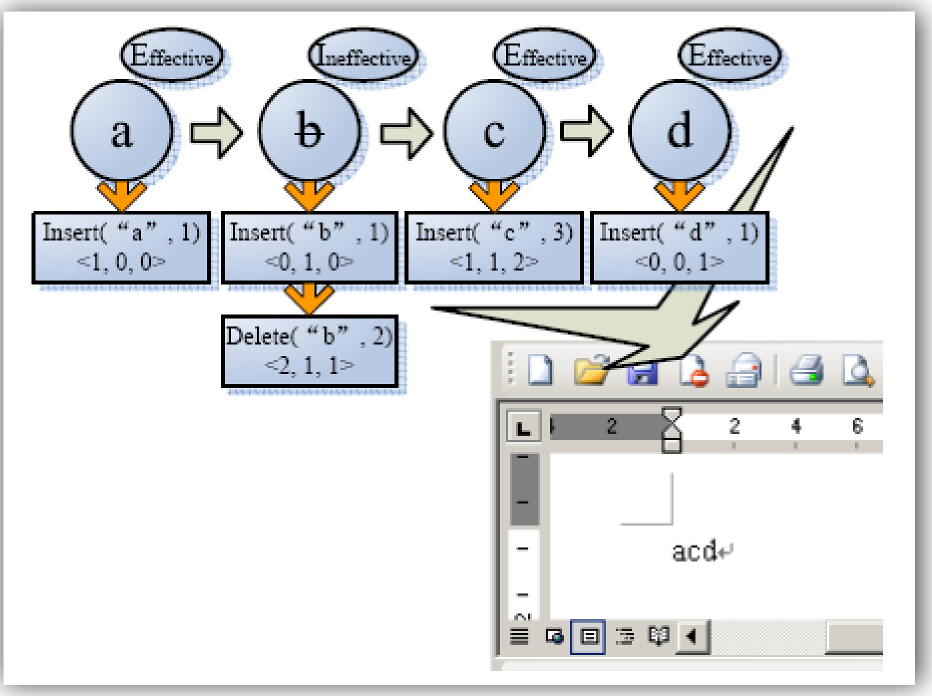

Effective
Ineffective
Effective
Effective
a
b
c
d
Insert("a" , 1) <1, 0, 0>
Insert("b" , 1) <0, 1, 0>
Insert("c" , 3) <1, 1, 2>
Insert("d" , 1) <0, 0, 1>
Delete("b" , 2) <2, 1, 1>
acd

Figure 10: Mark and Retrace

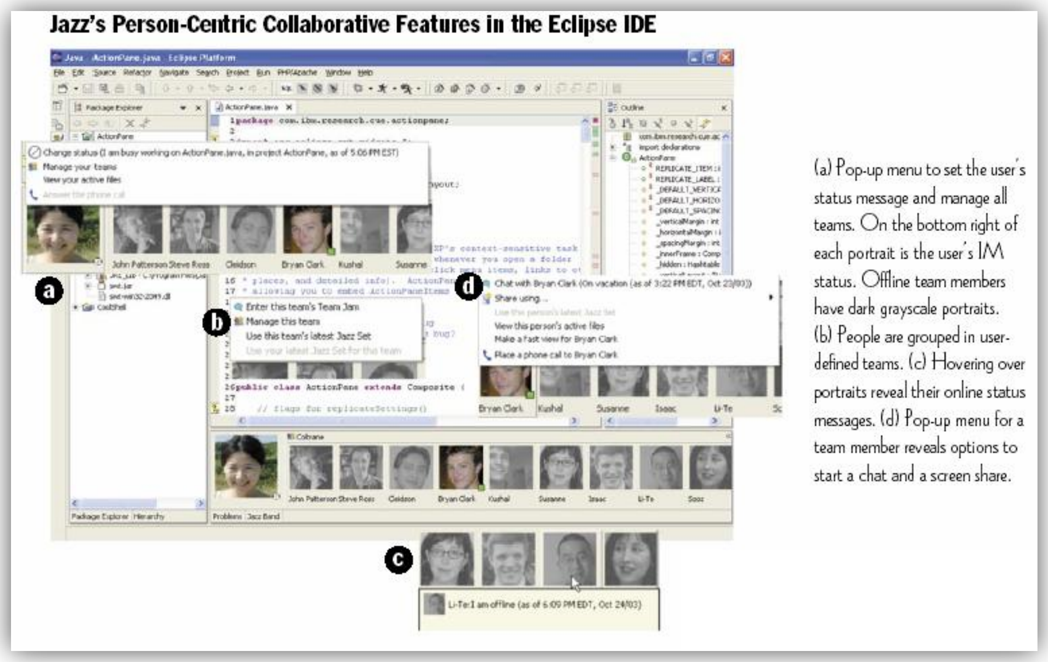
[20] presents work that allows for the extension of operational transformation techniques to be applied not only to linear text but also to tree-based XML/SGML/HTML documents. The SGML notion of a "grove" of data is utilized and the CCI model is adhered to [134]. Others [132] have applied the techniques 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 [140] within the domain of graphic editing systems, and the challenge of this approach remains achieving semantic consistency rather than syntactic consistency [156].

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 [63].

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 [121] and [36] 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 [27] and Visual Studio [149] 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) as shown in Figure 11 [13].


Jazz's Person-Centric Collaborative Features in 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 reveal their online status messages. (d) Pop-up menu for a team member reveals options to start a chat and a screen share.

Change status (I am busy working on ActionPane.java, in project ActionPane, as of 5:06 PM EST)
Manage your teams
View your active files
Enter this team's Team Jam
Manage this team
Use this team's latest Jazz Set
Chat with Bryan Clark (On vacation (as of 3:22 PM EDT, Oct 23/03))
Share using...
View this person's active files
Make a fast view for Bryan Clark
Place a phone call to Bryan Clark
Li-Te: I am offline (as of 6:09 PM EDT, Oct 24/03)

Figure 11: Integrating Collaboration into IDEs (Jazz)

Existing applications such as CoWord and CoPowerPoint [158], and CoStarOffice [122] 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.

When attempting to achieve multi-user collaboration, systems have taken existing single-user applications and modified them such that they can serve as a multi-user editing system. DistEdit [62] is one such system that integrates additional multi-user capabilities into an existing single-user editing system. Others include CoWord and CoPowerPoint [158], CoStarOffice [122], and CoOpenOffice [136].

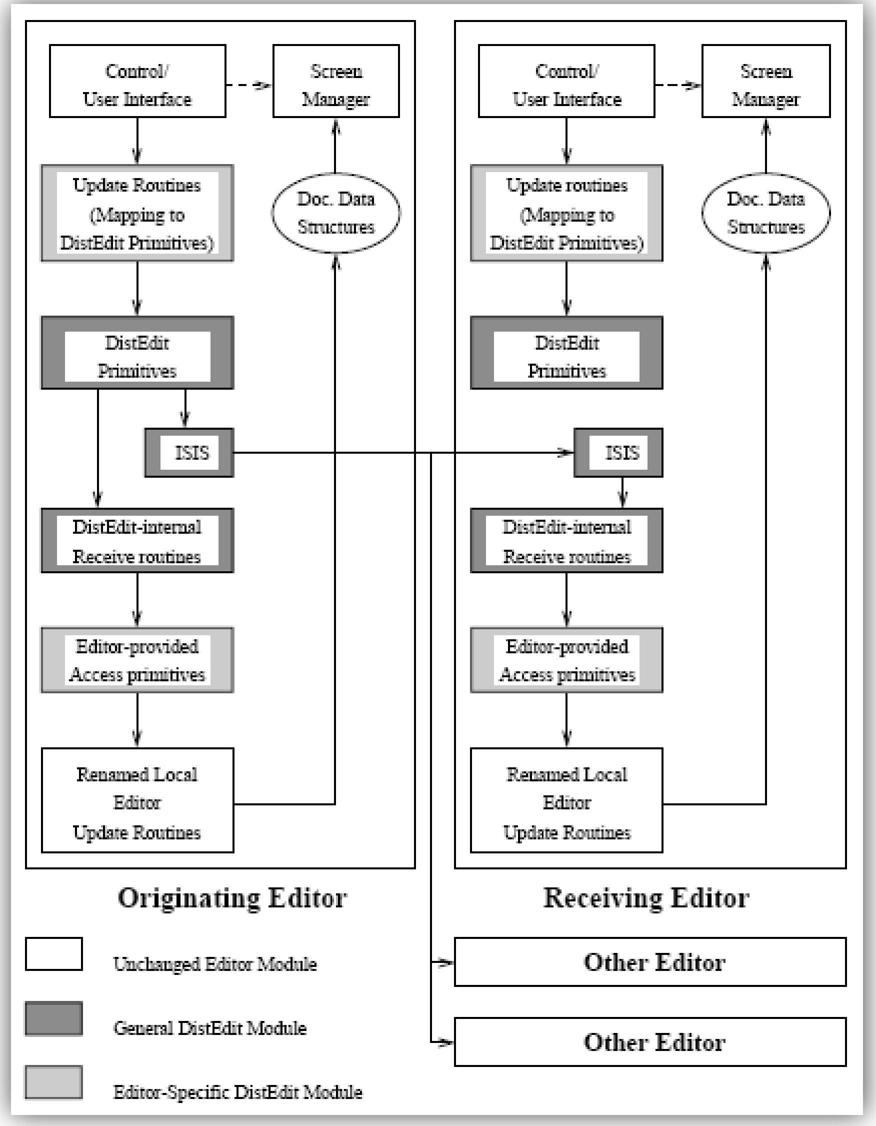
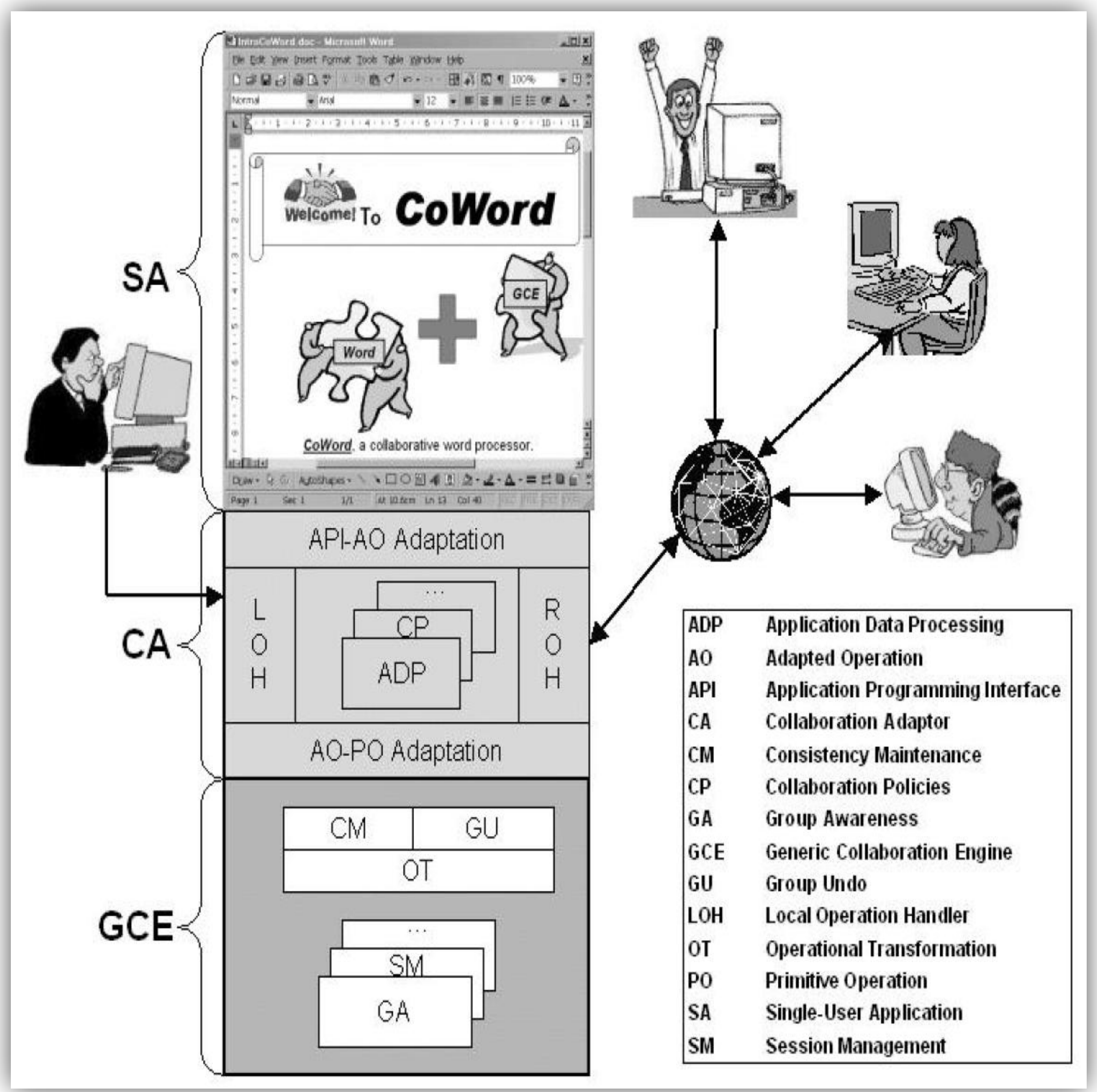

Control/ User Interface
Screen Manager
Update Routines (Mapping to DistEdit Primitives)
Doc. Data Structures
DistEdit Primitives
ISIS
DistEdit-internal Receive routines
Editor-provided Access primitives
Renamed Local Editor Update Routines
Control/ User Interface
Screen Manager
Update routines (Mapping to DistEdit Primitives)
Doc. Data Structures
DistEdit Primitives
ISIS
DistEdit-internal Receive routines
Editor-provided Access primitives
Renamed Local Editor Update Routines
Originating Editor
Receiving Editor
Unchanged Editor Module
General DistEdit Module
Editor-Specific DistEdit Module
Other Editor
Other Editor

Figure 12: The DistEdit Approach of Adding Collaboration to Existing Applications

Notice in Figure 12 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 [62].

The CoWord and CoPowerPoint 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 [158], as shown in Figure 13.



ADP	Application Data Processing
AO	Adapted Operation
API	Application Programming Interface
CA	Collaboration Adaptor
CM	Consistency Maintenance
CP	Collaboration Policies
GA	Group Awareness
GCE	Generic Collaboration Engine
GU	Group Undo
LOH	Local Operation Handler
OT	Operational Transformation
PO	Primitive Operation
SA	Single-User Application
SM	Session Management

Figure 13: The CoWord Approach to Adapting Single User Applications to RTCES

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 [68]. They 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 [22]. In this case, as shown in Figure 14, 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.

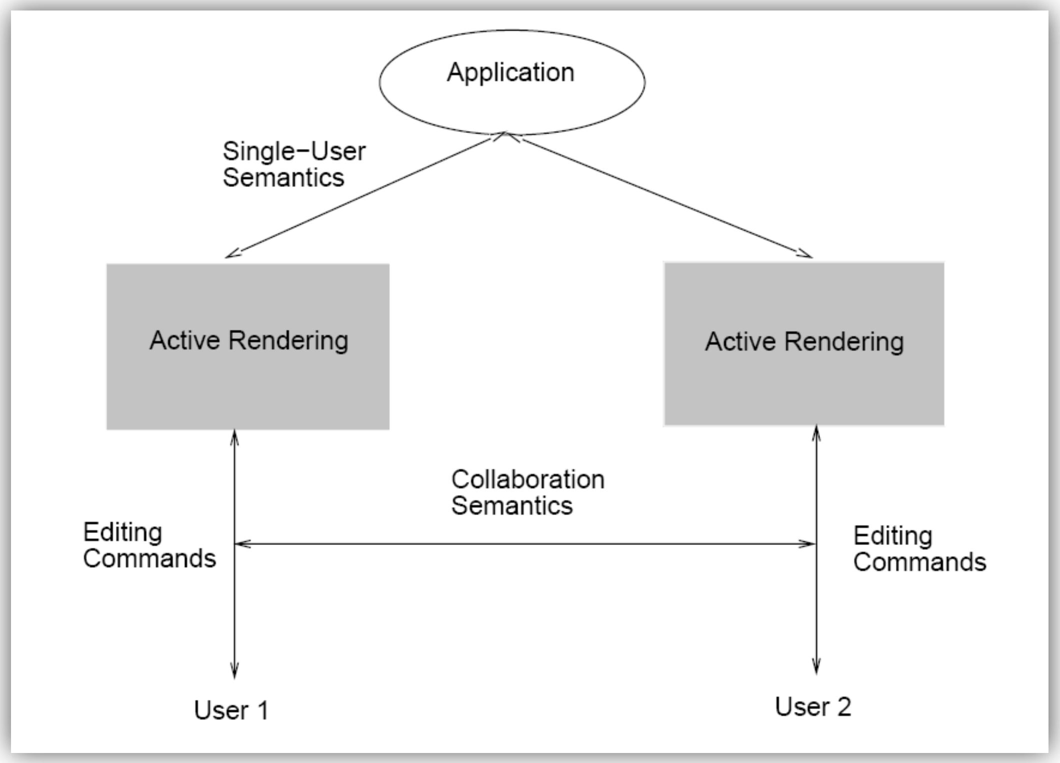
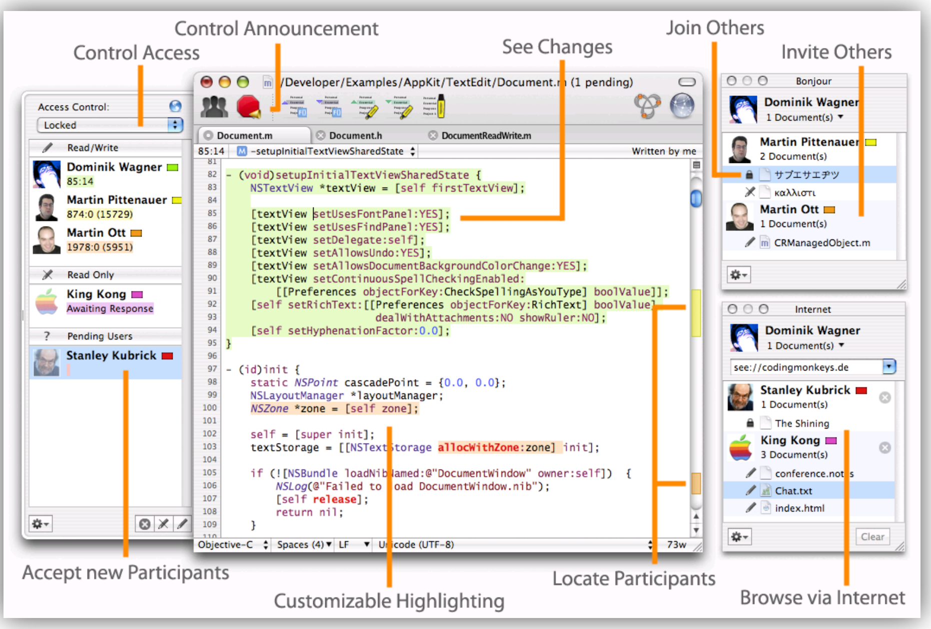


Figure 14: Generalized Collaborative Architecture

Another current, viable RTCES system is the SubEthaEdit system as shown in Figure 15. This system runs on the Mac OS and features many usability and awareness features of other CES editors. SubEthaEdit allows users to connect to a central server and collaborate in real time. This system shows presence information about each user (labeled as Locate Participants and Control Access in Figure 15) [127]. This system, like others such as Google Docs [37], Groove [42], and SharePoint [81], does not ensure true CCI as the level of coordination and state management uses some form of asynchronous (lock based) coordination and require explicit "check in" of the shared document to update remote states; thus some form of merge reconciliation is required to synchronize states if two collaborators change the same content.



/Developer/Examples/AppKit/TextEdit/Document.m (1 pending)

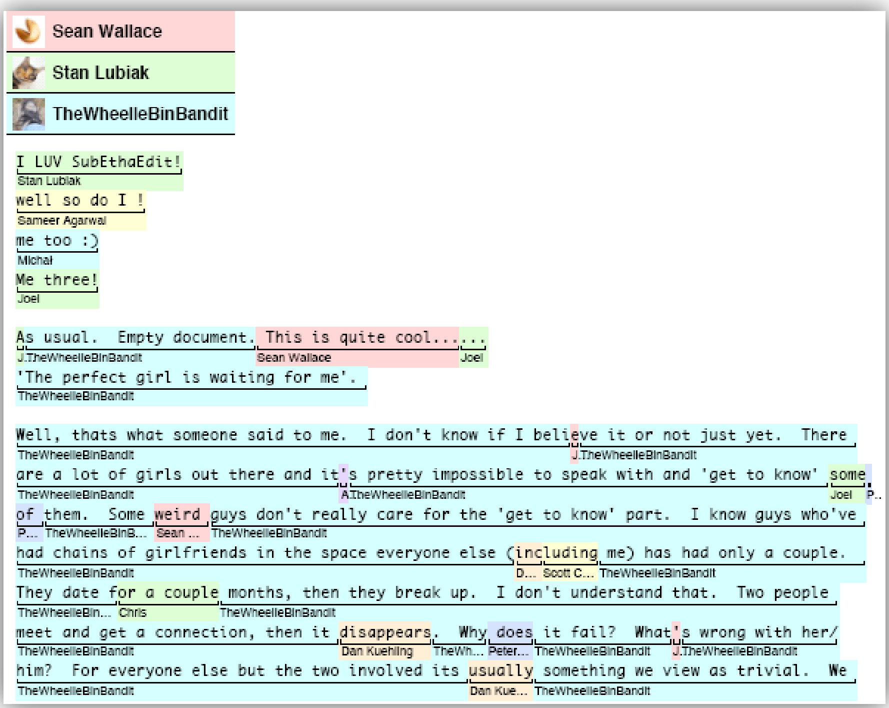
Document.m | Document.h | DocumentReadWrite.m

85:14 -setupInitialTextViewSharedState | Written by me

```
- (void)setupInitialTextViewSharedState {
    NSTextView *textView = [self firstTextView];

    [textView setUsesFontPanel:YES];
    [textView setUsesFindPanel:YES];
    [textView setDelegate:self];
    [textView setAllowsUndo:YES];
    [textView setAllowsDocumentBackgroundColorChange:YES];
    [textView setContinuousSpellCheckingEnabled:
        [[Preferences objectForKey:CheckSpellingAsYouType] boolValue]];
    [self setRichText:[[Preferences objectForKey:RichText] boolValue]
                    dealWithAttachments:NO showRuler:NO];
    [self setHyphenationFactor:0.0];
}

- (id)init {
    static NSPoint cascadePoint = {0.0, 0.0};
    NSLayoutManager *layoutManager;
    NSZone *zone = [self zone];

    self = [super init];
    textStorage = [[NSTextStorage allocWithZone:zone] init];

    if (![NSBundle loadNibNamed:@"DocumentWindow" owner:self])  {
        NSLog(@"Failed to load DocumentWindow.nib");
        [self release];
        return nil;
    }
```

Objective-C | Spaces (4) | LF | Unicode (UTF-8) | 73w

Figure 15: Collaboration via in SubEthaEdit



```
I LUV SubEthaEdit!
well so do I !
me too :)
Me three!

As usual.  Empty document. This is quite cool......
'The perfect girl is waiting for me'.

Well, thats what someone said to me.  I don't know if I believe it or not just yet.  There
are a lot of girls out there and it's pretty impossible to speak with and 'get to know' some
of them.  Some weird guys don't really care for the 'get to know' part.  I know guys who've
had chains of girlfriends in the space everyone else (including me) has had only a couple.
They date for a couple months, then they break up.  I don't understand that.  Two people
meet and get a connection, then it disappears.  Why does it fail?  What's wrong with her/
him?  For everyone else but the two involved its usually something we view as trivial.  We
```

Figure 16: Viewing Changes Made By Users – a SubEthaEdit Report

Many CES have adopted similar visualizations to SubEthaEdit's change/modification log [94] to assist users in tracking how changes are made. While useful, these change logs and reports are for post collaboration used (i.e., they show the changes some after they occur).

Historically, OT research has sought to achieve both TP_1 and TP_2, but TP_2 has been elusive/difficult to achieve until recently when it was solved via the TTF [90]. While TP_1 and TP_2 are necessary and sufficient to achieve convergence and causality preservation, intention preservation is still an active research area in the field of RTCES.

Unfortunately, the current OT approaches do not scale for a large number of operations and a large number of users. Since all operations must be broadcast to all users (except for the originating user of the operation), this approach is costly with respect to communication. Additionally, we assume that the number of operations performed in the collaboration is relative to the number of users within the collaboration, thus the total number of operations that must be sent across the network is relative to O(n^2) where *n* is the number of users within the collaboration.

Further, OT is costly with respect to the total memory required in storing the history buffers among all clients. The history buffers at each user's site must be large enough to accommodate the arrival of a highly-delayed operation arriving at a user's site such that this "late arriving" operation can be correctly applied in causal order. Thus OT approaches assume a highly-connected, synchronous editing environment where messages are not significantly delayed (or lost) when in transit across the network. If significant delay occurs on the network or if operations are not sent quickly to all users' sites, then the history buffers may grow significantly large, and consistency will not be achieved… and system performance and the collaboration will decay rapidly. As a result, our research goal is to improve RTCES beyond current OT-based systems.

CHAPTER 3

AN OPEN SYSTEMS APPROACH TO RTCES

Given that many users have their own favorite editing software on the client side and there are many existing server-side repositories that contain documents, it is advantageous to create a system that can support the use of these existing technologies. Users are often hesitant to adopt new collaborative tools that don’t have the same feature set or familiarity of their current tools [64]; as a result, we strive to provide a means by which a heterogeneous collection of existing client and server side technologies may be interconnected within a Collaborative Editing System such that user can retain the use of their favored tools and connect to the plethora of existing server repositories.

We note that many feature-rich editing systems such as OpenOffice, Microsoft Office, and various integrated development environments (IDEs) such as Borland’s JBuilder, Microsoft Visual Studio, and Sun’s NetBeans have a large existing user base. Likewise, many configuration management systems (CMS) and document repositories such as RCS, VSS, and CVS are currently implemented worldwide and store a large collection of documents.

Our work brings these existing client and server technologies together in an open-systems architecture that allows users to retain their favored tools and leverage on existing document servers through the use of Web-services. [4][79][160] discuss Web-service-based approaches similar to our system but their systems are coupled to specific tools (IDEs) whereas our approach allows for the integration of any IDE, CMS, and communication tools; consequently, our architecture is more flexible.

Central to our motivation is the need to allow users to synchronously and asynchronously edit documents. When accessing documents synchronously, users typically are made aware of other users in the system [46]. Our architecture handles the negotiation of awareness and concurrent access transparently to the users such that they can focus on the work at hand without being hindered by check-in and check-out level minutiae.

Figure 17 demonstrates the approach of our architecture in allowing varied technologies to connect and work together in a CES. On the client side, different document editors such as Microsoft Word, notepad, Open Office, etc. can be used by different clients within the CES, yet each has a listener entity that translates local changes to the shared document to be replayed by other clients on their chosen applications. Similarly, the Web services API provides a consistent interface by which clients may request files for check-in and check-out; the specific server technology remains hidden, so it does not matter if CVS, VSS, or another CMS technology is adopted. To achieve heterogeneity among the clients, it is necessary that a client application listener be employed that can detect changes to the document, translate these changed into an application-independent format, and then send these changes to other clients via the server-side coordination Web service.

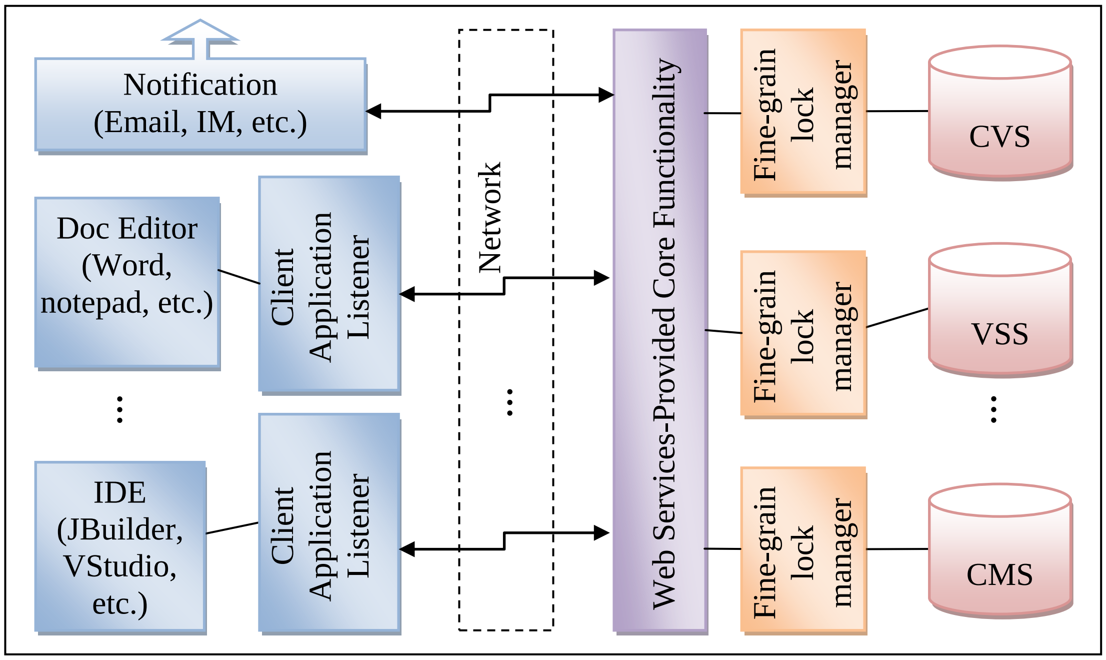

Notification (Email, IM, etc.)
Doc Editor (Word, notepad, etc.)
IDE (JBuilder, VStudio, etc.)
Client Application Listener
Client Application Listener
Network
Web Services-Provided Core Funcionality
Fine-grain lock manager
Fine-grain lock manager
Fine-grain lock manager
CVS
VSS
CMS

Figure 17: Heterogeneous Architecture

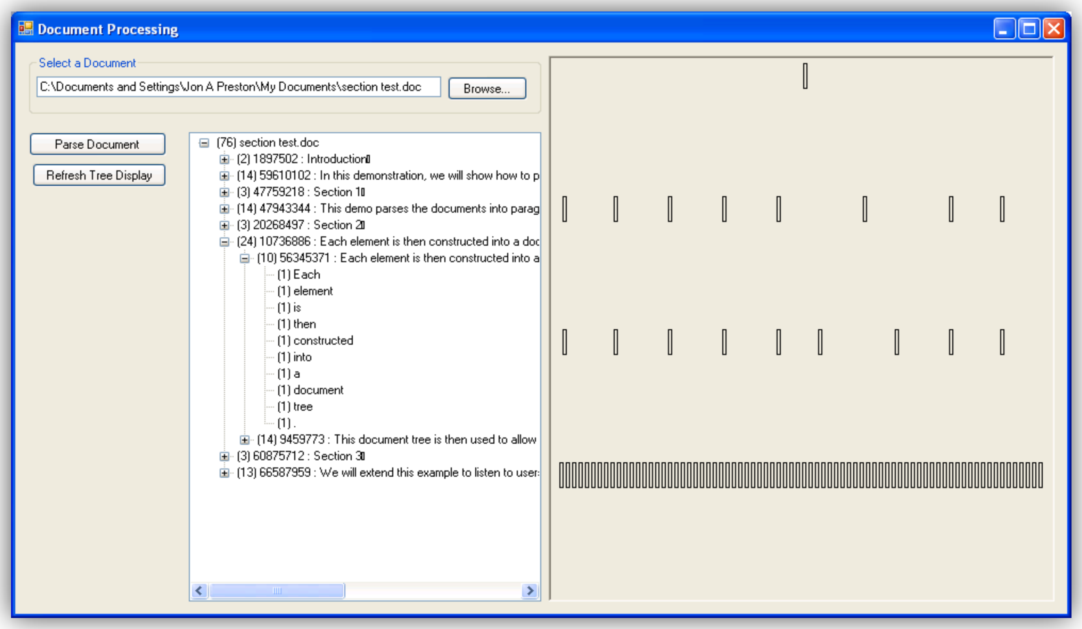
The remainder of this chapter is organized as follows: we first present how various client technologies may coordinate in a RTCES, and then present how various server technologies may coordinate in a RTCES. We then discuss how events on the clients must be translated from one client technology to another if a heterogeneous set of clients technologies is to be supported. Next, we present the overall heterogeneous architecture that combines the client and server technologies via Web services and discuss the event flow within the architecture. We present validation of our architectural approach via simulation and prototype implementation. Finally, we conclude with a discussion and summary.

3.1. Supporting Various Client Technologies

In order to support an existing client editing tool, two approaches are applicable: either transparency or aware collaboration technology. As previously discussed, it is difficult to support transparency within collaboration because all events (at the OS level) must be captured and the collaborative system has no knowledge of what these events mean within the context of the application; rather, the system is just capturing, broadcasting, and replaying system-level mouse click and key press type events. As a result, we focus on aware collaboration technology in which hooks may be connected to existing client editing applications and attain more knowledgeable events such as insertions, deletions, etc.

To create such an aware collaborative hook, it is necessary to enumerate the features (edit events) that are to be shared and supported within the collaboration. Triggers that are fired when such events are raised must be written such that when these edit events occur, the client hook may intercept the edit event and act accordingly. The response could simply pass the edit event to the existing client editor, but additionally, it could sent messages to a server to request write access to the section of the document the user is attempting to write to or broadcast the edit event to other users within the collaboration. We do not prescribe what must occur within these triggers, but do demonstrate the necessity of the triggers in supporting the client technologies.

To demonstrate that we can implement the client hooks necessary for an open-architecture system that supports any type of client, we developed a program that parses Microsoft Word documents into our hierarchical document tree data structure; in this case, we parse the document into paragraphs, sentences, and words using Word's internal document object model (DOM). This program's functionality is demonstrated in Figure 18 where a Microsoft Word document has been parsed into the tree view displayed in the middle of the application and is shown graphically in the right of the application. In this example, the atomic level of parsing is the word, so words appear as leaves within the document tree, and the non-leaf, structural nodes represent the assimilation of the words into sentences, then sentences into paragraphs, and paragraphs into the entire document (at the root).



C:\Documents and Settings\Jon A Preston\My Documents\section test.doc

```
(76) section test.doc
    (2) 1897502 : Introduction
    (14) 59610102 : In this demonstration, we will show how to p
    (3) 47759218 : Section 1
    (14) 47943344 : This demo parses the documents into parag
    (3) 20268497 : Section 2
    (24) 10736886 : Each element is then constructed into a doc
        (10) 56345371 : Each element is then constructed into a
            (1) Each
            (1) element
            (1) is
            (1) then
            (1) constructed
            (1) into
            (1) a
            (1) document
            (1) tree
            (1) .
        (14) 9459773 : This document tree is then used to allow
    (3) 60875712 : Section 3
    (13) 66587959 : We will extend this example to listen to user:
```

Figure 18: Parsing a Microsoft Word Document into a Document Tree

While not a complete solution, this brief prototype does show that the document object model (DOM) within Microsoft Office products such as Microsoft Word can be parsed into its semantic structure. Of particular interest in this prototype is that such semantic structure can be gleaned from even a closed-system and the proprietary format of Microsoft.

3.2. Supporting Various Server Technologies

Given that many different server repositories are currently in use and consist of a large set of documents, it would be advantageous to be able to connect to these existing technologies without the need to adopt a new, specialized system specific in supporting RTCES. Additionally, since there is a variety of technologies current in use, any architecture to support RTCES should take into account that it must support a heterogeneous collection of server technologies. To support these various, existing server technologies, we propose adding an architectural layer on top of the existing server repository that insulates the particular implementation from the client users. In this way, a standard API may be defined that all clients may make use of – enabling check in, check out, optimistic concurrency, pessimistic concurrency, subscription and notification upon changes to documents within the repository, and other such features. Supporting multiple repositories has been proposed and implemented by [85] and others, and our approach also utilizes a Web service interface by which clients may connect – realizing the open-systems approach of our proposed architecture. This process or layering additional API and features atop the existing server technologies is shown in Figure 19.

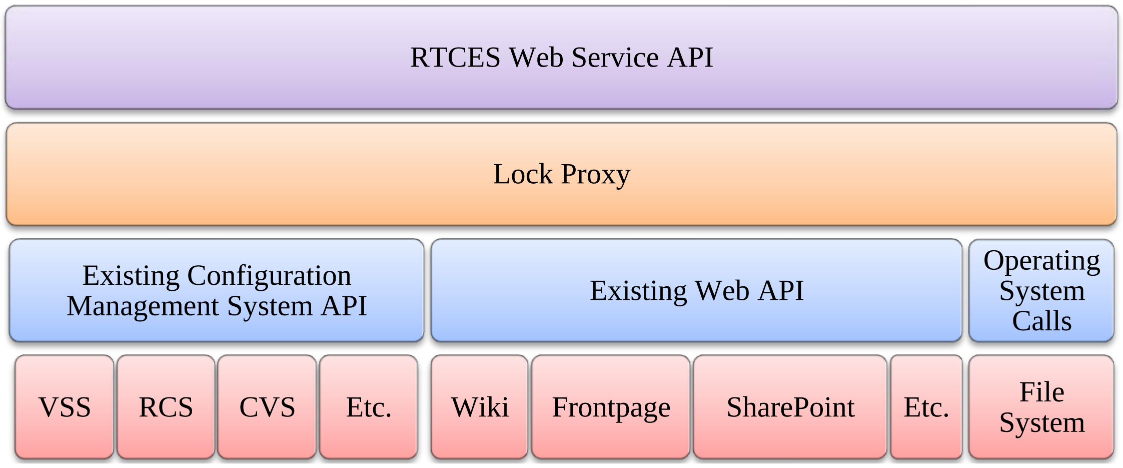


Figure 19: Layering the Lock Proxy and Web Service API atop Existing CMS

In Figure 19, the existing CMS, Web-based, and OS file system appear at the lowest level (pink) and have publicly accessible APIs at the next level up (blue). We add a lock proxy one layer higher (orange) that implements dynamic, hierarchical locking to increase concurrent access and manage computation and communication costs (see Chapter 4). To ensure that these services are accessible regardless of the client technology being employed, we adopt a Web service front-end (show in purple) atop the lock proxy. The lock proxy must connect to each server technology and map a subset of the RTCES events that the server previously provided (document check in, document check out, etc.) to the server API comments. For example, a document check in command issued within the RTCES would have to map to the CVS "ci" command if the server technology managing the shared document was CVS. On the other hand, some RTCES events would not pass down the layers to the existing server technology; client cursor movement and individual lock request and release commands would be handled in at the lock proxy layer without need to pass them further down. Thus the number of events to map to the existing server technology is limited and tractable.

The result of our approach is a server-side solution that insulates/hides the implementation details of the particular server technology employed so that any number of client technologies may make use of the documents in the servers' repositories. Additionally, the added capabilities of the dynamic locking are added atop the server without having to have access to the internal implementations of the server technologies (i.e., no code-level access or recompilation is required to add the new capabilities).

3.3. Translation of Events

While supporting an "aware" set of homogeneous client tools proves challenging because triggers must be written for each edit event that we would like to capture and respond to, the difficulty in supporting heterogeneous clients is even greater. In a heterogeneous environment, the client hooks must be written for each client technology to be supported in the system, but additionally, a mapping from each client technology edit event to each other client technology event must be written. It is no longer sufficient to simply transmit the operations occurring locally to remote clients because the remote client may not employ the same editing technology as the local user.

For example, if an event X is triggered at client C_1 using technology T_1, this event X must be mapped to X_i such that X_i achieves the same intention (results in the same document state) on T_i that X achieved on T_1 when replayed for each technology T_i (i.e., $\forall\ T_i \in T$, where T = the set of heterogeneous technologies employed by the users in the collaboration). This is undesirably complex and $O(T^2)$ as depicted in Figure 20.

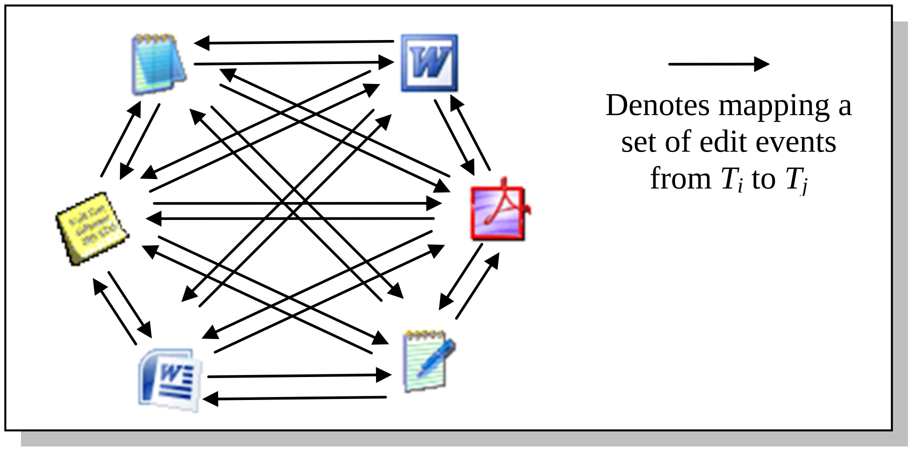


Figure 20: Mapping Client Events Directly to Each Other

A better approach would be to receive an edit event from the client application and translate this event into a “meta” language representing the intention of the event on the shared document within the RTCES. From this meta language, the event could be translated into a specific command for a target client technology. This is more efficient and only requires O(*T*) triggers to be written; further, it is more scalable in that when a new client technology is to be supported, none of the other client hooks need to be aware of the new technology – they still translate into the meta language and from there, the meta language translation tool can translate the event into the new client technology format. This process is depicted in Figure 21.

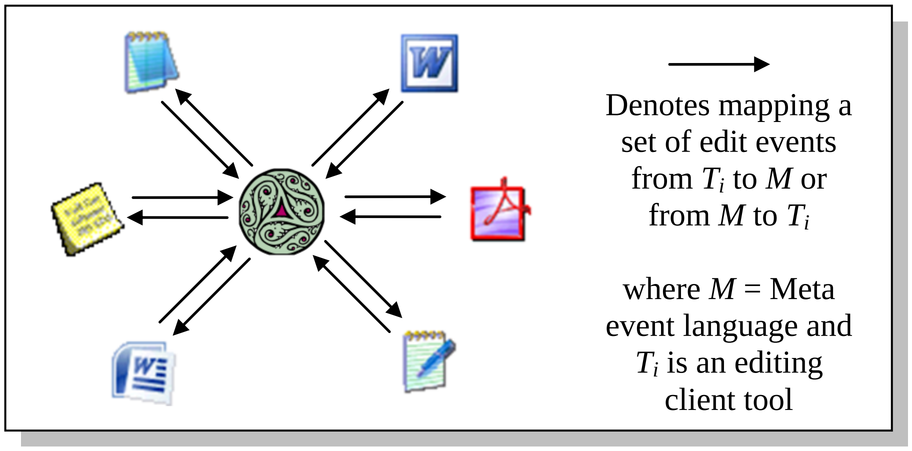

Denotes mapping a set of edit events from T_i to M or from M to T_i
where M = Meta event language and T_i is an editing client tool

Figure 21: Mapping Client Events to an Intermediate Meta Event Language

The downside of taking this centralized meta-language approach to translating the events from technology to technology is that it does require an additional computational step when compared to direct technology-to-technology translation because of the intermediate meta format. If it proves too costly/slow to move between the intermediate meta format, it is possible to implement direct translations for the most common client technologies and have the translation bypass the intermediate in these time-critical (and perhaps more common) situations.

3.4. Heterogeneous Architecture

Having discussed how various heterogeneous client and server components may be supported within a RTCES, we now integrate them into a proposed architecture.

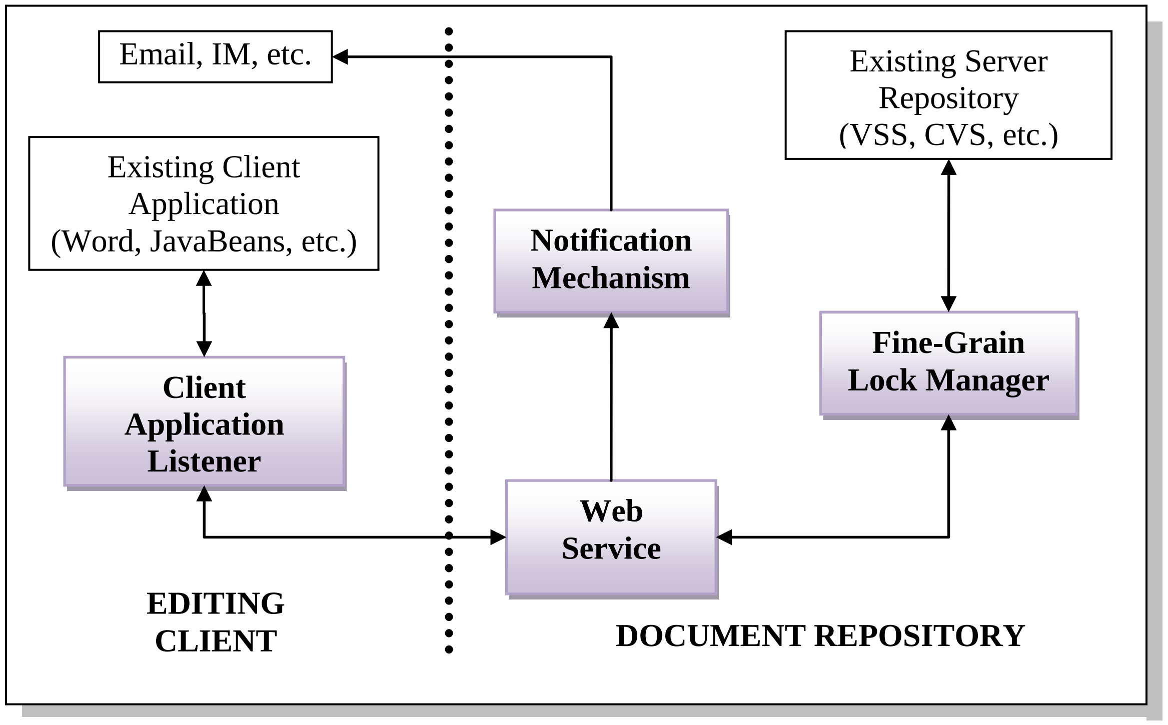

Email, IM, etc.
Existing Server Repository (VSS, CVS, etc.)
Existing Client Application (Word, JavaBeans, etc.)
Notification Mechanism
Fine-Grain Lock Manager
Client Application Listener
Web Service
EDITING CLIENT
DOCUMENT REPOSITORY

Figure 22: Architecture Components

The *Client Application Listener* component connects to existing client applications such as MS Word and IDEs like JavaBeans so that users may use their 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.

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. 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 lock manager intercepts messages from the network and processes them accordingly. The lock manager 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 lock manager 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 lock manager if there are no other clients currently accessing the subsection being requested.

The result of this additional lock manager 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. Furthermore, if the existing configuration management system does not support replication of the files among multiple clients, then our approach adds this capability by checking the files out and in via lock manager; thus the existing CMS is only aware of one user (the lock manager) and the lock manager is then responsible for coordination among the clients.

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.

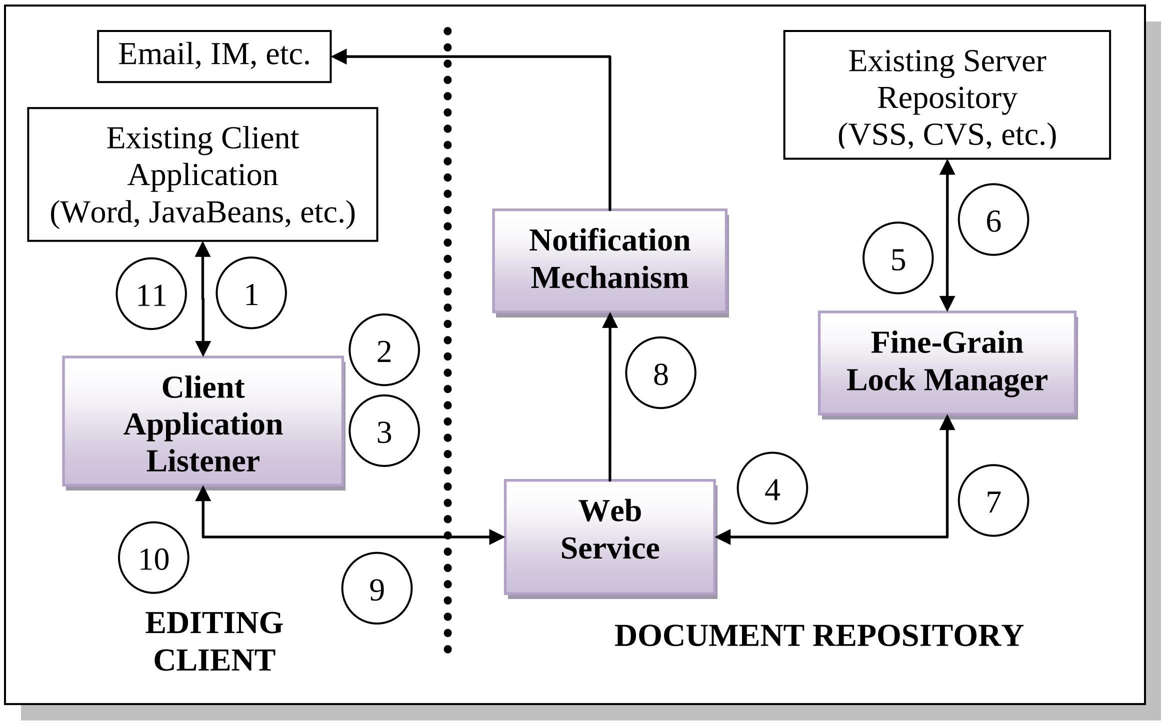

Email, IM, etc.
Existing Client Application (Word, JavaBeans, etc.)
Client Application Listener
Notification Mechanism
Web Service
Existing Server Repository (VSS, CVS, etc.)
Fine-Grain Lock Manager
1
2
3
4
5
6
7
8
9
10
11
EDITING CLIENT
DOCUMENT REPOSITORY

Figure 23: Events in the Architecture

The following 11 events are illustrated in Figure 23. 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.

The aforementioned architectural components enable heterogeneous client and heterogeneous server technologies to interact within a RTCES – allowing clients to use their preferred tools and enabling RTCES to work with legacy server repositories. The Web service approach acts to insulate the specific server implementations from the clients, and the client hooks facilitate interoperability among varied client technologies; further, the architecture supports subscription-based and asynchronous notification mechanisms for users that are interested on per-event awareness of changes within the RTCES.

3.5. Validation

To validate our approach and determine the communication costs associated with such a distributed architecture, we implemented two studies. The first verified what we believed intuitively that locking documents at a sub-file level would increase concurrent access to the shared documents via a lock proxy. The second verified that communication costs are reasonable to support such an open architectural approach. First, we discuss the background of the DEVS formalism used in the first simulation (and the simulation later described in Section 5.2) in modeling the components within our architecture; we then discuss our simulation in validating how adding a lock proxy improves concurrent access to files within legacy CMS; finally, we present our work in measuring communication costs associated with various events within the architecture.

3.5.1. Introduction to DEVS

The Discrete Event System Specification (DEVS) is a formalism for discrete event systems [161] and is the basis for the DEVS Java [162] simulation package used for validating our open systems architectural approach to RTCES; we also use the DEVS Java package for later simulations in this research (as described in Section 5.2). Formally, DEVS is a tuple:

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

X is the set of input values

S is the set of states

Y is the set of output values

$\delta_{int} : S \rightarrow S$ is the internal transition function

$\delta_{ext} : Q \times X \rightarrow S$ is the external transition function,

where $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total state set and e is the time elapsed since the last transition

$\lambda : S \rightarrow Y$ is the output function

$ta : S \rightarrow R+$ is the set of positive reals including 0 and ∞

Consequently, we can use DEVS to create models that reflect state transitions based upon internal (based upon internal timings) and external (based upon receiving inputs/messages from other entities) events. Additionally, these models can receive and generate events, which is easily mapped to an object-oriented implementation.

One of the fundamental classes of DEVS modeling is the atomic model/class which is defined as *M* above. In *M* we have states that the object can exist in, and based upon timing events, the model can transition to other states; additionally, the model can transition to other states based upon receiving an external event/input. When a model transitions state, it is able to generate an message to be sent as output of the model. The inputs are received by an atomic model via input ports where $X = \{(p, v) \mid p \in InPorts, v \in X_p\}$, and the outputs are sent out by an atomic model via output ports where $\text{Y} = \{\{(p, v) \mid p \in OutPorts, v \in Y_p\}$. The atomic model is shown in Figure 24.

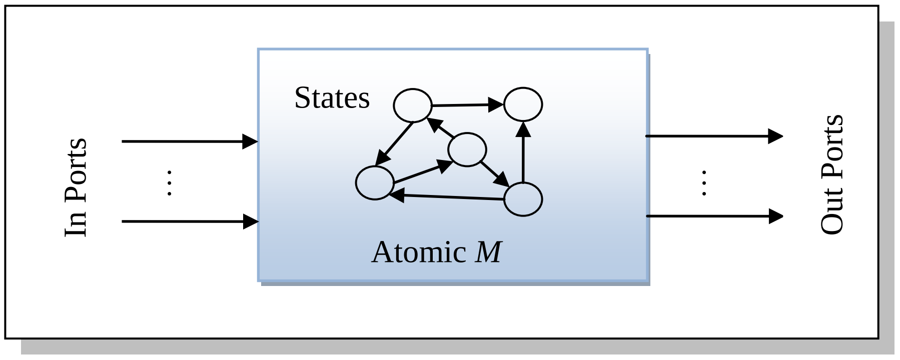


Figure 24: The DEVS Atomic Model

Another fundamental class of DEVS modeling is the coupled model which is defined as *N* below.

$$N = (X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC, Select)$$

where X and Y are the same as previously defined and *D* is the set of the component names where ∀ *d* ∈ *D*, *d* is an instance of *M* or *N* (i.e., is an atomic or coupled model). *EIC*, *EOC*, and *IC* are couplings between models as shown in Figure 25 where *EIC* is External Input Coupling, *EOC* is External Output Coupling, and *IC* is Internal Coupling. These various couplings are also shown in Figure 25. The *Select* entity is a tie-breaking function when two events are generated at the same simulation time.

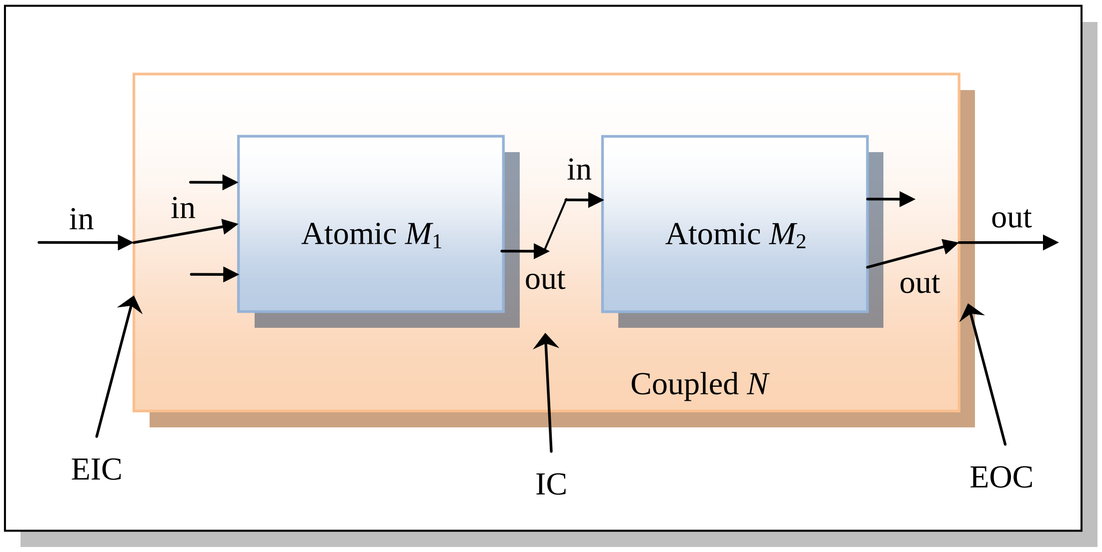


Figure 25: The DEVS Coupled Model

Coupled models may be combined to created hierarchical models to any arbitrary depth as needed in handling the complexity of the models being simulated.

In summary, DEVS is a powerful modeling framework in which to create models of any level of complexity to handle discrete events and maintain state information on a per-model/object basis. Having provided background to the DEVS framework, we turn our attention to its use to simulate our architectural approach to RTCES.

3.5.2. Adding a Lock-Proxy to a CMS

To validate our architecture and experimentally determine whether a lock proxy approach could improve concurrency, we simulated two configurations of our architecture – one in which the lock proxy was absent (as in a traditional distributed repository) and one in which the lock proxy was present (as serving to implement fine-granular locking). We utilized the discrete event DEVS Java simulation framework for this study [162].

Both simulations connected numerous clients to a set of servers hosting CMS (document repositories) through a network. Clients simulated 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 lock proxy component to the server that intercepted document requests from clients and processed these requests as a proxy to the server; this component is shown as a dashed box in Figure 26 to denote that it was not present in the original simulation configuration. The client edit behaviors were the same in both simulations. These simulation configurations are illustrated in Figure 26; note that if the lock manager was not present, the Web Services API would communicate directly with the document repository (CMS).

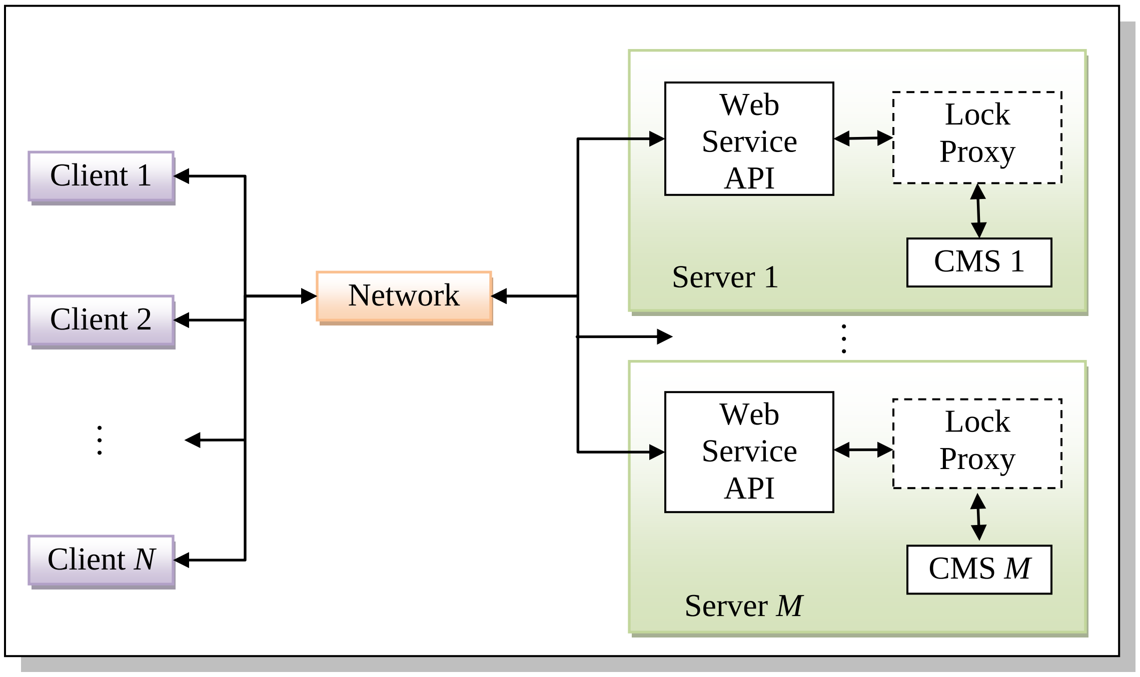


Figure 26: Simulation Configuration (shown with Lock Proxy)

Figure 27 shows the architecture implemented in the DEVS Java simulation package. The simulation was designed so that client users and servers could be added easily upon initial configuration; the lines connecting the components denote discrete event message paths within the simulation (i.e. requests for check in and check out, success or fail messages from the server, etc.), thus that the entire collaborative editing system was modeled accurately. In Figure 27, the lock manager component is shown and labeled as “middleware.”

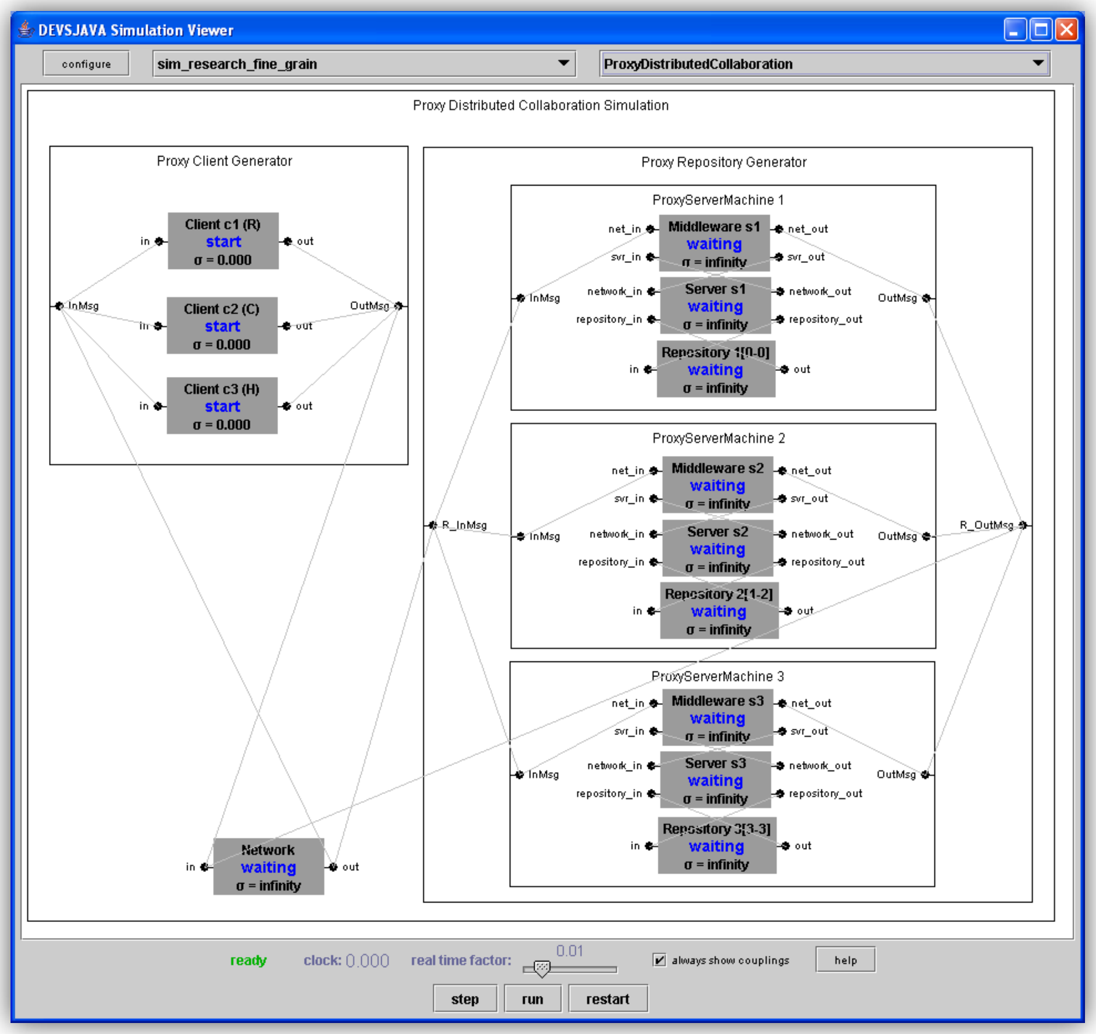

DEVSJAVA Simulation Viewer
configure
sim_research_fine_grain
ProxyDistributedCollaboration
Proxy Distributed Collaboration Simulation
Proxy Client Generator
Client c1 (R) start σ = 0.000
Client c2 (C) start σ = 0.000
Client c3 (H) start σ = 0.000
in
out
InMsg
OutMsg
Network waiting σ = infinity
Proxy Repository Generator
R_InMsg
R_OutMsg
ProxyServerMachine 1
Middleware s1 waiting σ = infinity
Server s1 waiting σ = infinity
Repository 1[0-0] waiting σ = infinity
ProxyServerMachine 2
Middleware s2 waiting σ = infinity
Server s2 waiting σ = infinity
Repository 2[1-2] waiting σ = infinity
ProxyServerMachine 3
Middleware s3 waiting σ = infinity
Server s3 waiting σ = infinity
Repository 3[3-3] waiting σ = infinity
net_in
svr_in
network_in
repository_in
net_out
svr_out
network_out
repository_out
ready
clock: 0.000
real time factor:
0.01
always show couplings
help
step
run
restart

Figure 27: DEVSJAVA Simulation of Lock Proxy

On the left side in Figure 27, you see a set of clients that represent the users in the CES; we did not specify which editing software/applications the clients were using – we simply send the check-out and check-in requests denoting that the clients desire to edit (check-out) and are done with editing (check-in). The network entity connects he clients to the servers. On the right side of Figure 27, you see a set of servers; we allow the set of documents to be spread over a heterogeneous set of servers, thus each server publishes a Web Service API that standardizes how clients may request check-ins and check-outs of documents. Notice that it is transparent to the clients as to whether the server is running any particular configuration management software (RCS, CVS, VSS, etc.). The connecting lines in Figure 27 denote the message paths from clients through the network, from the network to the servers, and internally within the servers' Web Services API to the lock manager/proxy and then to the existing/legacy CMS. Also note that the lock manager/proxy was only present in the section version of the simulation; it was left out in the first version of the simulation to see if the addition of this proxy improved check-out fail rates.

There are three types of clients in the simulation: random, clustered, and hybrid. These clients represent the broadest range of edit patterns among users/editors within a collaborative editing session. 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 checked 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 3 shows 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.

Table 3: Lock Proxy Simulation Configurations

		Client Distribution (# per type)			Repository Distribution (# Artifacts at each Server)								
Test	Iterations	Random	Clustered	Hybrid	S1	S2	S3	S4	S5	S6	S7	S8	S9
1	500	1	1	1	1	2	1	0	0	0	0	0	0
2	500	3	0	0	1	2	1	0	0	0	0	0	0
3	500	0	3	0	1	2	1	0	0	0	0	0	0
4	500	0	0	3	1	2	1	0	0	0	0	0	0
5	500*	1	1	1	10	20	10	0	0	0	0	0	0
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	0	0	0	0	0	0

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

Table 4: Lock Proxy Simulation Results

	Check-out Fail Rate		
Test	Without Lock Proxy	With Lock Proxy	Improvement
1	32.75%	7.27%	78%
2	23.33%	11.67%	50%
3	26.92%	6.38%	76%
4	19.64%	7.02%	64%
5	2.00%	0.75%	63%
6	16.39%	5.81%	65%
7	7.91%	2.62%	67%
8	9.08%	2.99%	67%
9	26.55%	7.24%	73%

As shown in Table 4, check-out fail rates for the simulation configuration without the fine-grain locking ranged from 2% (test 5) up to 32.75% (test 1). Check-out fail rates for the simulation configuration with the fine-grain locking ranged from 0.75% (test 5) up to 11.67% (test 2).

In all configurations, the version of the simulation that contained the fine-grain lock manager significantly outperformed the other version (without the lock manager) 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 study 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 study 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).

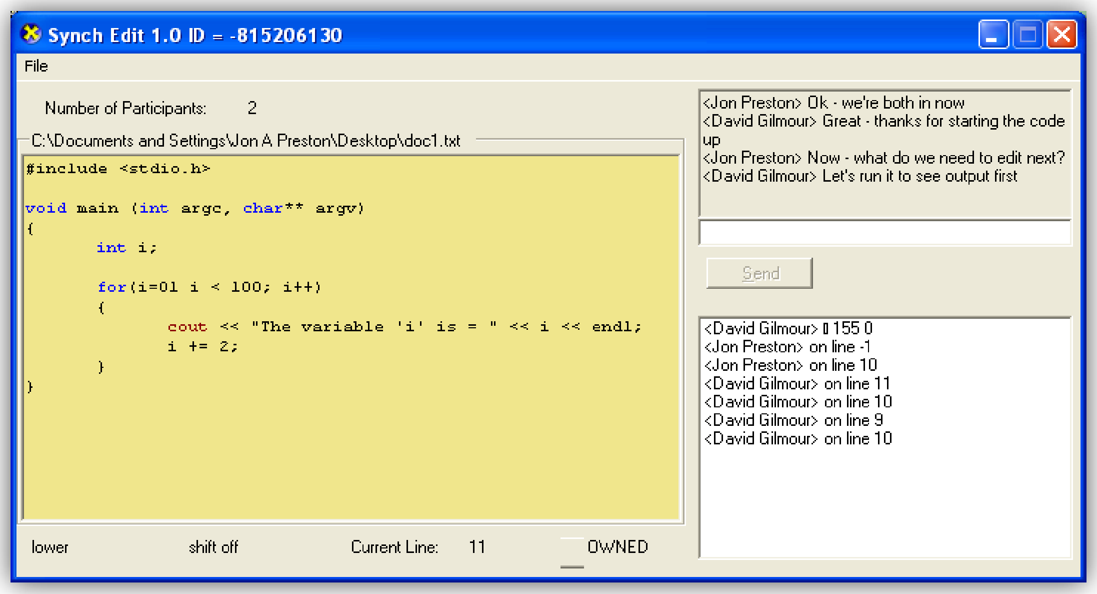
The results indicate that the concurrency is maximized at some number of artifacts relative to the number of clients. Examining the difference between test 7 and test 8, the decrease in the number of artifacts by 50% does not show any appreciable difference in the improvement rate. Consequently, we may infer that both of these tests had a sufficiently large set of artifacts from which the clients could make use of such 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 lock manager is added, even though the number of artifacts is large enough to handle the client requests well in both simulation configurations.

3.5.3. Measuring Communication Costs

The second study we performed to validate our architecture involves measuring the communication cost associated with keeping users notified with the RTCES. In this study, we implemented a simple client editor that communicated with other users within the RTCES using a P2P networking approach. The client editor allowed the users to share a common document and chat via an instant message (IM) window.

The system was implemented in C# with DirectX 9 using peer-to-peer networking. The visual interface provides the users the ability to edit the collaborative space, send text chat messages, and log all interactions with the shared space.

The peer-to-peer aspect of the system is particularly interesting; no centralized server acts as a single point of communication bottleneck or failure, and in this system, the host is able to migrate if the original peer host leaves the session.



Number of Participants: 2

C:\Documents and Settings\Jon A Preston\Desktop\doc1.txt

```
#include <stdio.h>

void main (int argc, char** argv)
{
	int i;

	for(i=01 i < 100; i++)
	{
		cout << "The variable 'i' is = " << i << endl;
		i += 2;
	}
}
```

lower shift off Current Line: 11 OWNED

```
<Jon Preston> Ok - we're both in now
<David Gilmour> Great - thanks for starting the code up
<Jon Preston> Now - what do we need to edit next?
<David Gilmour> Let's run it to see output first
```

```
<David Gilmour> ▯155 0
<Jon Preston> on line -1
<Jon Preston> on line 10
<David Gilmour> on line 11
<David Gilmour> on line 10
<David Gilmour> on line 9
<David Gilmour> on line 10
```

Figure 28: A Simple Real-time Collaborative Editor Using DirectX 9

The system is I/O bound, so communication time dominates. Though other messages are sent and managed by the DirectX 9 code for establishing connections and joining, there are only six types of data packets that are we send in this system:

(1) JOIN - A peer has joined and is added to each existing peers' local list of peers (i.e., the peers now "know" about the new user/peer).

(2) LEAVE – A Peer leaves the P2P collaboration and must be removed from each existing peers' local list of peers (i.e., the peers now "know" the peer has left).

(3) SYNCH - A peer requests the current state of the shared document, and the host responds with the current state of the shared document.

(4) CHAT - A peer has placed content into the chat window and sends this content. The chat content is sent to all peers. This message type does not deal with shard editing.

(5) POSITION - A peer has updated its position (line owned) in the shared content window, and the new position is sent to all peers. This update changes the mutex for each peer (i.e. each peer tracks what other peers "own").

(6) MODIFICATION - A peer has made a modification to the shared content, and the modification event is sent to all peers. Each peer must then update its local copy to ensure all copies are synchronized to include the modification.

Figure 29, Figure 30, and Figure 31 summarize the communication costs for simulations using various numbers of peers and different events/messages. High (100mbps) and low bandwidth (33.6kbps with 2% packet loss) tests were executed with reasonable/usable communication latency due to the system's low communication overhead.

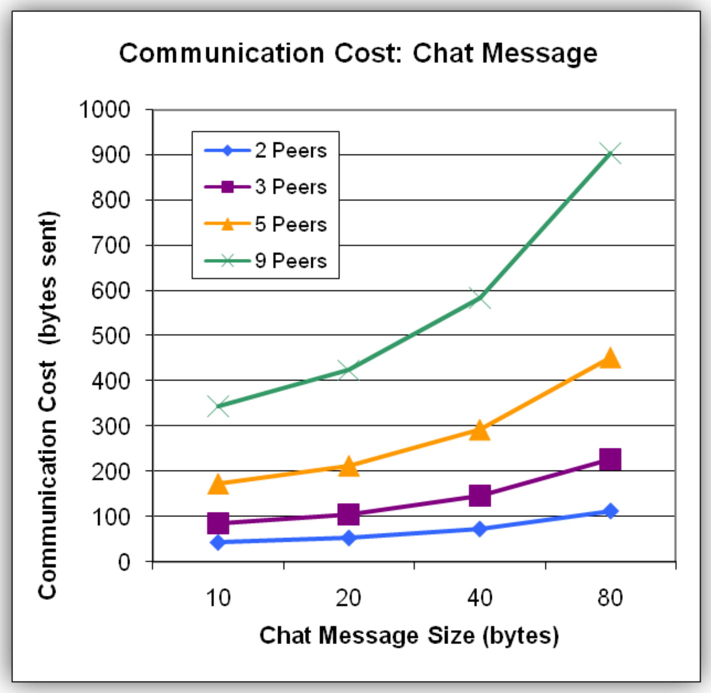


Figure 29: IM/Chat Communication Costs

The results shown in Figure 29 demonstrate the communication cost (in bytes) for sending chat/IM content between the peers. In this scenario, a user entered 10, 20, 40, or 80 bytes of content in to a communication chat box and then press a “send” button. The content typed would be then distributed to all peers. As expected, as the number of peers increases, the communication cost to distribute the message to the peers also increases. The number of packets sent in each event in this chat experiment is equal to 2(*n*-1) where *n* is equal to the number of peers in the system; this is as expected since the originator (the peer that generated the message) does not send itself the message, and each peer requires a *send* and an *acknowledgement* packet across the network. Also as expected, as the size of the message increases, the communication cost to distribute the message to the peers also increases since more total packets must be sent to all peers. When using a low bandwidth (33.6kbps with 2% packet loss) network simulator, the communication time was equal to approximately 55(*n*-1) milliseconds where *n* is equal to the number of peers in the system. This shows that, overall, the communication delay and total packets sent is small with little overhead for the communication of the chat/IM content – representing an efficient messaging system within this DirectX 9 implementation of our P2P architecture for a RTCES.

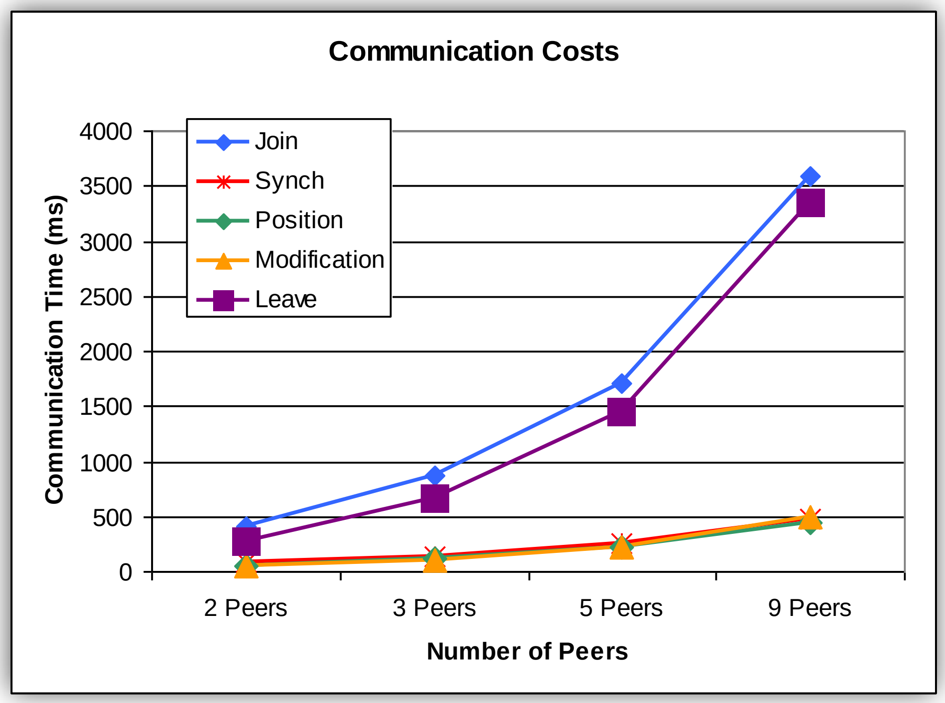


Figure 30: Communication Costs for DirectX 9 P2P RTCES Prototype

Figure 30 shows the communication cost (in milliseconds) for the five non-chat events/message types. The most costly of these are the join and leave messages where a user enters or leaves the RTCES; as the number of peers increases, as expected, it becomes more costly to notify all the other peers upon a join or leave event and have them update their internal data structures and communication channels to the peers within the system. It is important to note that the most common events – synchronization, position updating (i.e., a user moves to another section of the document), and modification (i.e., a user has made a change to the section they own) – do not incur a large communication cost. The position and modify events generate a cost of 2(n-1) packets and the synchronize event generates a cost of 2(n-1)+c packets where n is equal to the number of peers in the system and c is $\lceil$content size / 1000$\rceil$ (i.e., the number of packets to send the content itself), and the time cost is approximately 55(n-1) milliseconds. Thus again, overall, the communication delay and total packets sent is small with little overhead for the synchronize, position, modify, enter, and leave events – representing an efficient messaging system within this DirectX 9 implementation of our P2P architecture for a RTCES.

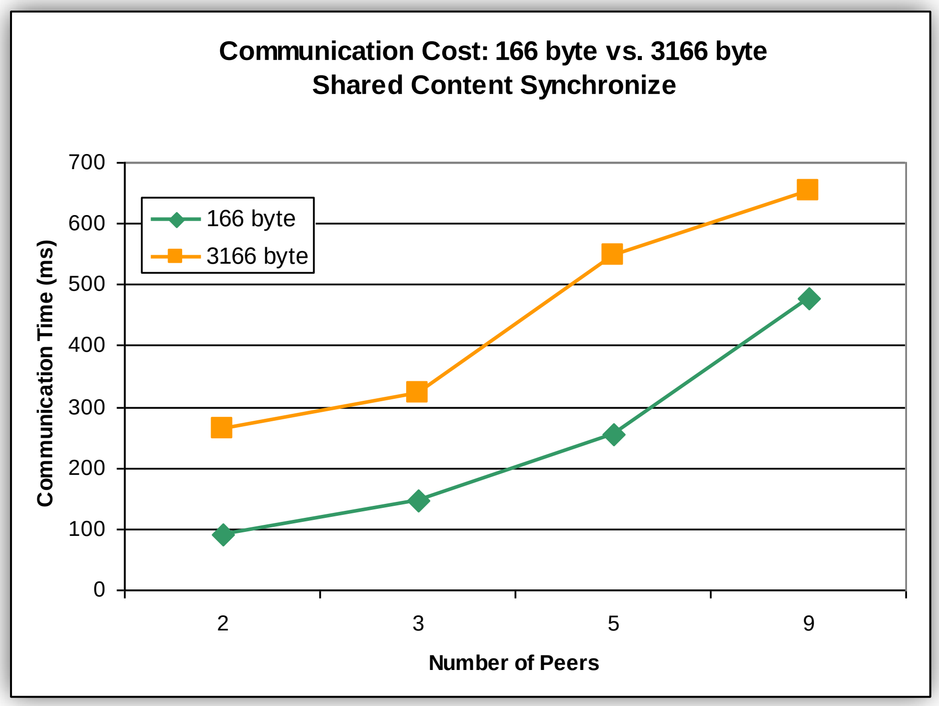


Figure 31: Synchronize Communication Cost for Varying Content Size

Figure 31 shows the communication cost/latency (in milliseconds) for synchronizing a peer with new content. This occurs when a peer enters a section that is "stale" within its local copy (i.e., not current with another peer's copy) and must be notified of the most current content. In this simplistic implementation, the synchronization request is broadcast to all peers, thus as expected, the communication cost increases with respect to the number of peers within the system. The synchronize event generates a cost of 2(n-1)+*c* packets where *n* is equal to the number of peers in the system and *c* is $\lceil \text{content size} / 1000 \rceil$ (i.e., the number of packets to send the content itself), and the time cost is approximately 55(*n*-1) milliseconds. As expected, as the number of peers increases, the time to synchronize increases. Additionally as expected, as the size of the content to be synchronized increases, the time to synchronize increases. In the most costly scenario with 9 peers synchronizing 3166 bytes of content, the effective communication time to broadcast the request to all peers and receive a response was 652 milliseconds. This shows that synchronizing content among peers is reasonable with respect to the number of packets communicated and time to complete the synchronization.

3.6. Discussion and Related Work

The open systems architecture presented in this chapter demonstrates that heterogeneous client and heterogeneous server technologies may be combined to support RTCES as well as asynchronous collaborative editing. Hooks may be added to existing client editing software to listen for edit events that should be sent to other users within the collaboration; and a Web service front-end may be placed atop existing server repositories to create a unified API to clients of these server-side tools.

Others in the RTCES community have proposed and even developed collaborative hooks into existing applications; we see CoWord as a real-time collaborative extension to Microsoft Word [138] and CoPowerPoint [158] as a real-time collaborative extension to Microsoft PowerPoint, and we see that this technique may be extended to open systems applications as well in CoStarOffice [122][136]. But the problem in all of these homogeneous systems is that a function to respond to the edit event and transmit it to each user within the collaboration must be written for each edit event in the existing client tool; given that there are many edit events and many paths to execute the same edit event in existing client editing tools, it is quite difficult to cover all features. As a result, these previous systems have only implemented a small subset of the features within their respective products. Our work is not so much seeking to replicate what has already been done, but rather we demonstrate an overall architecture by which these existing techniques and technologies can be incorporated into a larger system supporting users of various client technologies

Further, heterogeneous clients may also be connected collaboratively in real-time editing [68], and the problem of covering features within the client tool is exacerbated in that now each feature must be covered for each client in the heterogeneous set of clients, but also there must be a mapping/translation of edit events in each client tool to every other client tool.

Our proposed lock proxy may be added to legacy repositories and CMS to extend their capabilities in supporting asynchronous collaboration and a sub-file level locking such that more than one user may edit a shared document if the sections being edited by the different users do not overlap (i.e., the sections are distinct). Other systems such as Coven [16] and COOP/Orm [75] also allow the lock to be made at a sub-file level, and the POEM system [71] utilizes the hierarchical nature of software code to lock at a sub-file level; one of the shortcomings of these systems and a limitation present in our simulation in this chapter is that the unit of locking is fixed in size and is not adjustable. For example, if two users wanted to edit different parts of the first section of a shared document, all of the previous systems and our simulation would not be able to accommodate both users – only one would have write access. But if the amount of the shared document was not fixed in size and could dynamically adjust to accommodate users at various semantic levels within the document, then concurrent access could be maintained while still avoiding synchronization of all edit events (OT type consistency maintenance). This dynamic locking will be presented in the next chapter and is the continuation of our work.

One presupposition that the RTCES research community makes is that a replication of the shared document on each client's site is necessary given the network latency and to preserve responsiveness of the editor for the local user. But our research in this area indicates that even for moderately sized collaborations of up to 9 users, the communication latency was reasonable even when simulating a 33.6kbps with 2% packet loss network speed. As a result, we believe investigating RTCES that employ intelligent locking is merited.

3.7. Summary

Because the RTCES research community has primarily adopted a replicated approach with OT-based consistency management for sharing a common document, communication costs and the time needed to achieve consistency have not been previously addressed. It is assumed that high local response time is more critical and that consistency among the replicas may be delayed. Additionally, with a few exceptions, the RTCES community has not adequately addressed the opportunities for an open systems approach with regard to integrating existing client and server technologies. Our approach as presented in this chapter demonstrates that not only is such an open systems based architecture viable, the communication costs and latencies in supporting non-replicated (i.e., round-trip) consistency approaches for RTCES are sufficiently low. Better still, if the overall number of messages needed to ensure consistency can be kept small, then the communications costs of our architecture will outperform existing OT-based solutions. Thus, we focus the next phase of this research in adopting hierarchical locking techniques on document trees such that we can minimize the total messages required to ensure consistency within a RTCES. The next chapter details this next phase of our research.

CHAPTER 4

ENABLING RELAXED CONSISTENCY TO REDUCE RTCES COSTS

Having established a viable open-systems architecture for RTCES, we now focus on reducing communication and computation costs associated with traditional OT-based consistency approaches. OT approaches are costly in that all operations are immediately broadcast to all users within the collaboration, thus in effort to reduce costs in an RTCES, we adopt a more relaxed consistency model in which not all users within the RTCES have the most current copy of the document – rather, all users have the most current copy of the section of the document they are viewing (i.e., the visible/focused portion of the document is always current on a user-by-user basis). By relaxing the consistency constraint within the RTCES, we are able to reduce communication and computation costs while at the same time improve the intention preservation of users.

We agree with [28] that conflicts are a “naturally-arising side effect of the collaborative process” and “will occur simply because of the semantics of multi-user applications.” Further we agree with [47] that “temporary inconsistencies are necessary to achieve good performance” within collaborative editing systems. Our approach is motivated by noting that some distributed systems such as DNS that allow lazy updating and temporal inconsistencies through “eventual consistency” [144]. Thus, at various points in time, the copies of the document are not consistent, but the distributed, managed copy of the document in its entirety is correct and preserves user intention; further, we record ownership and change history sufficient to recreate the entire document as needed (i.e., when a user wishes to view any specific section). These changes will be communicated and replayed among local copies as the users move about and view new sections, and changes can also be sent among the users (moving changes up the tree – minimizing communication costs) at specified intervals if desired [109][110]. Selective multicast is employed to improve communication cost [70].

This chapter presents our research in relaxed consistency and caching utilizing a document tree residing on a server with client editors connecting to the server for document state changes and lock/unlock requests. We first discuss our approach in modeling a document as a tree in Section 4.1 and discuss the benefits of maximizing the space within the tree that a user owns in Section 4.2. We then present our data structures and algorithms in Sections 4.3 through 4.5. The complete listing of the algorithms is presented in Section 4.6 with an analysis of correctness and efficiency. We present the simulation results that validate our approach in Section 4.7. Finally, we conclude with a discussion and summary in Sections 0 and 4.9 respectively.

4.1. Modeling Document Structure via a Document Tree

Traditionally, research within CES has viewed documents to be a linear sequence of data; consequently, OT and other techniques to ensure the CCI model [134] are designed to work on linear content. More recently, others have proposed leveraging the semantic structure of the document and viewing it as a hierarchy [59][60][104] [Ignat 2002]. Operations to ensure CCI are more efficient when applied to sections of a hierarchical document as opposed to the entire document, and the system is better able to handle context-specific consistency/intention preservation [57][137].

Because any section of a document may contain any number of text elements (paragraphs, sentences, etc.) and may contain any number of sub-sections, we generalize our previous algorithms [103] for inserting and removing locks from the collaborative space to work within an n-ary tree data structure that is representative of a shared document.

We extend this view of the document as a hierarchical structure; in addition to better achieving context-specific consistency preservation, we can reduce communication and computational costs. Based upon the semantic structure of the document, the document may be broken up into sections, subsections, paragraphs, sentences, words, etc. If the document being shared is a CAD drawing, it may be broken into layers, objects, etc. If the document is programming source code, it may be broken into classes, components, methods, blocks, etc. Thus we do not have any preconceived notion of what the sections of the document contain, nor do we require any specific depth/level of decomposition. Our approach works well with a variety of document structures. Note that the document tree consists of internal nodes that represent structure, and all document content resides at leaf nodes.

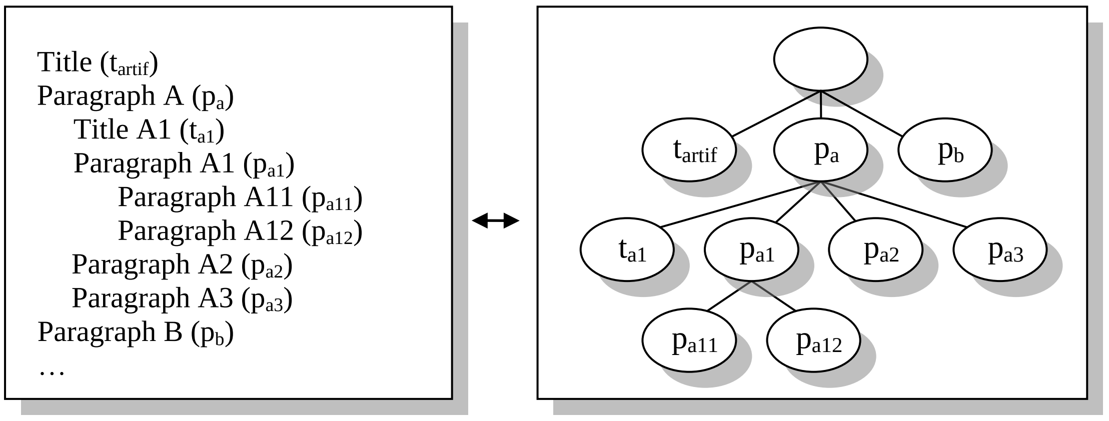


Figure 32: Mapping a Document to a Document Tree

The path finding algorithm of Rao and Kumar [111] uses binary encoding to uniquely identify a path from a vertex *n* to a vertex *v*. Since we do not mandate a binary tree structure, we extend this algorithm to support a mechanism for correctly identifying the path from vertex *n* to a vertex *v* in an n-ary tree. We do this by defining node identifiers and a function NEXTINPATH(N, V) from *n* to *v* as follows.

First, let *E* denote the identifier of a vertex *v*. *E* then defines a path *p* from the root to a vertex *v*; *E* consists of a string of *d* entities, where *d* is the depth of *v*. If the root is desired, then *E* = "" (empty string) since the root is at depth 0 (d=0). Each entity in *E* specifies which sub-tree to follow in the path to *v*. Consequently, the cardinality of *d* must be equal to the branching factor of the vertex with the largest set of children (i.e. $|d|$ = maximum branching factor of the tree). Assume $d = \{d_1, d_2, d_3, \dots d_n\}$, where $n = |d|$. If the path *p* contains the edge from vertex v_k to the ith child/sub-tree of v_k (where v_k is a vertex at depth *k*), then the (k+1) entity of $E = d_i$ (i.e. traverse into the ith sub-tree of v_k).

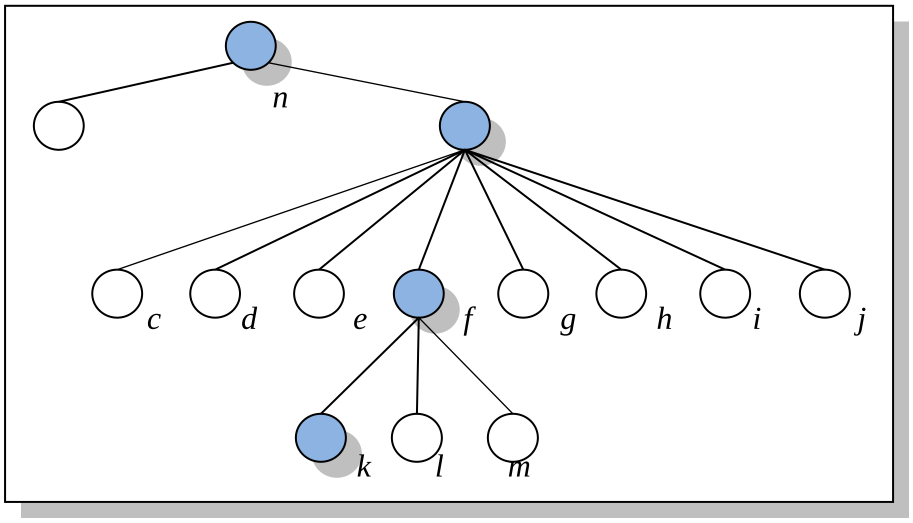


Figure 33: Path Finding in the Document Tree

Using as an example, in Figure 33, a path from the root *n* to vertex *k* may be defined by E_k = "241", and the path from the root *n* to vertex *h* may be defined by E_h = "26".

Thus we uniquely identify each vertex in the tree, and the identifying string for each vertex defines a path from the root to the vertex that can be found in O(1) and may traverse any path in O(*h*) where *h* is the height of the tree.

While this identifying scheme requires more memory than the simpler binary identification of Rao and Kumar, it is more flexible in that it works with n-ary trees. Given a tree depth of D_t and a maximum branching factor of B_t, the largest identifier required for any vertex would occur at a leaf node at depth D_t, be represented by B_t*D_t, and consist of $D_t * \lceil \log_2 B_t \rceil$ bits. Additionally the memory required to represent the entire tree is $\leq \sum_{i=0}^{D_t} B_t^{\,i}$, which is quite reasonable given that the branching factor of the document tree is defined by the largest number of subsections within any section, and the depth is defined as the "deepest" subsection of the document.

4.2. Maximizing Owned Space and Caching

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

The dynamic lock management algorithms focus on granting a user exclusive access for writing to the section of the shared document. In addition to supporting dynamic, exclusive writer locks, the system also supports multiple, simultaneous readers for a section of the document. It is permissible to allow multiple users to view the changes being made by another user, and thus the n-ary tree used to manage the write locks of the document is also used to manage the viewing positions of all users within the document.

For example, if a collaborative editing session included five users, $U = \{u_1, u_2, u_3, u_4, u_5\}$, where u_1 was editing Section 1, u_2 was editing Section 2, u_3 and u_4 were viewing Section 1, and u_5 was viewing Section 2, this would be stored in the n-ary tree, shown in Figure 34.

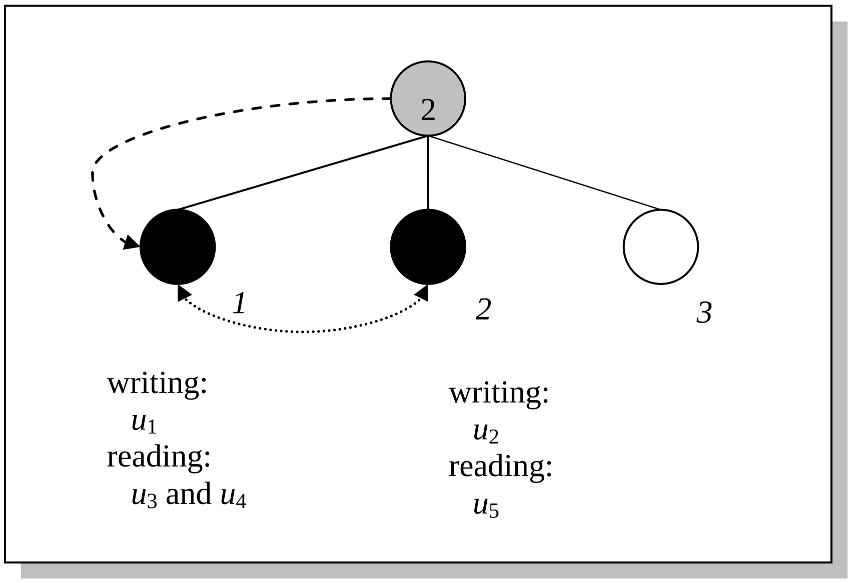


Figure 34: Supporting Multiple Readers and Writers

If we adopt such a cache based approach, then broadcasting all changes is not required (an improvement over existing OT approaches). We may communicate only to other readers/writers within the changed node. Additionally, readers need not perform

OT since they are not editing (they can't have any local changes on which to transform the new operation) and we can reduce the number of clients that need to perform OT (bound by the writers within the node). These are significant communication and computation improvements over existing OT systems. Further, our approach affords opportunities where we may decrease the history buffer (HB) size. This can be done when inverse operations (where inverse is denoted as ¬) are applied and a policy of flushing the previous operations is approved (i.e. we don't need to "undo" the operation and its inverse operation). In this case, Op + ¬Op allows removal of Op from HB (and avoiding placing the ¬Op into the HB at all). Additionally, we may reduce the history buffer size by consolidating multiple operations into single, semantically-higher-order operation within the tree; this can occur upon reduction or promotion as explained later in this chapter.

Central to our approach is the ability to employ lazy consistency in which portions of the document are current at only a subset of all users' copies. In this regard, we allow some portions of the copies of users to be "stale" and inconsistent (i.e., we allow operations on other users' copies to not immediately be sent/communicated to other users). We avoid the problem of the user being affected by this by tracking where the user is in the document and caching changes (not communicating these changes) if the user is not editing/viewing the space in which the change occurred.

The impact of this is that each user has the most current (and correct) content for the space within the shared document that they are interested in, but we minimize communication and computation costs by not having to immediately broadcast all changes to all users. Visually, we can view the overall document's correct (most current) state as being distributed among potentially many users as shown in Figure 35.

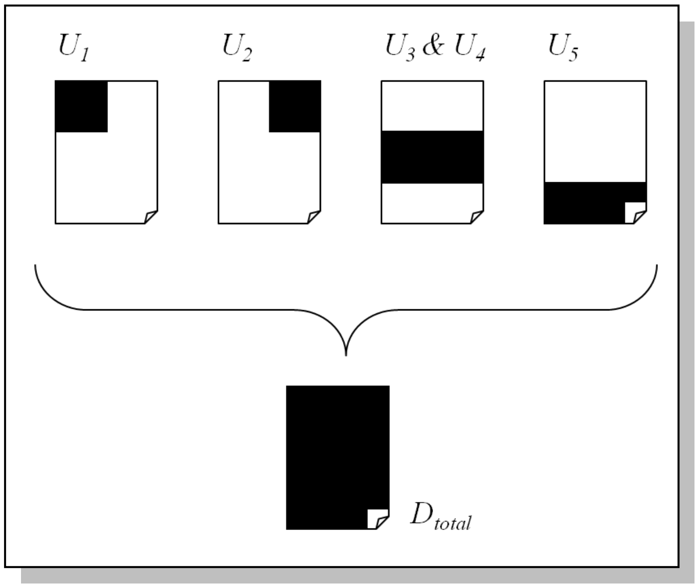


Figure 35: Distributing the Current Document State across Multiple Users

In Figure 35, the black area shows the section of the document that has been modified and cached locally among the user(s) that are currently writing in that section of the document. To compile the current state of the entire document, we can query each user and reconstruct the document according to the equation:

$$D_{total} = \bigcup_{i=1}^{n} (D_i + \Delta D_i)$$

Where D_{total} = the most current state of the entire document, D_i = the state of the document for section i (managed by some set of users), and ΔD_i = the changes that have been made (history buffer) at section D_i.

4.3. Data Structures and Algorithm Overview

Once established, the document tree is utilized to manage ownership of subsections within the document. Rather than locking the entire document, lock granularity is adjustable, ranging from the entire document (ownership marked at the root of the tree) to an atomic level (ownership marked at a leaf node in the tree). The size of a subsection is not specified within our algorithms, thus it is scalable to accommodate the semantic structure of the document being edited, similar to [93].

We allow many readers to be present within the same node within the document tree, and we define reading state based upon the visible frame within client editor (i.e., the client is assumed to have read access to any section visible within the client's editor view space – what portion of the document can be viewed within the client's editor). We may exclude multiple writers and adopt an exclusive write policy, denying other clients from writing to the locked section. Alternatively, multiple writers may be allowed within a node when exclusive writing is not desired, and this policy is defined on a per-node and per-client basis; in this case, we may adopt OT-based consistency maintenance among all writers sharing a section of the document represented by the node in the document tree. Thirdly, we can demote a lock if a client does not wish to share ownership of a larger section of the document and prefers to relinquish a portion of the owned space so that the original owner locks a portion of the document while the new, requesting client owns another, non-overlapping portion of the document. This demotion policy is also established on a per-node and per-client basis.

Rather than blocking other users from editing, lock granularity is adjusted via demotion of the lock down in the tree until the conflict among users is resolved.

Additionally, when a user leaves a section of the document and makes it available to other users, conflict among users is potentially reduced; as a result, our algorithm automatically promotes the lock to a higher level within the document tree – maximizing the amount of the document owned for the remaining user. The OBTAINLOCK and RELEASELOCK operations are the central algorithms. These algorithms traverse the document tree in a top-down fashion and are guaranteed to be deadlock free.

Each node in the document tree maintains a color (white, black, or grey) to denote whether it is available, currently being written to by another user, or if two or more users are editing sub-trees, respectively. Ownership (black coloring) of a vertex *v* by user *u* implies that *u* owns *v* and the sub-tree rooted at *v*, and is the only user that may edit node *v* or its sub-tree. If a node is white, no user owns (is currently writing) to that section of the document. Additionally, each node *n* in the tree maintains a numeric value that denotes how many nodes in the sub-trees of *n* are colored black. This is defined as the *grey-count* of the node *n*. This value is useful in determining if the node can be colored white or grey when a request to delete a user occurs and promotion is enabled (as explained later).

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

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

4.4. Lock Request

The basic idea behind the OBTAINLOCK algorithm is to traverse the tree from top to bottom toward the desired leaf node along an insertion path and eventually obtain a 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.

A user requests a section of the document to which he wants to write, and the system attempts to obtain a lock on that section of the document. The OBTAINLOCK algorithm works from top-to-bottom by examining nodes in the path from the root to the destination node. As it traverses this path, if a white node is found, then the lock request succeeds and the node becomes owned by the requesting user (and painted black). If a grey node is found, it continues down. If a black node is reached, then we need to *demote* (push down) this black node (its current owner/user), turn this node into grey thus making room for the new insert request to continue down.

Demotion works by moving the ownership of that user (and the black coloring) down the tree hierarchy while ensuring that the leaf node needed by that user is contained within the sub-hierarchy. If the black node reached is an "atomic" node, then we can't demote any further, and the insert operation fails (i.e., edit request is denied). Alternatively, if desired, optimistic concurrency control techniques such as OT may be employed at this atomic level; by keeping a list of writers, a selective multicast of all changes within this atomic section could be made to all writers, limiting the computation and communication cost to a subset of all users within the smaller section of the document.

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

When an OBTAINLOCK request is successfully fulfilled, we have two cases – (1) there was no contention and no demotion, and thus a white node is painted black, or (2) there was contention and this contention is resolved via demotion and by adjusting node coloring. Let's begin by starting with the document tree state as shown in Figure 36.

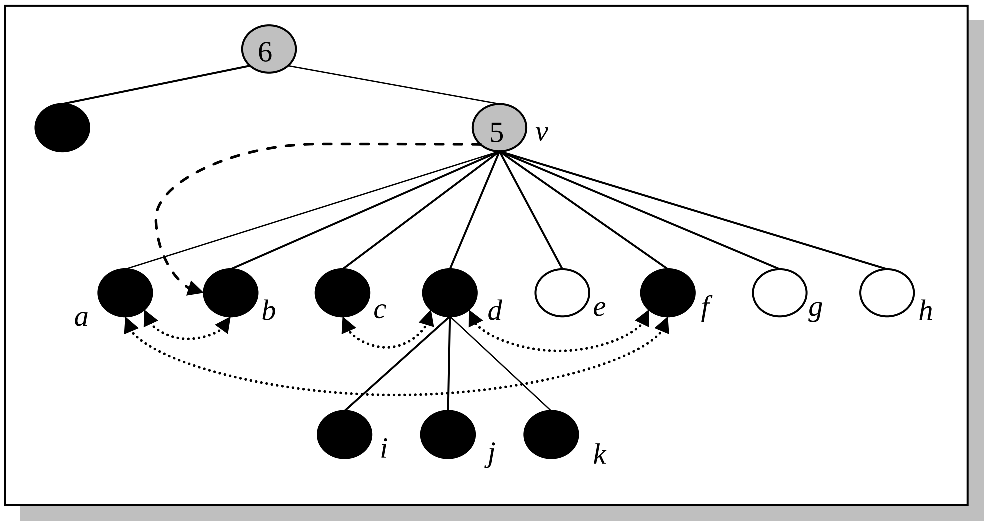


Figure 36: Original Document Tree State

In the first case (no demotion), a white node must be painted black, and the newly-painted black node must be added into the black sibling list of the grey, parent node. Assuming *h* was to be painted black, i.e. OBTAINLOCK(u_1, *h*) was invoked, then Figure 37 shows the result of painting *h* black and adding *h* as the head of the sibling list (if the document state was initially as represented by Figure 36).

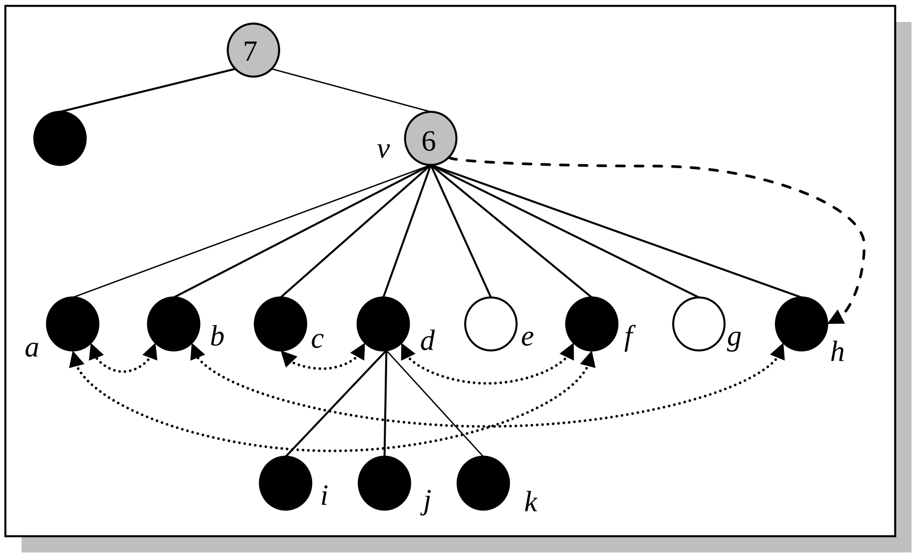


Figure 37: ObtainLock with No Demotion

In the case where an OBTAINLOCK operation requires the lock contention be resolved via a demotion of the lock, we must adjust the black sibling list to reflect the demotion of the lock. Additionally, we must link the two nodes now painted black.

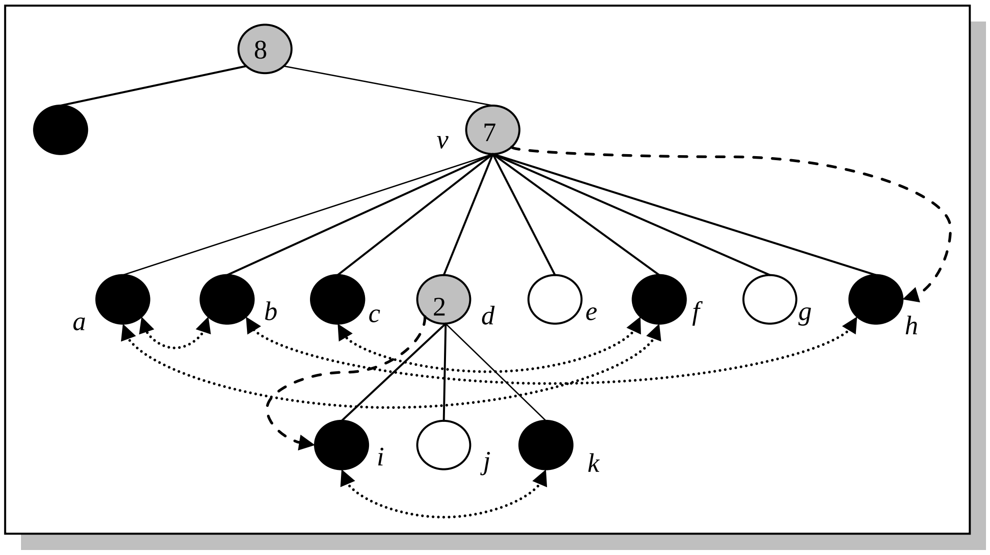


Figure 38: ObtainLock that Results in Demotion

Beginning with the document tree state shown in Figure 37, if OBTAINLOCK(u_2, *k*) was invoked and node *d* had previously been locked when u_1 had requested node *i* (i.e. node *d*'s original request reference is *i*), then the u_1's lock on node *d* will be demoted to node *i*, and then u_2 will acquire a lock on *k*. When this occurs, node *d* should no longer be in the black sibling list of its parent, node *v*. Thus we modify the OBTAINLOCK algorithm to remove this node whose lock was demoted from the black sibling list by joining the adjacent siblings of the node. Additionally, black sibling links must be established for the two black nodes that result from the demotion (nodes *i* and *k* in this example). The result for this example would be that node *c* and node *f* are now joined and node *i* and node *k* are now joined, as shown in Figure 38.

4.5. Lock Release

The REMOVELOCK algorithm works from top-to-bottom via handshake locks to avoid deadlock. As the path from the root to the node to be released is traversed downward, the grey-count for all nodes painted grey is decreased by one until a grey node with a grey-count of one (after decrementing) is encountered; when this occurs, a promotion is needed to ensure that the sibling of the to-be-unlocked node owns the largest sub-tree possible. This is the same behavior as the binary-tree based REMOVELOCK algorithm [103]. The only modification that must be made to accommodate an n-ary tree is that when promotion occurs, then the newly-promoted node *v* must be added into the black-sibling list of *v*'s parent.

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

The order that the black sibling nodes appear in the list is not significant as we only use this list to maintain adjacent siblings so that we know immediately which sibling to promote. Notice in the example shown in Figure 5, if the lock to be removed is associated with node *i*, then we know immediately without incurring a search cost that the lock associated with node *k* is the node to promote because node *i* and node *k* are marked as black siblings. Since promotion will only occur when there are two siblings (one of which no longer requires a lock and the other is associated with the lock to promote) order among the black siblings is not significant within the list. Consequently, when promotion does occur, we can simply place the node associated with the newly-promoted lock at the front of the black sibling list. Assuming we begin with the document tree state shown in Figure 38, if a REMOVELOCK(u_1, i) is invoked, then the lock u_2 has on node k should be promoted to node d. Node d is then added into the front of the black sibling list. The result of this promotion is shown in Figure 39.

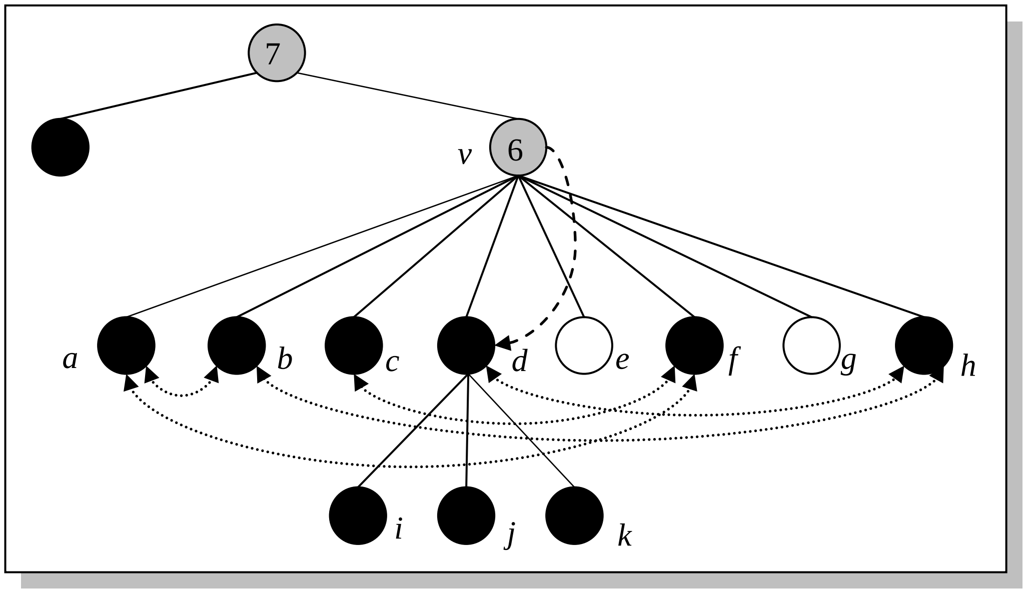


Figure 39: RemoveLock(u_1, i) - u_2 lock on node k is promoted to node d

It is possible for a situation to arise in which removal of a lock removes contention and the remaining user should be promoted through multiple levels within the tree. Figure 40 shows such a scenario, and as demonstrated, our algorithm handles this, promoting the remaining lock to maximize the portion of the document owned by the remaining user.

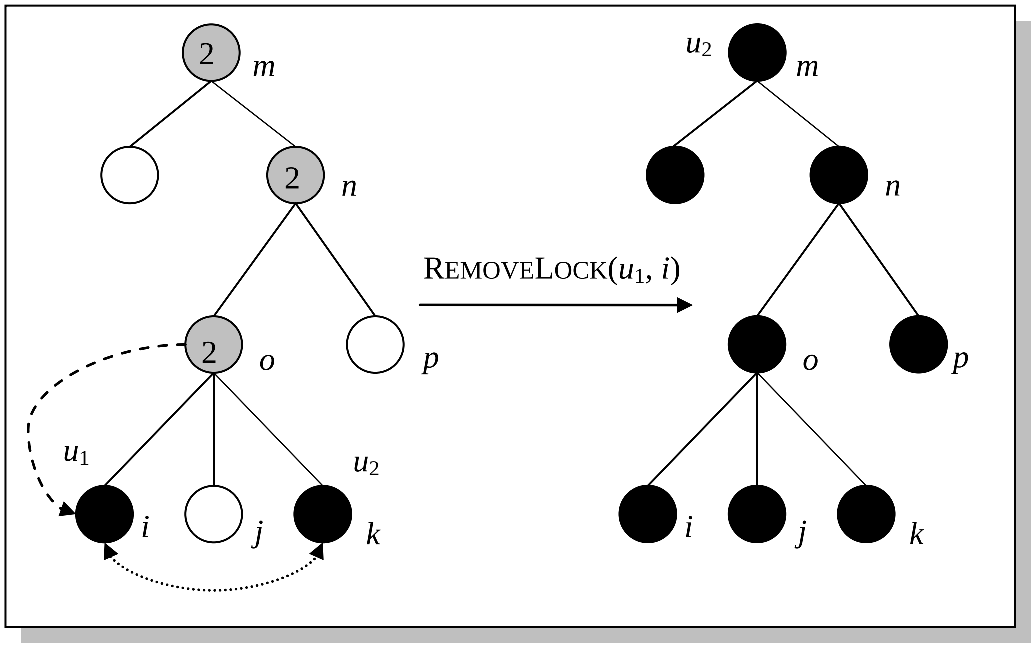


Figure 40: Promotion across multiple levels is permissible

When promotion is required during a REMOVELOCK action, we must have a way of efficiently resolving which sibling should be promoted (the remaining sibling); a brute force method could traverse all siblings until the remaining black node is found, but this is inefficient and requires O(*n*) work where *n* is the number of siblings (the maximum branching factor of the tree). Alternatively, we can maintain a back-sibling and forward-sibling reference for each node, linking the black siblings together in a list to maintain a subset of all the siblings; this subset consists of all nodes colored black (e.g., *a*, *b*, *c*, *d*, *f*, and *h* as shown in Figure 39).

4.6. Correctness and Efficiency Analysis

To demonstrate that our methods OBTAINLOCK and REASELOCK are sufficient to cover the activities that users perform within a CES, we identify a set of user actions within a CES and map these actions to events within our tree-based system. This mapping is demonstrated in Table 5. Note that if a lock is requested (via the OBTAINLOCK event) and the user already owns the lock, then no server request/communication is required. It is only when a user attempts to edit a section without having previously edited that section (i.e. the section is not owned by the user) that a request for the lock is required.

Note that these document tree events listed below support exclusive locking, but they also support multiple writers (where a lock request will never fail). In the case where multiple writers are allowed to own a section, care must be taken when the section is deleted, split, or two sections are combined where at least one of the sections to be combined are owned by other users. In these cases, coordination between the users can be enacted such that all users agree upon the action (delete, split, join), or a priority-based scheme could be adopted where a high-priority user may enact the action after the other lower-priority users' locks are revoked and reestablished. The document tree structure changes would need to be broadcast to all affected users and their locks reestablished after the structural changes have been completed. In the case where operations were performed concurrent to the structural changes, these operations could be transformed and replayed once the structural changes were completed.

Table 5: Mapping User Actions to CES Document Tree Events

User Action	**CES Document Tree Events**
Enter the CES	• Place user as a reader in the default section of the document
Exit the CES	• Remove the user from the CES and flush the user cache
Modify content within section A	• OBTAINLOCK for section A • If not successful, deny the edit
Move from section A to section B	• RELEASELOCK on section A • Place user as a reader in section B
Delete section A	• OBTAINLOCK for section A • If successful, remove section A from tree
Create section A	• Create a new node A and insert it into the tree
Combine section A and section B	• OBTAINLOCK on section A • OBTAINLOCK on section B • If either fail, release any successfully obtained lock and deny the request • Else merge sections A and B in the tree (removing section B and RELEASELOCK on B)
Split section A into sections A and A'	• OBTAINLOCK on section A • If not successful, deny the edit • Else create a new node A' as a sibling of A, move specified content from A into A'

We designed the OBTAINLOCK and RELEASELOCK operations such that the document tree is accessed only in a top-to-bottom, pipelined fashion; we do this to avoid race conditions. We enforce the policy that nodes must be accessed in a top-down manner such that we only access and 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 [111], ensures that a race condition on concurrent access to the tree data structure is avoided. As a result, our operations may be executed concurrently while maintaining their correctness.

The full presentation of the algorithms appears below in Figure 41 through Figure 43. Note that these algorithms are presented to show intent; the actual implementations feature an iterative/loop-based solution that employs a top-to-bottom, handshake-lock as the paths from the root to the desired nodes are traversed.

```
OBTAINLOCK(w, u_i)
    if w.owner ≠ u_i
        RECURSEOBTAINLOCK (ROOT, w, u_i)

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

Figure 41: The OBTAINLOCK Algorithm

Since the RECURSEOBTAINLOCK traverses from the root down to a leaf (or stops earlier if a white or black node is reached), this algorithm must traverse O(*h*) nodes, where *h* equals the height of the document tree. The work involved at each node is O(1) since the work in processing an individual node involves updating references/pointers, coloring, and grey count (integer) values. It is possible upon a lock request failure that the RECURSEREMOVELOCK function will be invoked, but this RECURSEREMOVELOCK (as discussed below) runs in O(*h*), thus it is not asymptotically greater than the existing O(*h*) work for the OBTAINLOCK algorithm. Thus the overall cost for the OBTAINLOCK algorithm is O(*h*).

```
REMOVELOCK($w$, $u_i$)
    if $w.owner = u_i$
        then  RECURSEREMOVELOCK(ROOT, $w$, $u_i$)

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

Figure 42: The REMOVELOCK Algorithm

Similarly, the RECURSEREMOVELOCK traverses from the root down to a leaf (or stops earlier if a grey or black node is reached), this algorithm must traverse O(*h*) nodes, where *h* equals the height of the document tree. The work involved at each node is O(1) since the work in processing an individual node involves updating references/pointers, coloring, and grey count (integer) values. Upon promotion, the FINDELIGIBLEPROMOTION function must be called, but it continues the traversal down the tree from the point where the promotion may occur, thus its work is also O(*h*). Thus the overall cost for the REMOVELOCK algorithm is O(*h*).

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

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

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

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

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

Figure 43: Supporting Functions

The supporting functions REMOVEFROMSIBLINGLIST, SETLOCK, LINKSIBLINGS, and UNSETLOCK are invoked by the RECURSEREMOVELOCK and RECURSEOBTAINLOCK functions. We present them here in Figure 43 to show that they all run in O(1) since they only update attributes of the nodes. The supporting function FINDELIGIBLEPROMOTION requires traversing down the path to the desired node to find the first black node along the path, thus it runs in O(h); but we note that this function is only invoked when a grey count is reduced to 1 when the RECURSERELEASELOCK function is running; when this occurs, some number of nodes have already been traversed, and the FINDELIGIBLEPROMOTION function must only process the remaining nodes below the reached node whose grey count is now equal to one. Thus the total number of nodes visited in the combination of the RECURSERELEASELOCK and FINDELIGIBLEPROMOTION functions is $\leq h$, where h is equal to the height of the document tree.

It is important to 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 44 and as the sub-tree α Figure 45 – are unaffected by the OBTAINLOCK operation. This improves the concurrent operations that are able to be performed on the tree (i.e., pipelining the operations from the top/root of the tree down. This is critical in ensuring that the lock request and release operations may be executed efficiently on the server without significant delay in responding to the clients making the requests.

Further, in the case of demotion for OBTAINLOCK as shown in Figure 45, the only modification to leaves occurs in increasing the grey count along the path from t to v and moving the ownership of u_2 to the sibling of the newly-acquired node (w in this example) when resolving conflict. If OT is adopted at a node, then no demotion is required and all sub-trees within the document (α and β in the preceding figure examples) remain unaffected.

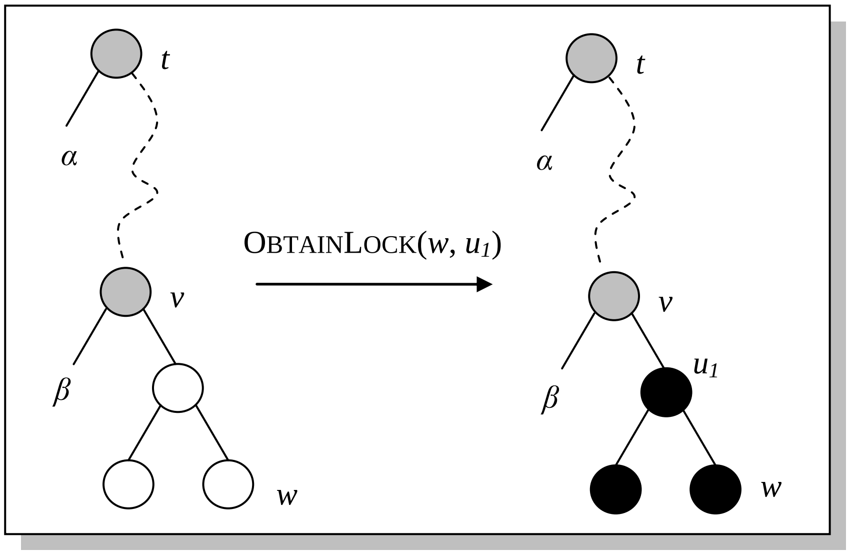


Figure 44: The OBTAINLOCK Operation without Demotion

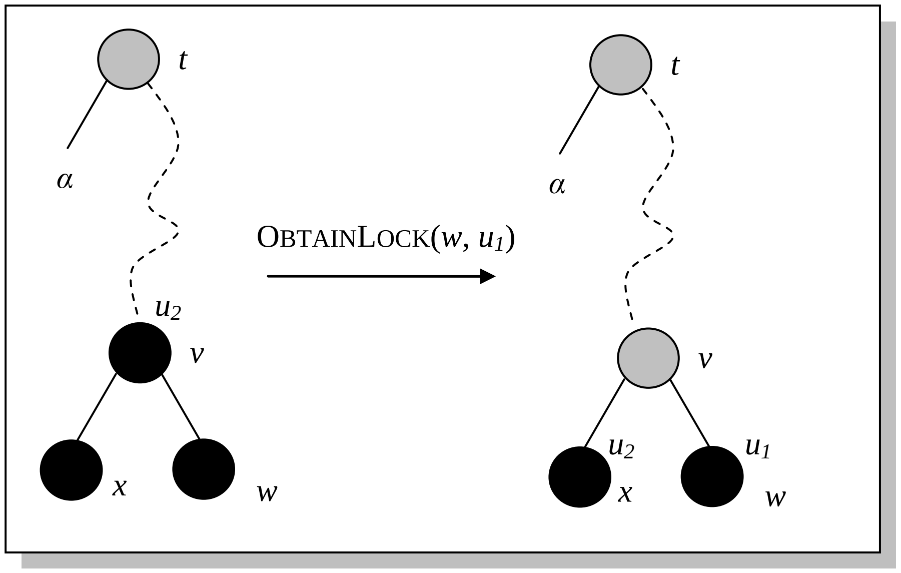


Figure 45: The OBTAINLOCK Operation with Demotion

4.7. Simulation with Exclusive Locking

This section presents our work in validating our theoretical algorithms presented earlier in this chapter. Because the intent behind our algorithms was to reduce communication costs when compared with existing OT strategies, we first discuss message costs associated with traditional, “pure” OT solutions, and then present simulation we utilized to measure the efficiency of our algorithms with respect to communication costs.

Past and present research in CES focus on the computational cost of ensuring the CCI model and assume that distributed views of the shared document are updated at the atomic user action level (i.e. character insertion and deletion); we refer to [45], [66], and [134] as exemplars. These OT-based systems send a network message (packet) upon each edit/write of any user to all other users within the CES (via broadcast). In contrast, our system caches changes locally and only distributes these changes when:

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

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

To validate the communication effectiveness of our dynamic locking algorithms, we implemented the algorithms and then ran discrete-event simulations which varied the number of users/agents as well as varied the structure of the shared document to capture communication and computation costs. Figure 46 illustrates the agent behavior states and actions modeled; the probability of the action being initiated at each time slice is denoted in parenthesis along each transition. These action probabilities are useful to obtain a mixture of reading and writing events within the simulations. Each configuration of the simulation was run such that each agent generated 1000 actions based upon the state diagram (Figure 46). To more clearly compare communication costs between our dynamic locking approach and an OT approach, within these simulations we do not allow for multiple writers.

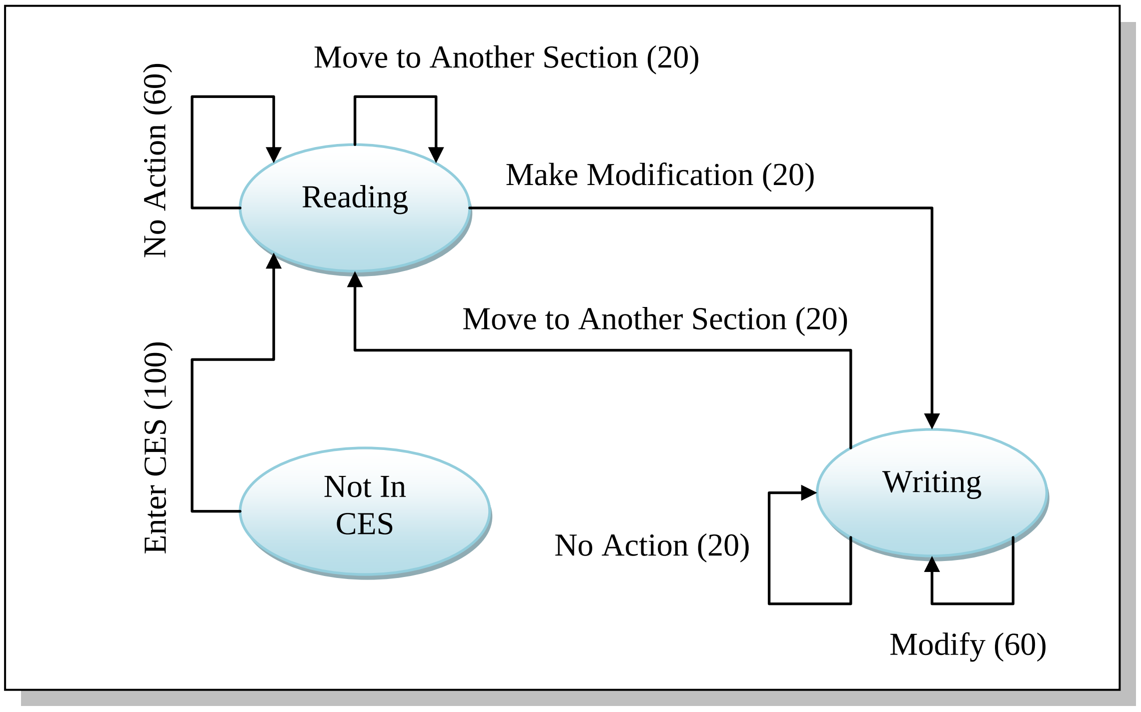


Figure 46: Agent Behavior States and Actions

The results from these simulations and the comparison to the OT-based communication costs are provided in Figure 47 and Table 6. The communication cost utilizing our approach is significantly less than the communication cost incurred by an OT-based system, and the communication cost improvement increases as the ratio of agents to sections within the document increases (as the collaboration becomes more “dense”). It is also important to note that lock/write failure is possible in the dynamic locking, but for all simulation scenarios in which the number of agents was less than half the number of document sections, no less than 64% of write attempts were successful. Of course, these write failures may be eliminated by incorporating OT at the atomic level within our document tree and using selective multicast among all writers within the shared subsection.

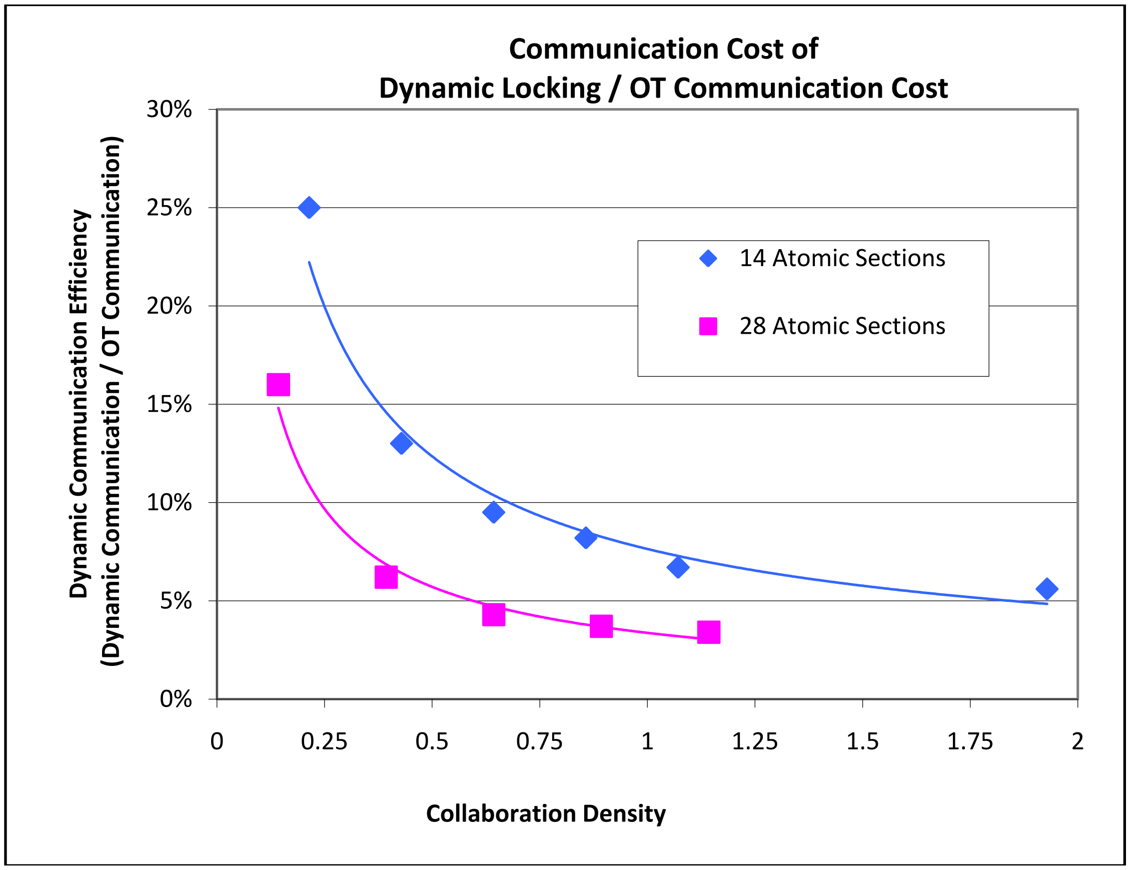


Figure 47: Communication Efficiency of Dynamic Lock Algorithm

As the data show, the efficiency of our dynamic, hierarchical locking algorithms is pronounce and we achieve a significant communication cost reduction when compared to existing OT techniques that employ global broadcast of all events. Further, as the collaboration density increases (i.e., the ratio of clients to the number of sections in the document increases), the communication savings of our algorithms over OT approaches becomes more pronounced – achieving as much as a 96.6% communication costs savings. Of course, as shown in Table 6, this efficiency gain comes at the cost of preventing some users from writing to sections of the shared document from time to time; this exclusive write policy is less than optimal.

Table 6: Dynamic Lock (Exclusive Writer) Simulation Results

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

Client to Server:	Transitioning from writer to reader necessitates flushing cached modifications to server
Server to Client:	P = Promotion; D = Demotion; lock update sent to client (adjust lock position/status)
Writer to Readers:	Incremental changes made by writer selectively multicast to readers within subsection
OT Messages:	# of write events * (# agents – 1) (since we multicast to all agents other than the originating writer)
DL Write Success Rate:	# successful modifications to document accomplished / total modifications attempted (only for the DL simulation since OT write success rate is by definition 100%)

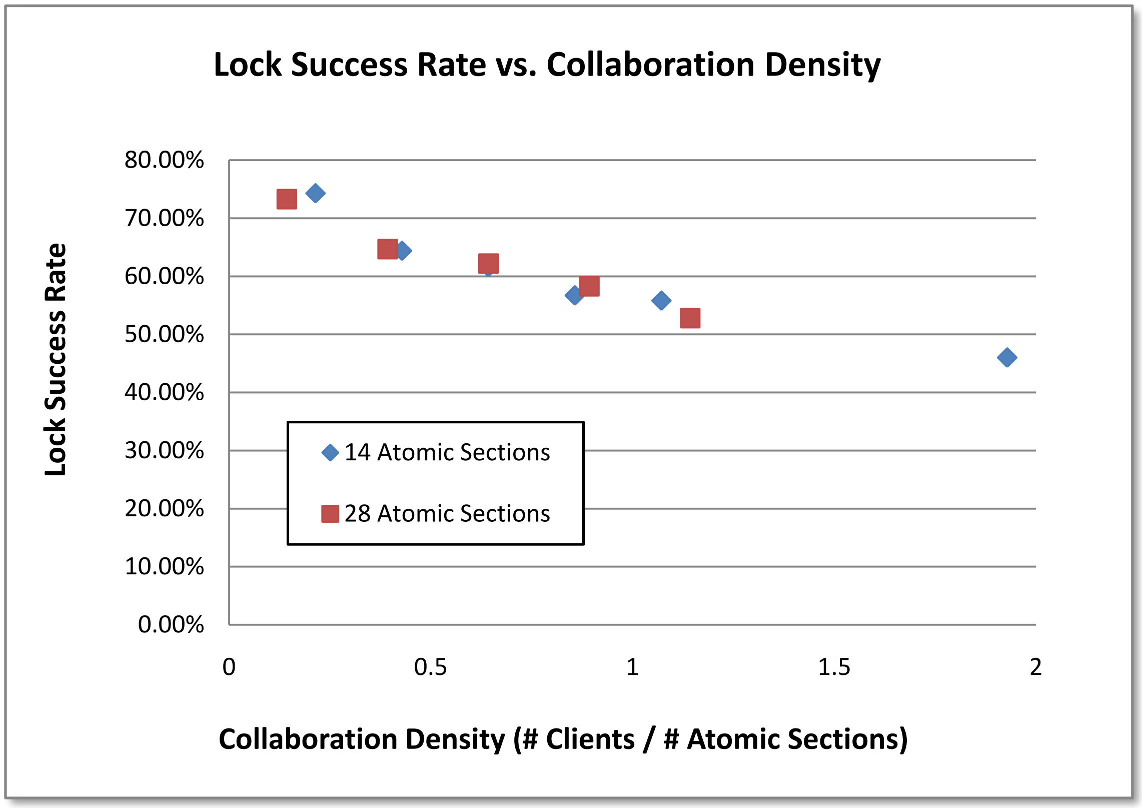


Figure 48: Lock Success Decreases with Increased Collaboration Density

As expected and as shown in Figure 48, as collaboration density increases, the chance of successfully acquiring a lock decreases. This is intuitive in that the collaboration density is the measure of contention for atomic nodes.

In conclusion, the results obtain in our simulation of the client-server algorithms that employ dynamic, hierarchical locking are able to significantly reduce the communication costs when compared to an OT approach while allowing for an improvement in concurrent access when compared to a pessimistic locking approach that only allows one user to access the document at a time.

4.8. Discussion

One may expect the lock success rate to be higher than shown in our study since contention should be reduced when some users are in a reading state; further, if the collaboration density is 1.0, then there should be a section for each user, thus lock success rate should be approximately 100%. We explain or reduced success rate in noting that in our modeling of clients' movement, clients begin in a "starting" section when they are added into the simulation; from there, when they move to another section, they determine a differential to the right or the left in the set of leaf nodes – moving earlier or later in the document. Thus as their movement progresses, if they enter the beginning (left-most leaf) or the end (right-most leaf) of the document, then they will have an increased probability of remaining in these positions. This could be solved by introducing the notion of the leaves "wrapping" such that the left-most leaf is adjacent to the right-most leaf. While this might be logical from a data structure perspective, it is not intuitive when modeling the document as a tree since the beginning of the document (left-most leaf) is not logically adjacent to the end of the document (right-most leaf). Thus another approach is needed to increase the lock success rate, and we focus on that in the next phase of our research in adding selective, localized OT to the document tree.

The structure/shape of the document tree also has an effect on the communication costs of our algorithm. The probability of promotion/demotion is related to the average branching factor of the document tree. If there is an imbalance in the branching factor of sub-trees, then the long chain of low-branching-factor paths "compress" and are not counted in promotion/demotion (i.e. clients will tend to cluster more often relative to each other, and consequently promotion/demotion will not be as likely).

If a tree is tall but narrow (small average branching factor), promotion/demotion will occur infrequently as contention remains high since there are not many branching points afforded in the document. This occurs when a document is structured such that there are few major sections but many subsections within the few sections.

Similarly, if a tree is short and wide (large average branching factor), then promotion/demotion will occur infrequently, as contention remains high since there are not many branching points afforded in the document. This occurs when a document is structured such that there are many major sections but not many subsections with these sections.

If we define:

D_l = number of leaves in document tree

D_{mh} = document mean height = Sum of heights of a leaves / D_l

Then these two extreme cases are shown below in Figure 49 where $D_{mh} \approx h$ and when $D_{mh} \approx 1$. In these cases, conflict between users is more likely and caching is less likely to occur.

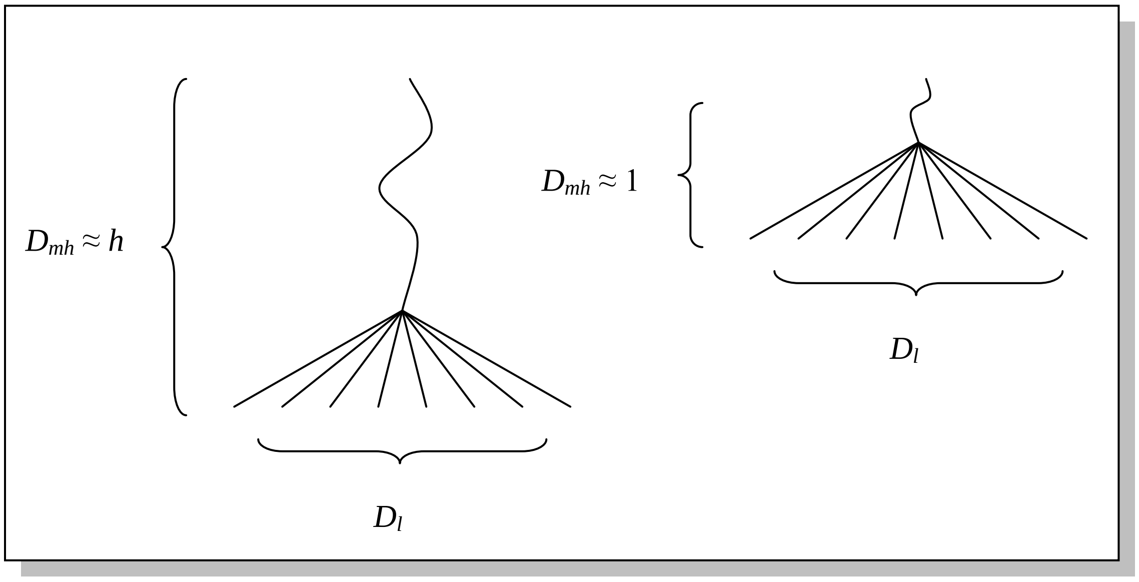


Figure 49: Two Extreme Cases of Document Tree Structure

To achieve a decrease in communication (i.e. increase localization and caching), clients will remain more often within their own sub-trees and move within their owned space without conflict with other users. This reduction in conflict occurs more often under clustered editing patterns where clients tend to cluster their edits around a single point; this reduction in conflict occurs less often under random editing patterns where clients move around the entire document (thus increasing the probability of entering an already-owned sub-tree and necessitating a demotion and a concomitant cache flush). Further, if the tree is balanced and deep with many branching paths throughout the tree (i.e., when the average branching factor is high), then more caching will be enabled and less promotion and demotion will occur. Of course, this assumes a uniform distribution of users within the document; certainly, if the users (or a subset of users) congregates within a small set of sections within the document (i.e., there is a portion of the document that the users are focused on), then the contention will increase within this portion of the document and the lock success rate would be small since only one user could edit each section.

4.9. Summary

While existing RTCES replication-based approaches offer a high rate of responsiveness to the user, the nature of the replication of the document state precipitates a need for consistency management such as OT. Unfortunately, broadcast-based approaches such as OT incurr a significant communication and computation cost. We have shown that revisiting the idea of locking is beneficial in reducing the communication and computation costs if the locks are dynamic and hierarchical. We have presented efficient algorithms for such hierarchical lock management that maximize caching of changes local to the client writer while still allowing for a reasonable level of concurrent access to the shared document. Unfortunately, as our simulation has shown, if an exclusive write policy is enacted at the atomic/leaf level within the document tree, this cost savings comes at the cost of rejecting some clients from being able to edit an already-owned section of the shared document. Thus, in the next chapter, we present our work in resolving this problem by integrating existing OT algorithms into our dynamic, hierarchical lock based approach such that all clients may write to any document section at any time, yet still retain the cost savings associated with our improved caching.

CHAPTER 5

INTEGRATION WITH OT

Having established that dynamic locking shows promise in reducing communication costs within a RTCES, we focus our attention now on solving the problem of write failures (blocking a user from editing the document if another user currently owns the section). We overcome this by adopting OT within nodes within the document tree; OT may be applied at any depth within the document tree. The algorithms for managing locks and integrating existing OT algorithms presented herein are complimentary, superior to the current best practices of existing OT algorithms over linear document representations, and significantly reduce the computational and communication costs. Further, this approach enables better intention preservation than existing OT algorithms. We achieve the performance improvement of [60] with the added improvement of avoiding bottlenecks associated with a centralized approach. As pointed out by [45], [58] and [66], the performance of OT algorithms degrades as the size of the document increases, so it is advantageous to minimize the size of the space in which OT is employed; our approach achieves this minimization by applying OT at leaf nodes within the tree and propagating these changes up the tree efficiently and allows peers to efficiently locate the peer who has the correct, up-to-date copy of the section of the document rapidly. Further, these algorithms are generalized and make no assumptions about the document's content or type and are effective on any document type – text, word processing, CAD, source code, etc.

This chapter expands our previous client-server lock management work by examining how OT may be integrated into our dynamic locking algorithms such that all users are always able to edit their copy of the document while avoiding costly global messaging. We show how our updated architecture and algorithms have been simulated using the DEVSJAVA package at the client and server, and then demonstrate the efficiencies achieved by our approach relative to existing OT algorithms. In scenarios featuring clustered editing, large document, and a large number of users, our system incurs up to 80% less communication cost than existing pure OT systems. Additionally, we discuss how our simulation design process has allowed us to simulate both client and server and then begin progress to a functional implementation of both client and server technologies – better achieving an efficient implementation of our algorithms and ideas based upon our empirical simulation results. The scalability of our approach is a significant contribution to the field in that no other RTCES has been tested with such a large number of clients (as many as 27 in our simulations).

Section 5.1 discussed how we generalize OT to handle operations on any object within the RTCES. Section 5.2 presents the validation of this approach via simulation and the progression to the realization of a prototype implementation of our models. Section 5.3 discusses related work, and Section 5.4 provides conclusions.

5.1. Generalized Operational Transformation

Similar to other CES research, we focus on text editing to demonstrate our techniques of replicating changes among peers and achieving consistency among all users; certainly our technique is applicable to other document types (CAD, graphics, objects, etc.), thus when we refer to modifying characters/strings, these could be objects.

The operations that a user may perform to change the document's content are the *Insert* and *Delete* primitives as defined in the GOT (Generic OT) algorithm [133]:

***Insert*[*S, P*]**: insert string *S* at position *P*

***Delete*[*N, P*]**: delete *N* characters started from position *P*

When representing the document as a linear string of text, P represents an index into the document. [60] extends these primitives to include the level within the document to apply the insertion/deletion – injecting the notion of context such that the string is inserted within a specific level of the document tree; one of the limitations of [60] is that the document is arbitrarily established to contain 4 levels of granularity (document tree height = 4): paragraph, sentence, word, and character.

We extend these OT primitives to be more generalized and flexible in incorporating changes made to any level within the document tree. As a result, the change is made relative to the semantic context of the change. More generally, these primitives may be expressed as:

***Insert*[*O, V, P*]**: insert object *O* within node *V* at position *P*

***Delete*[O, *V, P*]**: delete object *O* within node *V* at position *P*

Our generalized approach correctly implements GOT-defined primitives (i.e., there is a mapping from our primitives to the GOT primitives) as follows. If the document resides in a single node (as is the case of linear OT), then *V* becomes the entire document. In the case of an insert operation, *O* becomes the string to insert. Similarly, in the case of a delete operation, *O* represents the *N* characters to delete (i.e., $O = \{c_1, c_2, \ldots, c_n\}$ where c_i = the i^{th} character beginning at position *P*).

As examples of the correctness of this approach, consider that *O* could be a character being inserted into a word if node *V* represents a word; *O* could be a word being deleted if node *V* represents a sentence; *O* could be a sentence being inserted into a section if node *V* represents a section; etc. Consequently, we may employ Insert and Delete at any level within the tree to incorporate large or small changes depending upon the context. Since OT algorithms work with the *Insert* and *Delete* primitives, we may adopt any previously-defined OT algorithm into our system.

5.2. Validating the OT Integration via Simulation

To validate our approach of supporting hierarchical lock management via document trees and integrating OT into our approach, we extend our initial DEVS Java simulation [107]. In this simulation, we increase the complexity of how the lock proxy manages the subsections – using our more complex tree algorithms with OT integrated at the leaves. The overall structures of the simulation models remain consistent in that the simulation models consist of a client machine, a network, and a server machine. But since we now adopt OT at the leaf level, all write requests are satisfied so all users may concurrently edit any section within the document; the cost of such increased concurrency is that more messages are generated among the clients and the server, thus we must measure this increased communication cost and see if our approach is efficient when compared to existing pure-OT approaches.

5.2.1. The Client Model

The client machine is modeled to act as a state machine that begins outside of the document, may check out the document and becomes a reader, and then is either reading or writing to a specified section of the document [105]. When the client requests to write to a section, a lock request message is sent to the server and the server responds by notifying the client how much of the document it owns. The client is then free to move within the owned space and make changes, caching these changes locally. If the client receives a promotion or demotion message from the server, then it adjusts its ownership space accordingly and flushes its cache as needed. In contrast, if the client is sharing a section with other clients (via OT), then changes must be communicated immediately to the other clients.

The client editing behavior is determined as either *random* (the client will randomly move within different sections of the document) or *clustered* (the client's editing will be centered on a point within the document and the client will move within a small space around that point), and a *hybrid* that acts as a mix between the random and the clustered behavior. While more complex editing behavior may be modeled in future studies based upon examining log files of configuration management system repositories, these three behavior patterns demonstrate the extremes and a middle behavior that clients may exhibit.

The client model maintains a state of either writing or reading and maintains a current position in the document. The client transitions between reading and writing according to the editing behavior being simulated (see above). Messages are sent to the network via an outbound message queue, and messages are received from the network via an inbound message queue.

Additionally, we created a complex model Proxy Client Generator that allowed us to quickly create a set of client machines; this was done to make it convenient to change the client behavior configuration and create multiple clients easily, but it does not affect the simulation as this complex model does not process messages or transition states.

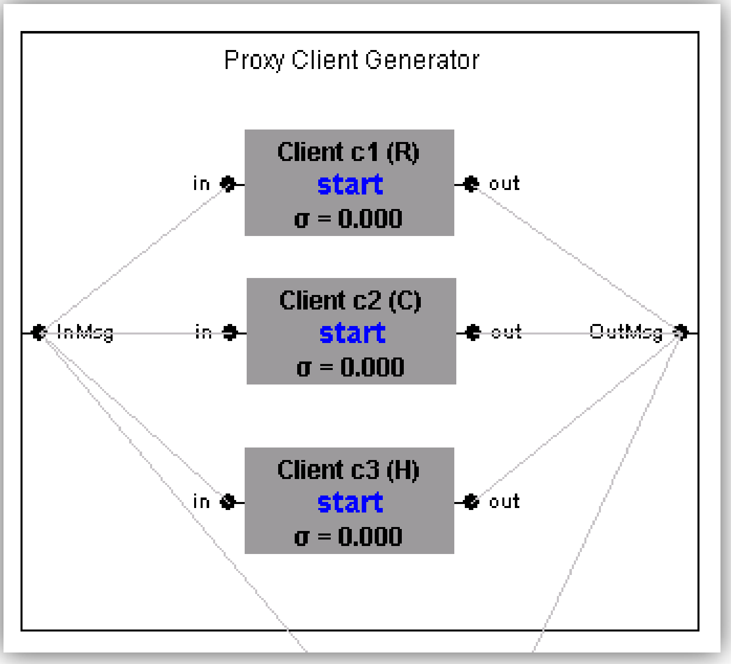


Figure 50: Modeling the Client in DEVSJAVA

5.2.2. The Server Model

The server machine is a complex model that consists of a *repository* model, a *server*, and a *lock proxy*. The *repository* is responsible for maintaining a set of documents/artifacts that can be checked in and out (similar to a standard configuration management system (CMS) like CVS or RCS). The *server* is responsible for receiving check-in and check-out requests and passing them to the repository; thus the server models a machine that would have a CMS running on it. The *lock proxy* is responsible for receiving messages from the network and parsing them to adjust the locks within the document tree. The lock proxy will only check out and check in a document if needed – thus it checks out and checks in document via proxy on behalf of the clients and keeps the server and repository ignorant that any complex management is taking place; as a result, we show how our dynamic lock management system can be added to existing repositories and easily increase their capabilities. Once checked out, the document is managed by the lock proxy and lock requests, lock releases, promotion/demotion, and OT-related messages are handled by the lock proxy and communicated to the clients.

The lock proxy model is the key model of the server machine model; this proxy model maintains the state of which documents are checked out of the server/repository models and maintains which users are present in each document and notifies clients upon promotion and demotion and passes on all OT-related messages to clients.

Additionally, we created a complex model Proxy Repository Generator that allowed us to quickly create a set of server machines; this was done to make it convenient to create multiple servers easily, but it does not affect the simulation as this complex model does not process messages or transition states.

We use a single server in this research, but our models allow for distributing the repository of documents across multiple servers as we did in [103].

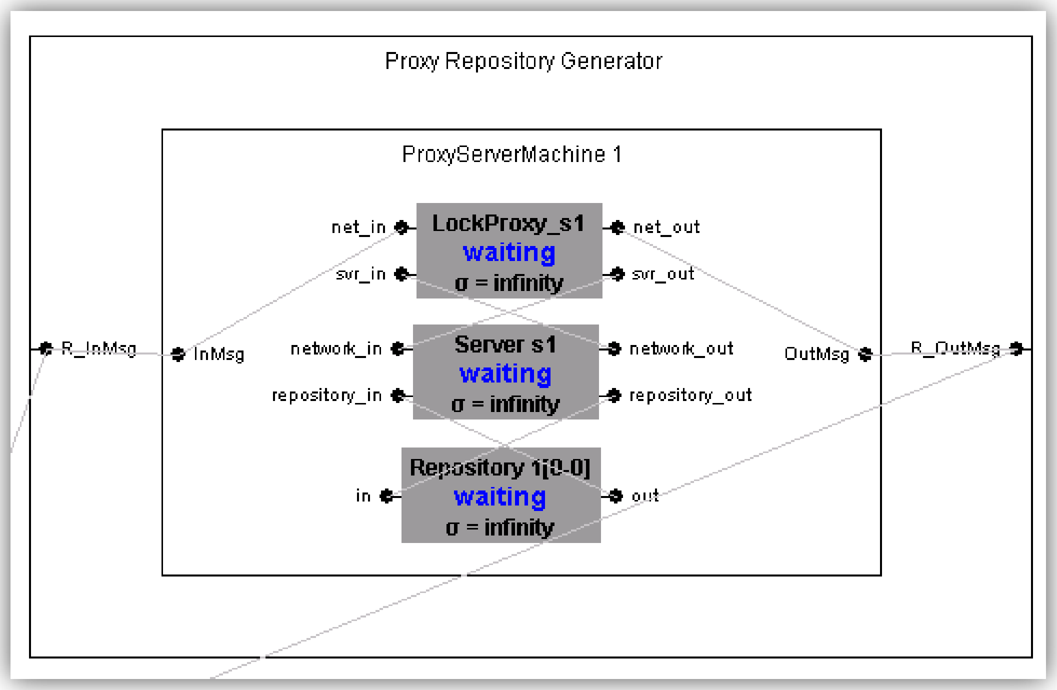


Figure 51: Modeling the Server in DEVSJAVA

5.2.3. The Network Model and Message Types

The network is modeled to receive and send messages to and from the clients and the server. All messages within the system are modeled as strings with a source, destination, and payload so that each entity within the simulation knows that the message is designated for it. For the purposes of this simulation, we assume the time to transmit a message is consistent from each client and server to all other clients and servers, but we could easily create a lookup table within the network model to adjust costs dynamically based upon sender and recipient and bandwidth congestion. But such fidelity of the network was beyond the scope and interest of this research as we were interested in the number of messages, not the real-time performance of the network, especially since the network performance can vary considerably in different RTCES scenarios. The network uses inbound and outbound message queues to receive and send messages from clients and servers.

Figure 52 shows the models running within the DEVSJAVA Simulation Viewer; in this figure, there are three clients and one server machine connected via the network model.

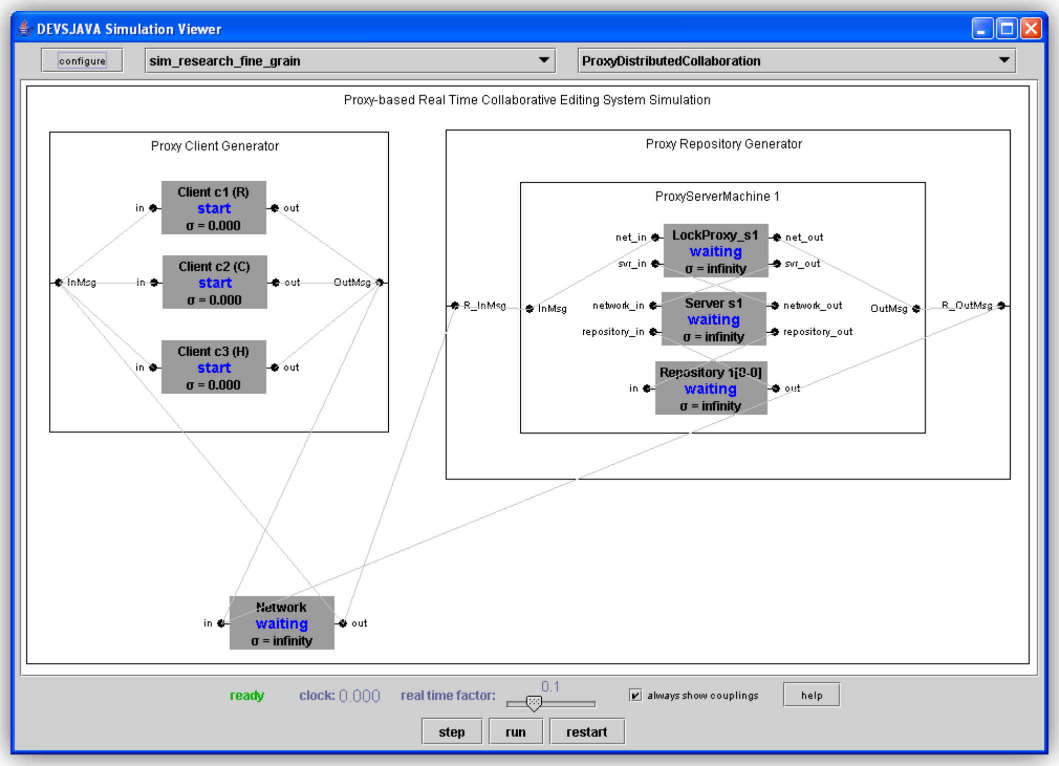


Figure 52: The Connecting Network Model in the DEVSJAVA Simulation Viewer

As the purpose of this simulation is to measure communication costs, we use the network model to capture all messages being sent to and from the clients. The following 10 message types are captured and measured within the simulation:

1. *Document Check-out* (CO) – the client would like to check out and become a reader of a document.

2. *Document Check-in* (CI) – the client is no longer interested in the document and releases it.
3. *Lock Request* (LK) – the client wants to write to a section of the document
4. *Unlock* (ULK) – the client has left the section and no longer needs the ability to write to it

In response to each of the above messages from a client to the server, the server may respond that the request succeeded or failed – for a total of eight (8) response types.

Further, since an existing client who owns a section of a document may have his lock promoted (moved up in the tree such that the client owns more of the document) or demoted (moved down in the tree such that the client owns less of the document), clients may also receive the following messages from the server indicating their new ownership status:

5. *Promotion* (P) – informs the user that he now owns more of the document that he previously owned.
6. *Demotion* (D) – informs the user that he now owns less of the document that he previously owned.

Additionally, messages must be passed to clients when a new user is added into the set of users writing to a section concurrently; these clients must perform OT among themselves to ensure CCI within the section of the document. Thus we have the following messages:

7. *OT Added* (OTA) – signals a user within a section that another user has been added to the section and future changes must be sent to this new user

8. *OT Deleted* (OTD) – signals a user within a section that a user has left the section and no longer needs to have changes sent to him
9. *OT Join* (OTJ) – tells the user requesting a lock that he has been granted write access to a section that is already using OT; this message contains a list of the existing users within the section so that the new user can send future changes to these users
10. *OT Modify (OTM)* – this message tells a client that the section has been modified and a local OT must be performed based upon the operation being communicated.

5.2.4. Results

We gathered results from 48 different runs of the simulation while modeling both the client and the server. There were six different document structures used in the simulations as shown in Table 7. Varying the structure of the document allows us to explore how varying the collaboration density (the ratio of users to leaves in the document structure) affects the messages generated in the simulation. Document structures 5 and 6 are representative of 4-page and 8-page conference papers respectively assuming the leaf nodes represent paragraphs.

Table 7: Document Structure Types

Document Structure	Number of Leaves	Maximum Depth	Average Depth
1	4	3	2.75
2	8	4	2.875
3	16	4	2.875
4	48	3	3
5	96	3	3
6	192	3	3

There were twelve configurations varying the number of clients and the document structure as shown in Table 8; for each of these twelve configurations, we ran simulations using four configurations of clients' editing behavior configurations: all random, all clustered, all hybrid, and a uniform distribution among all three types. Thus there were 48 runs of the simulation total, and each simulation ran for 10,000 iterations.

While running the simulations, all message types were recorded as the clients made lock requests, updated their states, and notified other clients editing the same section of the document as defined in Section 5.2.3. As we had previously not utilized OT at the leaf nodes, we are particularly interested in how much communication overhead is due to adding OT to our system.

Table 8: Client/Document Configurations

Simulation Configuration	Number of Clients	Document Structure
1	3	1
2	9	1
3	3	2
4	9	2
5	3	3
6	9	3
7	3	4
8	9	4
9	3	5
10	9	5
11	3	6
12	9	6

We define the percentage of messages dealing with OT out of the total messages generated to be the *Dynamic OT Rate*. As shown in Figure 53, as the collaboration density (as measured by the ration of the number of clients and the number of leaves in the document) increases, the Dynamic OT Rate increases. Since collaboration density is directly proportional to how often users will share the same space within a document, it is natural to see the messages related to OT increase as collaboration density increases.

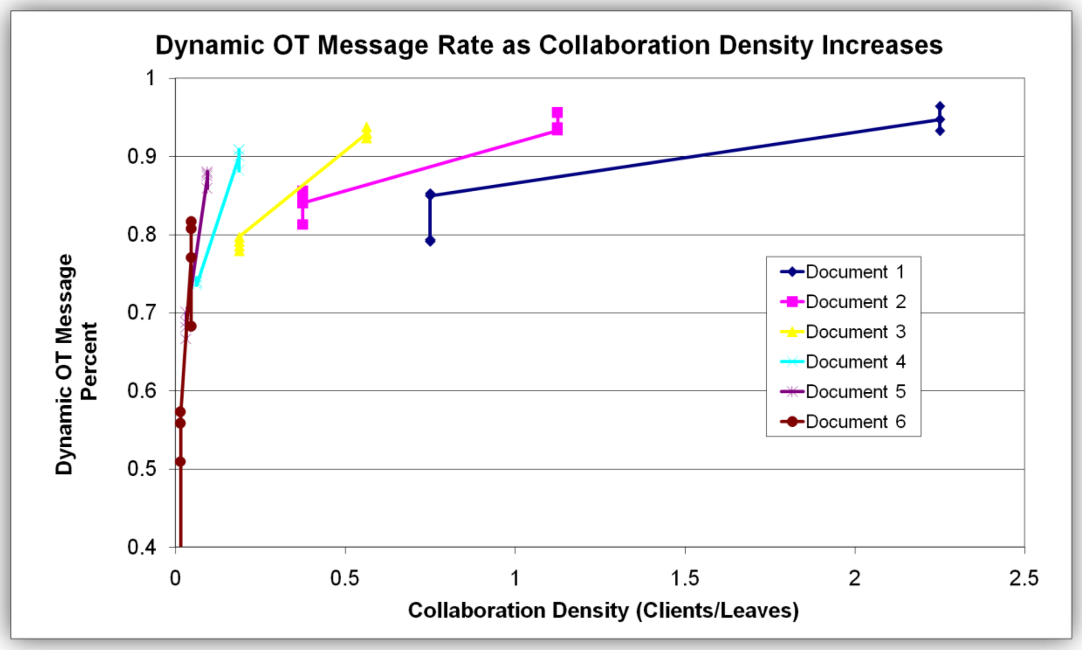


Figure 53: Dynamic Operational Transformation Cost as Collaboration Increases

Since all messages are broadcast to all users other than the originating user in a pure OT system, we define the number of messages generated in a pure OT system as

$$M_{PureOT} = (n-1)W$$

where n is the number of users and W is the number of write requests (the number of times users modified the document).

Then the relative message overhead, M_o, of our dynamic lock OT system is defined as

$$Mo = \frac{LK + ULK + P + D + OTA + OTD + OTJ + OTM}{M_{PureOT}}$$

Note that we do not consider message types LK and ULK since they are the same in our dynamic system and a pure OT system.

Thus a relative message overhead of 1 reflects the dynamic lock with OT system incurs the same number of communication cost as a pure OT system. M_o above 1 reflects our system incurs more communication that a pure OT system. M_o below 1 reflects our system incurs less communication than a pure OT system. Thus a lower value is a reduction in communication costs.

Figure 54 through Figure 57 show how our system employing dynamic locking and OT at the leaf level compares with using a "pure OT" (defined as broadcasting all changes to all users) performed with respect to communication for all 48 simulation configurations. Figure 54 shows all of the data included in Figure 55 and Figure 56 so that an overall picture can be seen of the data; Figure 55 and Figure 56 show data specific for 3 and 9 users respectively.

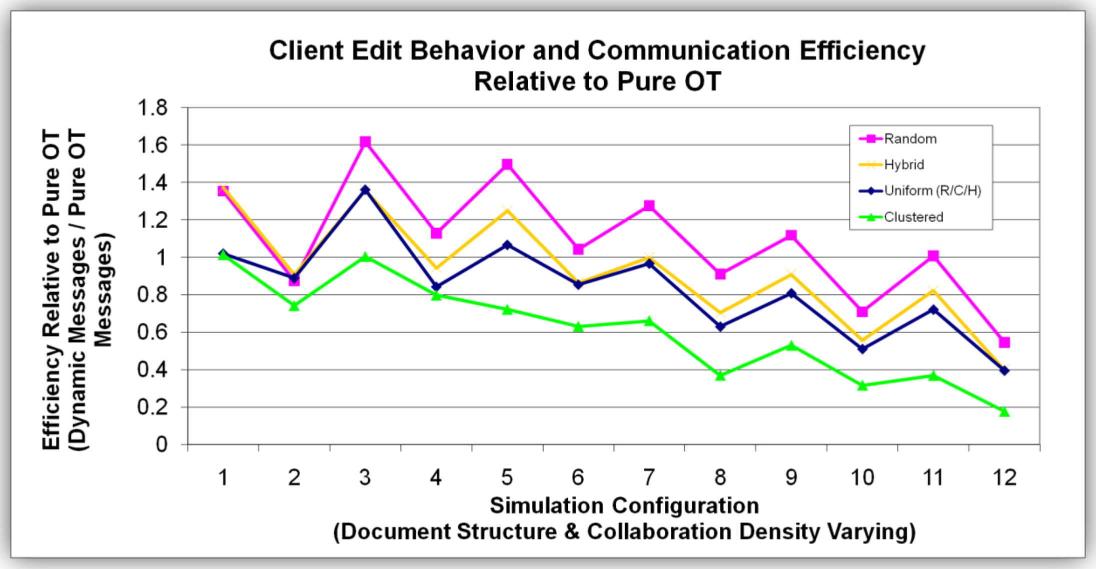


Figure 54: Edit Behaviors and Communication Efficiency

From the results presented in Figure 54, it is clear that our system performs better relative to pure OT when all other variables remain the same and the number of clients increases (note that odd-even pairs reflect an increase from 3 to 9 in the number of clients). Additionally, when clients cluster their edit behavior, our system performs better relative to pure OT; this is intuitive in that the caching benefits of our system are better utilized when edits are localized/clustered. Further, the trend in Figure 54 shows that as the size of the document increases, our system increasingly outperforms pure OT.

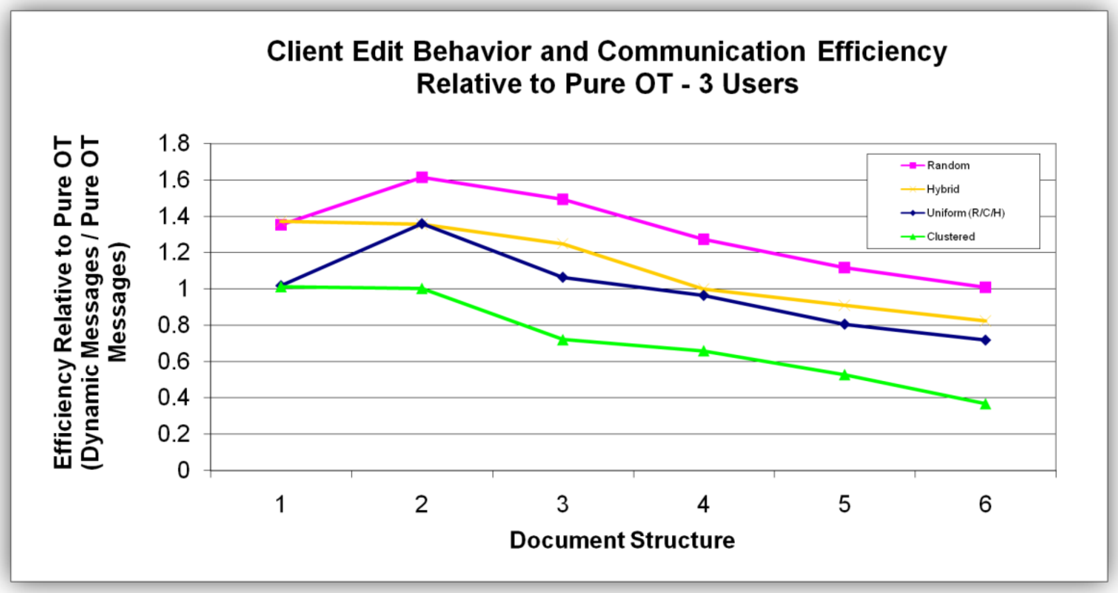


Figure 55: Edit Behavior and Communication Efficiency – 3 Users

Figure 55 shows that communication costs for our system are better than costs for an OT-only system for clustered editing behaviors, and our performance improves as the document size increases. For the random, clustered, and hybrid client series in Figure 55, there were 3 users simulated on documents 1-6. For the uniform editing behavior series, one user was simulated for each of the three different editing behaviors (random, clustered, and hybrid) for a total of 3 users. Our approach outperformed the OT-only approach for larger documents (document 6) and when clustered editing behavior was used. The data associated with Figure 55 appears in Table 9.

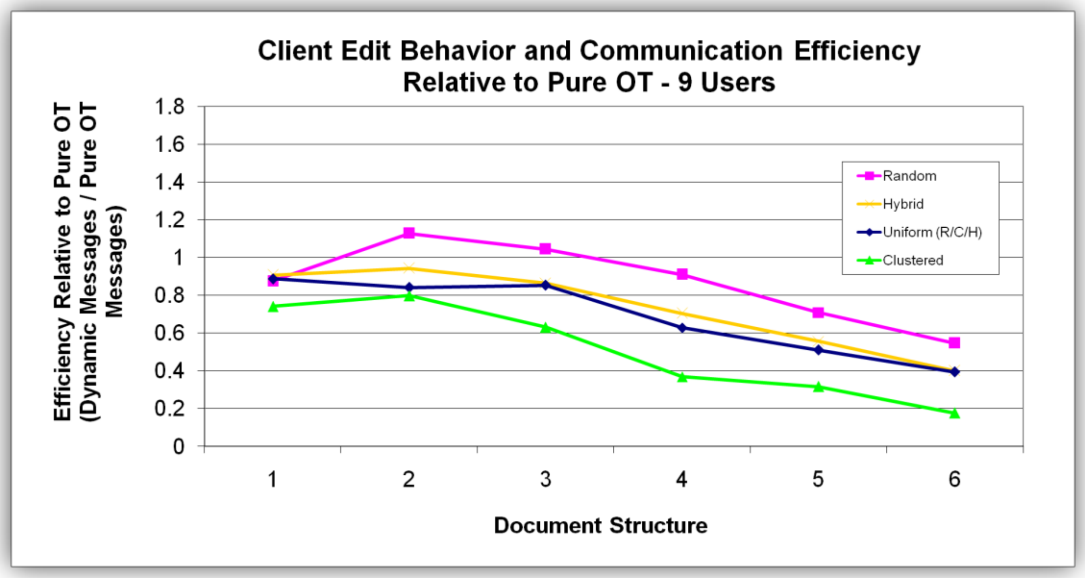


Figure 56: Edit Behavior and Communication Efficiency – 9 Users

Figure 56 shows that communication costs for our system are significantly better than costs for an OT-only system. For the random, clustered, and hybrid client series in Figure 56, there were 9 users simulated on documents 1-6. For the uniform editing behavior series, three users were simulated for each of the 3 different editing behaviors (random, clustered, and hybrid) for a total of 9 users. In all cases other than document 1-3 using random editing behavior, our approach outperformed the OT-only approach, and the trends as previously discussed of improvement increasing as document size increases and as clients adopt a clustered editing pattern continue to hold. The data associated with Figure 56 appears in Table 10.

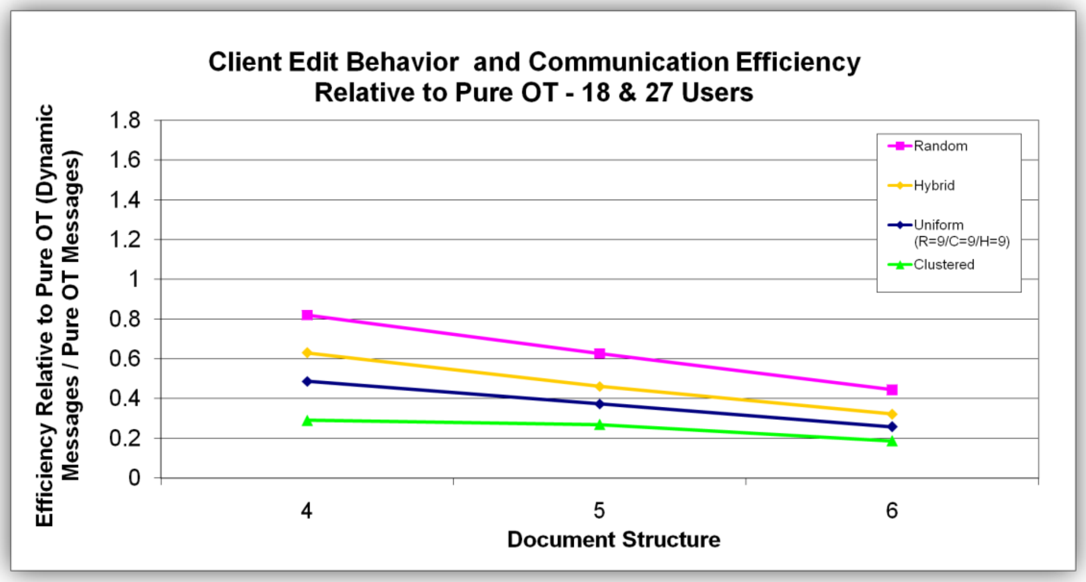


Figure 57: Edit Behavior and Communication Efficiency – 18 & 27 Users

Figure 57 shows that communication costs for our system are significantly better than costs for an OT-only system. For the random, clustered, and hybrid client series in Figure 57, there were 18 users simulated on documents 4-6. For the uniform editing behavior series, nine users were simulated for each of the three different editing behaviors (random, clustered, and hybrid). In all cases, our approach outperformed the OT-only approach, and the trends as previously discussed of improvement increasing as document size increases and as clients adopt a clustered editing pattern continue to hold. The data associated with Figure 57 appears in Table 11.

Table 9: Simulation Results – Communication Costs with Structures 1-3

Doc ID	Clients			Write requests	Messages									TOTALS		OT%	MO
	R	C	H		LK	ULK	U	D	P	OTA	OTJ	OTD	OTM	DM	POT		
1	1	1	1	1458	254	241	5	62	59	257	179	194	1719	2970	2916	0.79	1.02
1	3	0	0	955	249	233	5	25	22	311	202	275	1262	2584	1910	0.79	1.35
1	0	3	0	2243	290	276	1	52	50	362	246	301	2969	4547	4486	0.85	1.01
1	0	0	3	864	170	156	4	16	11	243	140	222	1411	2373	1728	0.85	1.37
1	3	3	3	7904	1540	1388	30	7	4	6977	1512	6477	38156	56091	63232	0.95	0.89
1	9	0	0	6615	1646	1422	47	6	3	6645	1594	5918	29020	46301	52920	0.93	0.87
1	0	9	0	8286	909	828	11	8	5	3960	893	3680	38836	49130	66288	0.96	0.74
1	0	0	9	8769	1739	1580	28	10	7	7887	1699	7358	43323	63631	70152	0.95	0.91
2	1	1	1	2070	434	404	5	15	11	610	363	575	3210	5627	4140	0.85	1.36
2	3	0	0	1645	485	456	9	24	20	696	432	654	2541	5317	3290	0.81	1.62
2	0	3	0	2856	385	362	13	37	33	446	315	407	3737	5735	5712	0.86	1.00
2	0	0	3	2049	444	420	7	11	8	615	379	588	3090	5562	4098	0.84	1.36
2	3	3	3	7617	1755	1609	22	14	12	6867	1669	6389	32905	51242	60936	0.93	0.84
2	9	0	0	4557	1370	1276	8	6	3	6704	1311	6360	24070	41108	36456	0.94	1.13
2	0	9	0	7941	1149	1053	15	12	9	5020	1105	4696	37581	50640	63528	0.96	0.80
2	0	0	9	7000	1709	1585	17	6	3	7433	1633	7025	33411	52822	56000	0.94	0.94
3	1	1	1	2942	636	589	9	41	33	651	443	593	3269	6264	5884	0.79	1.06
3	3	0	0	1936	600	554	8	44	37	740	452	691	2660	5786	3872	0.79	1.49
3	0	3	0	3706	539	497	38	58	49	429	333	381	3024	5348	7412	0.78	0.72
3	0	0	3	2746	637	596	17	75	68	750	496	687	3538	6864	5492	0.80	1.25
3	3	3	3	6864	1716	1553	17	18	10	6778	1606	6260	28856	46814	54912	0.93	0.85
3	9	0	0	5817	1888	1724	14	51	43	8304	1761	7737	27053	48575	46536	0.92	1.04
3	0	9	0	10276	1651	1494	28	41	33	5215	1468	4738	37158	51826	82208	0.94	0.63
3	0	0	9	7577	1872	1700	17	35	25	7397	1759	6824	32671	52300	60616	0.93	0.86

Table 10: Simulation Results – Communication Costs with Structures 4-6

Doc ID	Clients			Write requests	Messages									TOTALS		OT%	MO
	R	C	H		LK	ULK	U	D	P	OTA	OTJ	OTD	OTM	DM	POT		
4	1	1	1	4042	959	859	19	85	70	754	578	671	3806	7801	8084	0.74	0.96
4	3	0	0	2887	960	855	10	59	44	925	666	839	3000	7358	5774	0.74	1.27
4	0	3	0	5489	816	739	118	116	99	547	413	487	3919	7254	10978	0.74	0.66
4	0	0	3	3764	946	849	15	99	83	773	575	691	3493	7524	7528	0.74	1.00
4	3	3	3	8905	2318	2055	28	48	33	6186	2086	5585	26440	44779	71240	0.90	0.63
4	9	0	0	7507	2597	2298	5	52	40	9165	2354	8314	29818	54643	60056	0.91	0.91
4	0	9	0	10850	1841	1656	62	124	107	2990	1457	2693	21092	32022	86800	0.88	0.37
4	0	0	9	9655	2582	2287	29	58	41	7529	2282	6772	32818	54398	77240	0.91	0.70
5	1	1	1	4638	1053	910	59	125	95	669	539	578	3458	7486	9276	0.70	0.81
5	3	0	0	3665	1241	1078	11	159	126	901	693	784	3198	8191	7330	0.68	1.12
5	0	3	0	5473	797	711	194	127	103	405	333	359	2765	5794	10946	0.67	0.53
5	0	0	3	4429	1179	1031	26	160	126	774	603	667	3490	8056	8858	0.69	0.91
5	3	3	3	10308	2698	2325	44	114	84	5371	2141	4675	24558	42010	82464	0.87	0.51
5	9	0	0	8873	3166	2712	9	128	100	8019	2740	6964	26498	50336	70984	0.88	0.71
5	0	9	0	11766	1942	1719	194	184	155	2779	1313	2445	18993	29724	94128	0.86	0.32
5	0	0	9	11154	3024	2598	36	152	120	6495	2526	5612	29134	49697	89232	0.88	0.56
6	1	1	1	4879	1185	1015	680	307	256	439	402	355	2383	7022	9758	0.51	0.72
6	3	0	0	4080	1425	1190	556	259	204	706	592	557	2734	8223	8160	0.56	1.01
6	0	3	0	5494	653	574	1187	193	166	140	140	115	884	4052	10988	0.32	0.37
6	0	0	3	5081	1350	1147	471	332	273	640	540	497	3113	8363	10162	0.57	0.82
6	3	3	3	10300	2830	2357	1803	270	213	3557	1916	2950	16594	32490	82400	0.77	0.39
6	9	0	0	10142	3608	2985	1026	284	228	6195	2729	5175	22052	44282	81136	0.82	0.55
6	0	9	0	13607	1647	1452	2606	207	167	1442	910	1255	9448	19134	108856	0.68	0.18
6	0	0	9	12194	3329	2800	912	262	204	4502	2314	3772	20826	38921	97552	0.81	0.40

Table 11: Simulation Results – Communication Costs with 18 and 27 Users

Doc ID	Clients R	Clients C	Clients H	Write requests	Messages LK	ULK	U	D	P	OTA	OTJ	OTD	OTM	TOTALS DM	POT	OT%	MO
4	18	0	0	13721	4859	4228	9	54	38	35085	4624	31416	111012	191325	233257	0.95	0.82
5	18	0	0	16545	5913	5015	10	115	84	30905	5479	26795	102049	176365	281265	0.94	0.63
6	18	0	0	18677	6709	5508	9	213	160	23236	5929	19414	79898	141076	317509	0.91	0.44
4	0	18	0	20712	3599	3190	94	67	51	10564	3131	9449	72150	102295	352104	0.93	0.29
5	0	18	0	21520	3706	3242	200	120	90	10024	3049	8804	69137	98372	365840	0.93	0.27
6	0	18	0	20098	3321	2929	313	268	223	6373	2464	5689	42411	63991	341666	0.89	0.19
4	0	0	18	17888	4805	4219	25	43	26	28289	4525	25333	124492	191757	304096	0.95	0.63
5	0	0	18	20387	5554	4716	39	119	86	22733	4981	19517	102422	160167	346579	0.93	0.46
6	0	0	18	22004	6164	5142	63	246	190	16434	5179	13786	73302	120506	374068	0.90	0.32
4	9	9	9	26398	6812	5889	67	41	26	49195	6485	43382	222741	334638	686348	0.96	0.49
5	9	9	9	28363	7493	6384	73	117	85	39877	6943	34469	180275	275716	737438	0.95	0.37
6	9	9	9	29303	7933	6655	72	222	167	27577	6869	23497	123515	196507	761878	0.92	0.26

Doc ID – Document Structure ID

Clients R – Random
Clients C – Clustered
Clients H – Hybrid

Write Requests - # times clients modified document

LK – Lock Request
ULK – Unlock (Lock Release)
U – Update Position
D – Demotion
P – Promotion

OTA – OT Add
OTJ – OT Join
OTD – OT Delete
OTM – OT Modify

DM – Messages using Dynamic Locking Algorithm
POT – Messages using Pure OT Algorithm

OT% = (OTA + OTJ + OTD + OTM) / DM

MO – Relative message overhead (DM / POT)

5.3. Discussion and Related Work

While there is much literature on OT research such as [66],[134], etc., but the prior work assumes that the document structure is linear in nature and operates exclusively on character-level insertion and deletion operations. Prior OT research supporting rich-text document formats (thus supporting objects) claims that their approach is generalizable to other non-character insertion and deletions, but all such OT researchers describe their algorithms in terms of character insert and delete operations; few discuss the details of supporting other semantic levels of operations. Those that do support non-linear OT algorithms enforce strict semantic levels and are not flexible to arbitrary document structures or depths of document trees. For example, [58] discusses algorithms for merging two different versions of a document by accepting changes/operations at a word, sentence, or paragraph depth/level; this constraint of only applying operations at specified levels within the semantic structure is not as broad and flexible as our generalized approach as presented herein. [57] also demonstrate promise in managing history buffers in a hierarchical document structure and applying operations at varying semantic levels within the document; but again the semantic depth at which the changes are managed are constrained to paragraph, sentence, and word levels. Further, their approach applies operations from top to bottom, so all operations must flow through the document tree root – posing a significant bottleneck in processing the operations. Rather, our approach is flexible in supporting operations at any semantic depth and begins the process of managing and applying these operations within the leaf nodes where they occur.

Other research has supported XML/HTML type structures in the RTCES [20] and [59] are notable contributions that allow for editing of structured content; but while these systems employ semantic knowledge of what an XML element, attribute, their operations remain rigid relative to specific types of content being modified and are not generalizable to any object. Additionally, the Draw-Together [56] and other graphics editing systems have shown promise in managing graphical objects and applying conflict resolution (OT) for groups of objects. This work is particularly interesting in that it allows for any set of objects within the shared document to be grouped together and resolves overlapping sets as defined by different users; for example, if user U_1 selects objects O_1, O_2, and O_3 and performs Op_1 on them, while user U_2 selects objects U_2, U_3, and U_4 and performs Op_2 on them, the algorithms Draw-Together correctly applies the operations such that the replicas at U_1 and U_2 converge. While the history buffer maintained in the Draw-Together algorithm is maintained globally for the entire document (rather than hierarchically) this research is interesting to and relates to our research in that it shows that grouping of objects at any arbitrary time is possible, and further it is possible to achieve CCI after performing concurrent operations on overlapping groups/sets.

The ability to publish some sections of a collaborative environment and keep some sections of a collaborative environment private is discussed in [125]. While this is similar to our caching of locally-performed operations, Souza's work is more akin to having a "sandbox" where local changes can be applied for testing out ideas before publishing them to the shared space – similar to traditional CMS that allow users local copies of a shared document in an asynchronous fashion.

5.4. Summary

In this chapter we have demonstrated that our dynamic, hierarchical locking approach may successfully integrate existing OT algorithms to allow all clients the opportunity to write to any section of the shared document, thus resolving the problem of exclusive write locks as previously presented in Chapter 4. The improved algorithms presented in this chapter demonstrate that localized OT among a smaller subset of the total clients can reduce the communication costs dramatically – achieving a significant decrease in messages sent among clients; our simulation results demonstrate communication costs savings as much as 80%, and show that such improvement are achieved over an OT-only approach as the number of clients increases, the number of sections in the document increases, and when clustered editing behavior is exhibited by the clients. This chapter has demonstrated that the scalability of our approach is a significant contribution to the field in that no other RTCES has been tested with such a large number of clients. While the results presented in the chapter are significant, we recognize that the client-server model used in our approach results in a potential bottleneck at the server. As a result, we extend our approach to support peer-to-peer communication among the users and remove the bottleneck and single point of failure of the server. These P2P extensions are presented in the next chapter.

CHAPTER 6

PEER-TO-PEER DOCUMENT MANAGEMENT

We extend our previous work on centralized document trees by distributing the lock management among all peers within the CES and allowing the cached changes (history buffers) to be applied at an arbitrary level in the hierarchical document tree. These p2p algorithms for managing locks and distributing existing OT algorithms are complimentary, superior to the current best practices of existing OT algorithms over linear document representations, and significantly reduce the computational and communication costs. Further, this approach enables better intention preservation than existing OT algorithms. We achieve the performance improvement of [60] with the added improvement of avoiding bottlenecks associated with a centralized approach. As pointed out by [45], [58] and [66], the performance of OT algorithms degrades as the size of the document increases, so it is advantageous to minimize the size of the space in which OT is employed; our approach achieves this minimization by applying OT at leaf nodes within the tree and propagating these changes up the tree efficiently and allows peers to efficiently locate the peer who has the correct, up-to-date copy of the section of the document rapidly. Further, these algorithms are generalized and make no assumptions about the document's content or type and are effective on any document type – text, word processing, CAD, source code, etc.

This chapter begins by discussing the central issues of moving from a client-server to a P2P architecture and how our algorithms must be modified to support the new P2P approach. We present the modified lock request algorithm in Section 6.2, how to handle the user modifications to content and structure of the document tree in Section

6.3, the modified lock release algorithm in Section 6.4, and how users may move within the document in Section 6.5. A discussion of the correctness and efficiency analysis is presented in Section 6.6. We then discuss the new problem of locating the peer with which to communicate in Section 6.7, and present the benefits of replication, congestion avoidance, and fault tolerance in Section 6.8. Finally, we conclude with a summary in Section 6.10.

6.1. Extending the Client-Server Algorithms

In the client-server architecture, all messages related to lock request and release, promotion and demotion, and OT join, OT delete, and OT add had to pass to or from the server; this creates a centralized point of failure and bottleneck with respect to message processing at the single server. In contrast, a P2P approach may allow each peer to manage a section of the document tree such that messages (lock request, release, etc.) pertaining to that portion of the document tree may be handled by that peer while other peers handle messages pertaining to other portions of the document tree. But in moving from a client-server to a P2P architecture, we introduce complexity and new problems to be solved. First, how must our previous client-server lock request and release algorithms be modified to allow for peers to manage sub-trees within the overall document tree (i.e., what algorithmic changes must be made with regard to successfully manage the locks on the document tree)? Second, how can we correctly and efficiently locate which peer manages the section of the document tree a requesting users is interested in; since now there is no centralized server to query, before a lock request can be made, we must locate the peer to which to make the request. Third, how may the P2P approach improve the scalability (via load balancing and reduction in message congestion) and improve fault tolerance (via replication of portions of the document tree structure and content among various peers).

Further, given the peer-to-peer nature of this approach, we adopt an adjustable locking policy that is established on a per-section basis. As a result, users may select whether to share their active section and allow multiple writers (thus adopting OT or some other coordination mechanism), choose to disallow other users from entering their owned section (denying the lock request of other writers wishing to enter the section), or allow for demotion of their lock to a sub-section to resolve the conflict. The policy adopted may vary according to any user (i.e., one user may select a sharing policy while another selects an exclusive lock policy while another selects a demotion policy) and also very according to which section is active (i.e., a user might adopt an exclusive lock policy when editing section X, but the same user might adopt a sharing policy when editing section Y). Of course, global policies based upon user priority, etc. can also be adopted to “trump” local policies if desired (such that a high-priority user can override the lock policies of another lower-priority user if desired/needed). Thus the P2P algorithms discussed in this chapter assume such lock policies are on a per-node basis and are queried at each node upon a lock request or release.

The preceding client-server approach taken for lock management is shown in Figure 58 where the server is a central bottleneck and point of failure. The entire document tree is managed by the server. In this figure, local OT is being applied among users 3 and 4, but other than this, all communication is handled via the server.

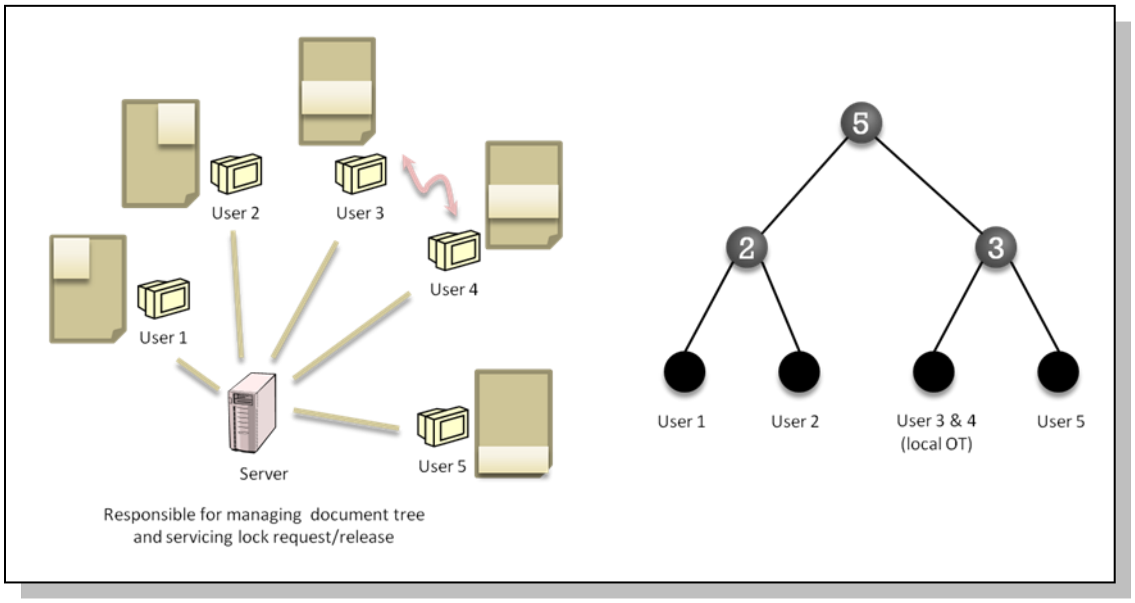


Figure 58: The Client-Server Lock Management Model

shown in Figure 59. Notice in the P2P model, each user is responsible for managing the portion of the document tree that is associated with the portion of the document that they are editing and each peer is able to communicate directly with all other peers in the system. Local OT is still permissible as demonstrated in the sharing and OT among users 3 and 4.

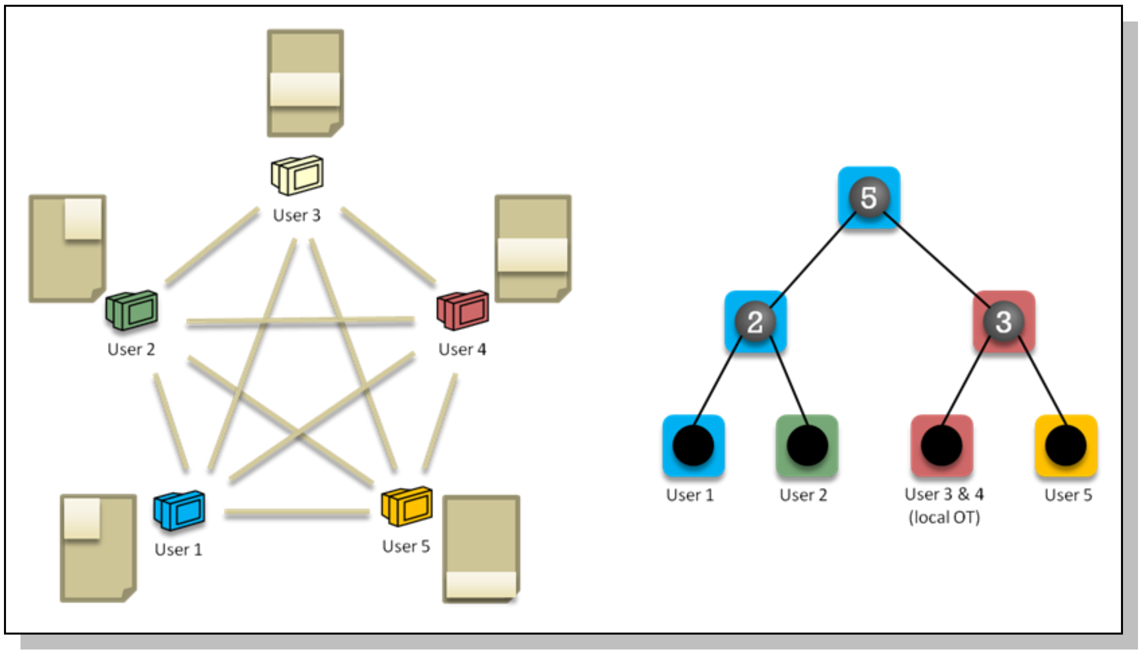


Figure 59: The P2P Lock Management Model

6.2. Lock Request

When a user, U_1, enters/initiates the CES, this user is the only user in the system and consequently has the entire document updated and cached in its computer. Assuming a locking policy has been adopted and sharing is not permitted, when another user, U_2, enters the system, U_1's portion of the document is reduced to accommodate the new user such that the contention between U_1 and U_2 is removed. We assume that U_1 and U_2 are interested in authoring disparate sections; if U_1 and U_2 are interested in editing the same section of the document, then either U_2's request to enter the section "owned" by U_1 can be rejected (a failed write event) or an OT-based multi-writer policy may be adopted. Figure 60 demonstrates the demotion of U_1 from the entire section v down to the sections denoted by $\{w_1, \ldots, w_n\}$ and the injection of U_2 at the section denoted by x. Any changes made so far by U_1 to x (denoted by Δx) must be passed to

U_2. At this point, U_1 contains the most current copy of the sections $\{w_1, \ldots, w_n\}$, and U_2 contains the most current copy of section x. Since the Δx is being transmitted to U_2, it is appropriate to apply reduction to the history buffer at x when such a demotion occurs; since these nodes are locked by U_1, we avoid any form of deadlock in achieving the messaging to U_2.

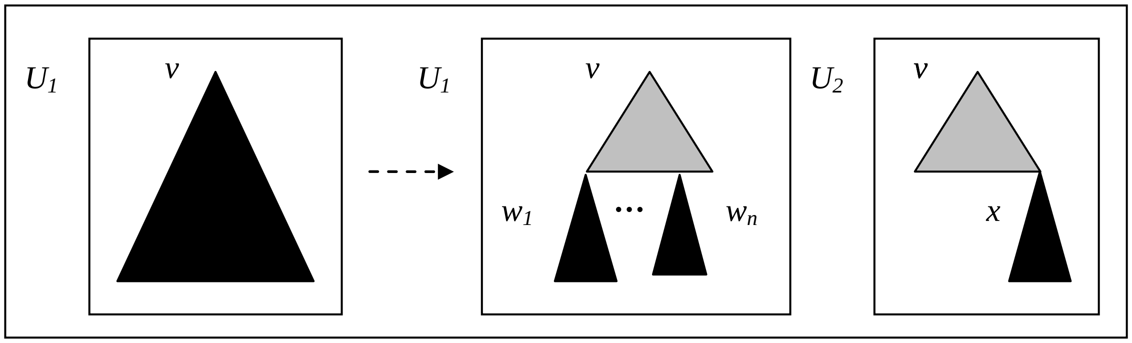


Figure 60: Peer-to-Peer Lock Request

A user requests a section of the document to which he wants to write, and the system attempts to obtain a lock on that section of the document. The OBTAINLOCK algorithm works from top-to-bottom by examining nodes in the path from the root to the destination node. The correct path is determined by first querying the peer who manages the root, and then descending further down by following peers' references to other peers (see Section 6.7). 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 either adopt an OT strategy if multiple writers are allowed at this node, or we *demote* (push down) this black node (its current owner/user), turn this node into grey thus making room for the new insert request to continue down. Demotion works by moving the ownership of that user (and the black coloring) down the tree hierarchy while ensuring that the leaf node needed by that user is contained within the sub-hierarchy. As in our previous, centralized algorithms [103][105], we avoid deadlock among peers by employing handshake locks on parent/child nodes and by always moving downward through the tree.

6.3. Editing Content and Modifying the Structure of the Tree

Given the structure of the document tree, all content is stored at leaf nodes; all other nodes act as structural support and represent sections and subsections. When a user U_1 owns a section denoted by node v, then all changes made to the content of the sections rooted at v are cached locally on U_1. Four types of edits/changes may be made within the system by a user U_1:

1. The content of a leaf v may be changed. In this case, U_1 modifies some element of the document that is represented by v. No structure change is made to the tree.
2. U_1 removes/deletes a node v. In this case, node v may be either a leaf node or a non-leaf node. If v is a leaf node, then the entity/content that v stored is deleted from the tree. If v is a non-leaf node, then v and all of its child nodes are removed from the tree (denoting a removal of a section and all its subsections). In this case, it is valid to remove all sub-trees since by definition U_1 has write permissions to node v or the change would be rejected.
3. U_1 splits a node v into two nodes, v and v_2. In this case, U_1 is creating a new section, paragraph, etc. v_2 is added as a sibling to v, some of the content of the original v is moved to v_2, and U_1 owns both v and v_2.

4. U_1 creates a new section. This is a modified case of the case 3 in that the new node v_2 is created, except in this case no content is moved from an existing node. The node v_2 is added into the tree and is owned by U_1.

In the above cases, no communication is needed between peers – all of the changes are cached locally. If other users are interested in the sections rooted at *v* (as either readers or writers), then any changes made can be selectively multicast to these other users and an OT can be employed to maintain consistency among all peers interested in sections rooted at *v*.

6.4. Lock Release

The REMOVELOCK algorithm also works from top-to-bottom. As the path from the root to the node to be released is traversed downward, the grey-count for all nodes painted grey is decreased by one until a grey node with a grey-count of one (after decrementing) is encountered; when this occurs, a *promotion* is needed to ensure that the sibling of the to-be-unlocked node owns the largest sub-tree possible. When a REMOVELOCK request is fulfilled that necessitates a promotion, the node whose grey count has been reduced to one must be painted black and must be added into the black sibling list of the grey, parent node. Since this algorithm works strictly downward along the tree, we avoid deadlock and are guaranteed to be able to promote the lock if only one peer remains in the sub-tree.

When a user, U_1, leaves a section *w* of the document and does not plan to return (or does not plan to return in the near future), it is appropriate to release the lock held by U_1 on *w* and promote (if possible) another user's (U_2) lock such that the portion of the document held by U_2 is increased. Since U_1 is leaving *w*, there is no contention on *w* with other users, so if there remains only one user, this user can assume ownership of a larger portion of the document. Alternatively, it is possible to cache the changes on U_1 and update U_2's ownership at a later time (if at all). This would be appropriate in the case where it is foreseeable that U_1 would return to w before any other user desires to read/write to w.

Let Δw = changes made by U_1 on w. In the case where Δw is being communicated from U_1 to U_2, we guarantee that Δw represents all changes to w and U_1's copy of w is up-to-date (i.e., Δw = the history buffer of w at U_1). Consequently, we must communicate Δw to another user U_2 and replay Δw on U_2's copy of w to achieve the up-to-date version of w at U_2. This is shown in Figure 61. In this example, U_2's ownership is being promoted from x to v. As a result, only Δw needs to be communicated, and we avoid having to communicate the entire contents of w to U_2. x is current since U_2 owns it, and w is now current because Δw has been "replayed" at U_2. Thus U_2 contains a proper and complete, up-to-date version of v since v is defined by w and x (i.e., v is current because $v = w + x$ and $\Delta v = \Delta w + \Delta x$). Note that Δv is easily constructed in constant time since Δw and Δx are independent and do not conflict – thus Δv is the concatenation/simple-merge of Δw and Δx.

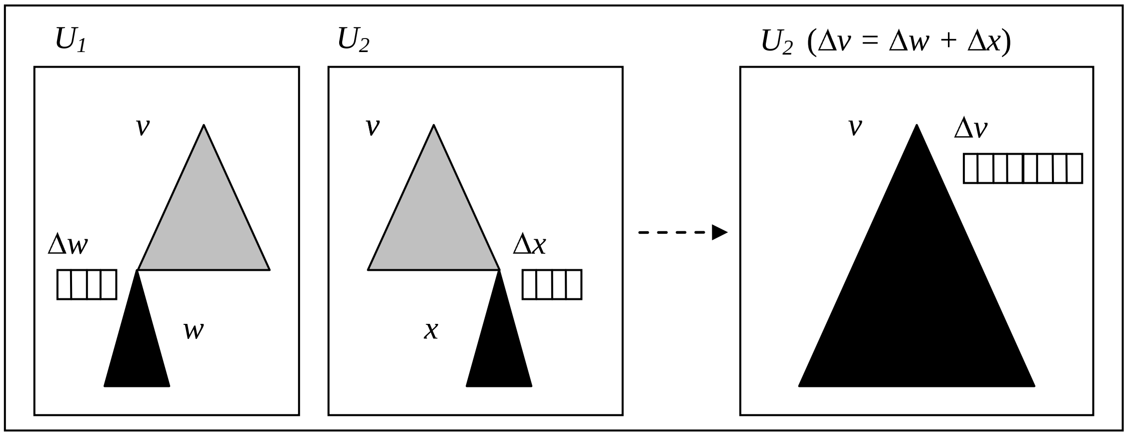


Figure 61: Peer-to-Peer Lock Release

Since the Δw is being transmitted to U_2, it is appropriate to reduce the history buffer at *w* before such a promotion occurs; even though we are moving up the tree, we avoid deadlock in achieving the promotion and messaging to U_2 by using a window lock on *v*, *w*, and *x*. Reduction may be applied safely and recursively up to v. Here, when we state we are "moving up" in the tree, this is logically up; all operations are performed top-to-bottom using handshake locks and deadlock is avoided.

When Δw is communicated to U_2, U_2 may elect to incorporate Δw into its copy of *w*, or if desired, U_1's changes to *w* (Δw) may be rejected. This acceptance or rejection of changes by other users could be done automatically by the system based upon embedded rules or done explicitly by users as prompted by the system.

When a user, U_1, leaves the CES, all of the cached changes are flushed to another user within the system. The policy of flushing the cache could be set to broadcast the changes to all peers or send the changes to a single peer (or selectively send specific sections' changes to various peers) who would assume ownership of the sections that U_1 had previously owned.

6.5. User Movement within the Document Tree

If user U_i is currently editing/present in the section denoted by node v and wishes to move to the section denoted by node x, then three situations may arise (see Figure 62):

1. U_i owns x; this may arise for two reasons: either U_i owns (i.e., has a lock on) a node n that is an ancestor of nodes x and v, or the common ancestor n may be marked grey because U_i owns x and v but another user, U_j, owns a node within the n-rooted tree. In this case, we move U_i to x without any contention with other users. U_i can retain the lock on v or release it (user preference), and no communication is necessary.
2. x is not owned (i.e., colored white). If this is the case, then either U_i can release its lock on v and acquire the lock on x, or, if desired, U_i can retain its lock on v and acquire the lock on x (this would be desirable if U_i was entering x temporarily and knew a priori that he wished to return to v after a brief edit to x). In this situation, there must exist another user, U_j, that owns another node w rooted at n since U_i does not own n (case 1); thus n must be colored grey.
3. Another user U_j owns x (or owns a tree which contains x); again, n must be grey due to the contention between U_i and U_j (and possibly other users). If this is the case, then U_i must wait for U_j to leave x and release the lock on x – assuming a single-writer policy is employed at x. Alternatively, if a multi-writer policy has been adopted at x (i.e., U_j allows other writers within x), then U_i may enter x and an OT-based coordination policy is adopted among the writers.

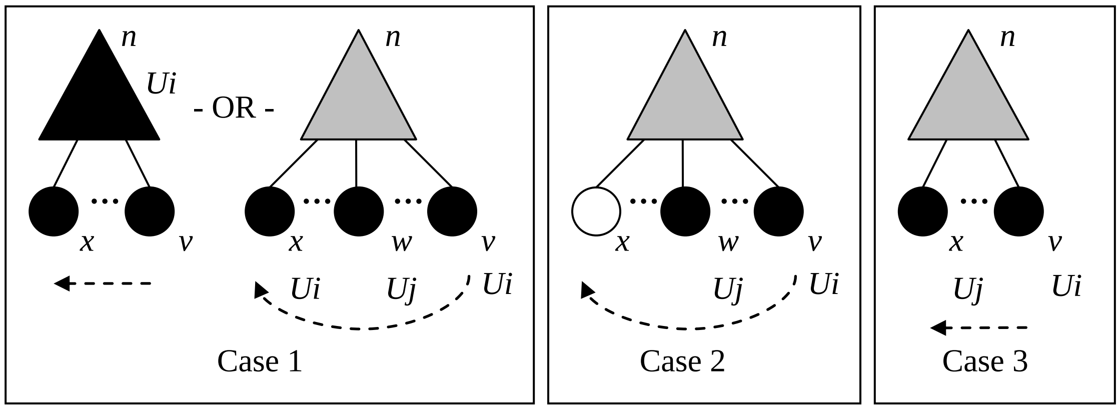


Figure 62: Three Cases of a User Moving from *v* to *u*

In cases 1 and 2, no communication is required if the user retains his lock on node *v*; in case 1 the user is moving within the user's currently-owned sub-tree and the move is permissible and does not conflict with any other user; in case 2 the user is moving to a white node which implies that no other user was previously in this desired node. In case 3, the history buffer at node *x* must be communicated to user U_i since U_i now has entered *x* and must have the latest state of *x*.

If the user elects to release his lock on node *v*, then the cache (history buffer) for node *v* will be flushed and communicated to the node that assumes management of *v* (which could be the original owner U_i if no promotion occurs in which case no communication is required; otherwise, the new manager of *v* will be node promoted as a result of U_i leaving *v* and the history buffer (cache) of *v* must be communicated to the promoted node).

6.6. Correctness and Efficiency Analysis

Similar to the client-server algorithms for lock management, we designed the P2P versions of the OBTAINLOCK and RELEASELOCK operations such that the document tree is accessed only in a top-to-bottom, pipelined fashion; we do this to avoid race conditions. We enforce the policy that nodes must be accessed in a top-down manner. As a result, our P2P operations may also be executed concurrently while maintaining their correctness.

The full presentation of the algorithms appears below in Figure 63 through Figure 65. Note that these algorithms are presented to show intent; the actual implementations feature an iterative/loop-based solution that employs a top-to-bottom, handshake-lock as the paths from the root to the desired nodes are traversed. These P2P algorithms are nearly identical to their client-server counterparts except that the communication of the history buffers (cache) and the reductions are now included.

Herein, we present the algorithms and a discussion of their associated costs in detail.

```
OBTAINLOCK($w$, $u_i$)
        if $w.owner \neq u_i$
    RECURSEOBTAINLOCK (ROOT, $w$, $u_i$)

RECURSEOBTAINLOCK($n$, $w$, $u_i$)
        if $n.color$ = white
            // destination reached and lock is permissible
            then   SETLOCK($n$, $u_i$, $w$)
                   LINKSIBLINGS($n.parent$, $n$, $n.parent.firstBlackChild$)
        else if $n$ ISATOMIC or ($n.color$ = black and not OTENABLED($n$))
            // lock failure, so undo grey-count inflation
            then   RECURSEREMOVELOCK (ROOT, $w$, $u_i$)
                   return failure
        else if ($n$ ISATOMIC or $n.color$ = black) and OTENABLED($n$)
            // lock sharing is permissible, so join and apply OT
            then   $\Delta n'$ = REDUCE($\Delta n$)
                   COMMUNICATE($\Delta n'$, $u_i$)
                   replay $\Delta n'$ at $u_i$'s copy of $n$
                   add $u_i$ to $n$'s distribution engine
        else if $n.color$ = grey
        // conflict/destination further in path, so determine next peer to
        // communicate with and proceed further down the tree
            then   $n.greyCount$ = $n.greyCount$ + 1
                   RECURSEOBTAINLOCK(NEXTINPATH($n$, $w$), $w$, $u_i$)
        else       // demotion occurs at a black node
                   $b$ = NEXTINPATH($n$, $w$)
                   $a$ = NEXTINPATH($n$, $n.originalRequest$)
                   REMOVEFROMSIBLINGLIST($n$)
                   SETLOCK($a$, $n.owner$, $n.originalRequest$)
                   $n.color$ = grey
                   $n.greyCount$ = 2
                   update distribution engine subscription for nodes $a$ and $b$
                   if $a \neq b$
                       // conflict resolved, so communicate $\Delta w$ to $u_i$
                       then   SETLOCK($b$, $u_i$, $w$)
                              LINKSIBLINGS($n$, $a$, $b$)
                              $\Delta w'$ = REDUCE($\Delta w$)
                              COMMUNICATE($\Delta w'$, $u_i$)
                              replay $\Delta w'$ at $u_i$'s copy of $w$
                       else   // keep looking further down the tree to remove conflict
                              RECURSEOBTAINLOCK($a$, $w$, $u_i$)
```

Figure 63: P2P OBTAINLOCK Algorithm

Since the RECURSEOBTAINLOCK traverses from the root down to a leaf (or stops earlier if a white or black node is reached), this algorithm must traverse O(h) nodes, where h equals the height of the document tree. The work involved at each node is O(1) since the work in processing an individual node involves updating references/pointers, coloring, and grey count (integer) values. It is possible upon a lock request failure that the RECURSEREMOVELOCK function will be invoked, but this RECURSEREMOVELOCK (as discussed below) runs in O(h), thus it is not asymptotically greater than the existing O(h) work for the OBTAINLOCK algorithm. Additionally, if sharing or demotion occurs, then the reduction algorithm is run and the history buffer must be incorporated into the requesting user's copy of the requested node, and this will incur O(b) work where b is the size of the history buffer. Thus the overall cost for the OBTAINLOCK algorithm is the cost to update the coloring of at most h nodes (as traversal down the tree occurs) + the cost of updating the coloring of the siblings of x (which is O(1)) + the cost of reduction and enacting Δx on the requesting user's copy of x – for a total of O($h + b$).

Communication occurs when the lock is granted where there was a previous owner – either when a black node is reached that has adopted OT sharing or when a black node is reached and demotion is resolved. In either of these cases, only one history buffer is communicated to the user requesting the lock, thus the communication cost for transmitting this cached history buffer is O(b) where b is the size of the single reduced history buffer communicated. Additionally, as the algorithm traverses down the tree, peers that managed each of the nodes along the path traversed must handle the lock request; thus as many as O(h) peers must be involved in resolving the lock request – and this incurs O(h) communications among a pair of peers (between the peer that manages the node and the requesting peer). Thus the total communication cost in RECURSEOBTAINLOCK is O(b + h).

```
REMOVELOCK($w$, $u_i$)
    if $w.owner = u_i$
        then RECURSEREMOVELOCK(ROOT, $w$, $u_i$)

RECURSEREMOVELOCK($n$, $w$, $u_i$)
    if $n.color$ = black and $n.owner = u_i$
        // remove the lock, but no promotion is possible at this point
        then    REMOVEFROMSIBLINGLIST($n$)
                UNSETLOCK($n$)
                if OTENABLED($n$)
                        remove $u_i$ from n's distribution engine
    else if $n.color$ = grey
        then    $n.greyCount = n.greyCount - 1$
                if $n.greyCount = 1$
                // promotion is possible, so locate the correct remaining sibling
                // and promote it to n after obtaining a window lock on the nodes
                then    $a$ = FINDELIGIBLEPROMOTION($n$, $w$)
                        SETLOCK($n$, $a.owner$, $a.originalRequest$)
                        LINKSIBLINGS($n.parent$, $n$, $n.parent.firstBlackChild$)
                        $\Delta w'$ = REDUCE($\Delta w$)
                        COMMUNICATE($\Delta w'$, $a.owner$)
                        replay $\Delta w'$ at a.owner's copy of w
                else if $n.greyCount = 0$
                        // removal occurred before delayed promotion
                        // so just cleanup lock state
                    then UNSETLOCK($n$)
                else    // keep traversing down the list, reducing grey-count as we go
                        // all the while, using the peer-chain to locate the
                        // next peer with which to communicate
                        RECURSEREMOVELOCK(NEXTINPATH($n$,$w$), $w$, $u_i$)
```

Figure 64: P2P RemoveLock Algorithm

RECURSEREMOVELOCK traverses from the root down to a leaf (or stops earlier if a grey or black node is reached), this algorithm must traverse O(h) nodes, where h equals the height of the document tree. The work involved at each node is O(1) since the work in processing an individual node involves updating references/pointers, coloring, and grey count (integer) values. Upon promotion, the FINDELIGIBLEPROMOTION function must be called, but it continues the traversal down the tree from the point where the promotion may occur, thus its work is also O(h). Thus the overall cost for the REMOVELOCK algorithm is O(h). Additionally, if promotion occurs, then the history buffer must be reduced and incorporated into the promoted user's copy of the node being release, and this will incur O($b\log b$) work where b is the size of the history buffer. Thus the overall cost for the OBTAINLOCK algorithm is = the cost to update the coloring of at most h nodes (as traversal down the tree occurs) + the cost of updating the coloring of the siblings of x (which is O(1)) + the cost of reducing and enacting the history buffer (Δx) on the promoted user's copy of x – which is O(h + $b\log b$).

Communication occurs when either a black node is reached that has adopted OT sharing (and the releasing user must be removed from the OT user set) or when a black node is reached and promotion occurs. In the case of the user being removed from the OT user set on a node, this incurs O(n) communication cost where n is the number of users in the OT sharing set on the node being released (since all users in the set must be notified of the user leaving the set). In the case of promotion, one reduced history buffer is communicated to the user that is promoted, thus the communication cost for transmitting this reduced history buffer is O(b) where b is the size of the single reduced history buffer communicated. Additionally, as the algorithm traverses down the tree, peers that managed each of the nodes along the path traversed must handle the lock request; thus as many as O(h) peers must be involved in resolving the lock request – and this incurs O(*h*) communications among a pair of peers (between the peer that manages the node and the requesting peer). Thus the total communication cost in RecurseObtainLock is O(*n* + *b* + *h*).

```
NextInPath($n$, $w$)
        Assuming we are currently at node n, determine the next peer in the
        communication chain to the destination node w and return this next
        peer (i.e., begin communication with the next peer)

Communicate($\Delta n$, $u$)
        Send the history buffer $\Delta n$ to the peer/user u

Reduce($\Delta n$)
        // Combine operations in $\Delta n$ such that
        // the size of $\Delta n$ (i.e., # operations) is reduced.
        Sort all operations based upon their position
        Remove all pairs of Op and ¬Op as they have no resultant effect
        Combine all adjacent Insert operations
        Combine all adjacent Delete operations

The following supporting functions remain the same with the client-server
implementations (see Figure 43):

        FindEligiblePromotion($n$, $w$)
        SetLock($w$, $u_i$, $r$)
        LinkSiblings($n$, $a$, $b$)
        UnsetLock($w$)
        RemoveFromSiblingList($n$)
```

Figure 65: P2P Supporting Algorithms

The supporting functions FindEligiblePromotion , RemoveFromSiblingList, SetLock, LinkSiblings, and UnsetLock implementations and analysis are the same as presented in Section 4.6. As presented in Section 7, Reduce runs in O(*n*log*n*) where n is the size of the history buffer. NextInPath requires O(1) as it only looks down one level to a child of the current node. COMMUNICATE requires O(b) where b is the size of the history buffer (Δn) to communicate.

The cost associated with a user editing the content or structure of the document tree contains 4 cases to be considered as defined in Section 6.3. In case 1, the edits occur locally, so there is no communication cost and the computation cost equals the cost of inserting the operation into the history buffer – which is constant time. In cases 2, 3, and 4, these involve modifying the document tree's structure which incurs constant computation cost; if there are other peers in v, then these changes incur a communication cost of a multicast message to the peers in v to update the peers of the structural change + a unicast message to the parent of v to denote the deletion or creation operation.

Leaving a section w and retaining ownership on w is equivalent to moving with multiple-writers (case 3 of as defined in Section 6.5). Most costly would be when a user leaves a section w and Δw is transmitted to the user managing the sibling of w (as seen in Figure 61). In this case, locating the remaining peer (U_2 in this example) is achieved in constant time and no communication since v maintains references to its black and grey children and there is only one remaining black child (otherwise promotion would not occur). Updating the coloring of v is also achieved in constant time with no communication. The dominant cost of this event is defined by transmitting Δw to U_2 (the remaining peer). Thus, the overall computation cost is constant (since we can create Δv in constant time) and the communication cost is proportional to a multicast message to the peers in x (since Δw must be transmitted to each of them to construct Δv).

Moving a user U_i from one section v to another section x involves removing the user from v and inserting the user into x. Optionally, the user may retain ownership on *v*. The most costly case involves removing and inserting – the combination of the costs of OBTAINLOCK and REMOVELOCK – for a total of O(h + b).

In existing OT algorithms, all changes are broadcast to all peers within the system, incurring a substantial communication cost. Even if these changes are cached locally, transmitting them in batch to other peers to reduce the overhead cost of small-payload messages incurs a communication cost proportional to the number of operations performed. The computation cost of OT algorithms is also proportional to the number of operations that are passed into the OT engine. We improve upon this by localizing the OT engine to a single node, achieving the performance gains of [66] or [90] but we also reduce the number of operations performed overall through our propagation technique outlined earlier in this section; since the REDUCE function aggregates many, smaller operations into fewer, larger operations, fewer operations must be transmitted to peers and run through the OT engine. Our REDUCE function combines n operations performed at v into fewer number of operations to be performed at the parent of v. Consequently, each time a set of changes made at v is propagated up the document tree, fewer operations must be implemented at the parent of v.

Note that in the P2P version of FINDELIGIBLEPROMOTION, no communication is needed with other peers since we remain in the peer who owns the sub-tree rooted at n.

6.7. Locating the Peer and Ownership

It is essential that peers within the system efficiently locate nodes that are managed by other peers; for example, if user U_1 desires to edit node v, user U_1 must be able to determine which other peer in the system holds the up-to-date cached copy of v. Peers must be able to traverse through the document hierarchy efficiently, and since this tree is distributed among the peers, we employ references at each node that point to the parent and children of owned nodes. As a result, all grey nodes maintain references to the peers within the system that manage the grey and black sub-trees; additionally, all grey and black nodes maintain references to their parents within the hierarchy. We note that although initially those peers owning/maintaining the root and its close descendants must handle more navigation traffic, most users will operate at the lower levels, thus spreading the traffic load over time.

When a user enters or leaves a section, it is possible to adjust the lock/ownership information of other peers (either demoting them in the case of entering a section or promoting them in the case of leaving a section). It is essential that the user is able to locate the peer that holds the node to be promoted or demoted. An algorithm for peer location such as Chord [126] may be adopted to efficiently locate the peer.

In the case of demotion, and using Figure 60 as an example (when user U_2 enters and user U_1's ownership is reduced to not include x), U_2 begins its search for the owner of the desired node (x) at the root of the document tree or by querying cached peers that had previously been visited upon descending originally through the tree. There are three cases at any node: (i) it is painted white, in which case ownership is obtained and U_2 obtains maintenance of x; (ii) it is painted grey, in which case this grey node maintains a reference to all of its grey and black children, and one of these can be followed using a technique similar to [111]; and (iii) it is painted black, in which case the destination peer has been found and OT can be employed or demotion can be employed and U_2 obtains maintenance of x.

In the case of promotion and using Figure 61 as an example (when a user U_1 leaves a node *w* and contention is removed), the user can immediately locate the peer to promote, *x*, as *x* is the only sibling of *w* that is black (all others must be white). Thus U_1 queries the peer that maintains its parent node (*v*), and this peer responds by promoting U_2's ownership to *v*.

6.8. Replication, Congestion, and Fault Tolerance

As pointed out in [144], reliability and performance are the two primary reasons for replicating data. When a distributed system such as a RTCES utilizes replicas of the shared document, local response time (performance) is improved, but communication costs increase; further, reliability is increased because each user has a copy of the shared document, so if one user's replica is lost, the other users may communicate the document state to restore the session for that user.

We may increase the reliability and fault tolerance by replicating the top portion of the document tree among all peers (or a subset of peers). For reliability, a few upper nodes may be replicated (shared) using an OT policy. While this increases the cost in processing the OBTAINLOCK and REMOVELOCK algorithms (since all peers must perform OT to maintain consistency regarding the lock states among the shared top portion of document tree), this approach does overcome the single point of failure of a single server (or a single peer) managing the root. This replica-based approach for the top of the tree is visualized in Figure 66. Here, the top two levels of the tree are replicated among all users in the RTCES, and OT consistency maintenance is applied to ensure each replica contains the same state for node coloring and ownership. At depth 3 and below, individual users (user 1 = blue, user 2 = green, user 3 = red, and user 5 = orange) maintain the document tree state and handle specific lock release and request operations in these sub-trees.

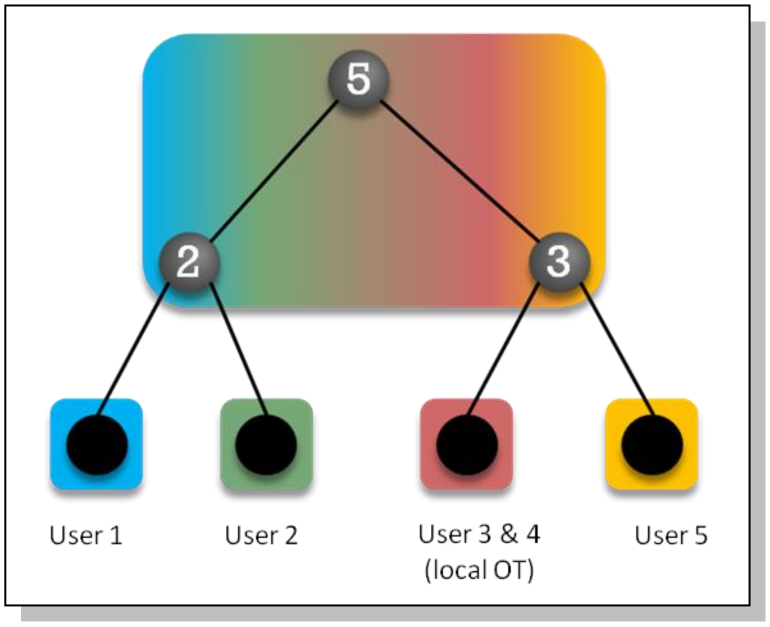


Figure 66: Replication of the Top of the Document Tree and Localized Management Below

While our initial client-server dynamic locking approach reduces this communication time, since most messages (all non-OT update messages) pass through it, the server suffers as a bottleneck for communication. Our motivation in developing the peer-to-peer version of our dynamic locking algorithms was to avoid this problem by distributing the work of lock management among the peers. Initially, it would seem that this work and communication is distributed uniformly among the peers, but the drawback remains that all messages must be processed from the root down. The grey counts must be maintained from the root down to ensure proper promotion and demotion. Thus if a single peer is responsible for managing each node in the tree, some peer must maintain the root and will then become the bottleneck as in the client-server approach.

To address this problem of congestion and an imbalance of the workload falling to a single peer who manages the root node, we examine how this workload may be balanced among multiple peers. Note that in Figure 59 the root is managed by User 1 since User 1 was the first user to enter the RTCES, and unless another policy is adopted to balance the workload of the root, User 1 will continue to manage the root until he leaves the RTCES. Thus all OBTAINLOCK and RELEASELOCK requests must pass through User 1 – creating an imbalance in the workload. We correct this by noting that it is possible to implement a shifting approach to managing the root as follows. When an OBTAINLOCK operation is performed, the user requesting the operation begins managing the nodes along the path in the document tree visited in fulfilling the OBTAINLOCK operation. But when a RELEASELOCK operation is performed, this implies that the user is leaving a section and thus it is not advantageous to have the user begin managing the nodes along the path in the document tree visited in fulfilling the RELEASELOCK operations. In this manner, we adopt a “most recently requested” policy in that all nodes n_i will be managed by the user who’s OBTAINLOCK request was fulfilled by passing through n_i (i.e., n_1, n_2, ... n_k is in the path from the root to n_k, where n_k is the desired node or the node at which the lock request is fulfilled).

If such a “most recently requested” policy for lock management is adopted, then the most consecutive requests a single peer p must serve would be O(n) where n is the number of peers in the collaboration. This is true because if an OBTAINLOCK request is handled, then the node acquires a new manager other than p. Only RELEASELOCK requests can be fulfilled and keep the same manager *p*, and there can be at most *n* consecutive RELEASELOCK request since any more would necessitate a lock request (i.e., a peer can't release a lock it doesn't have). Given the repetitive nature of lock request and release of users moving from section to section of a document, the workload of managing the nodes within the document tree should be balanced as the amortized time a peer manages a node should be approximately equal to the amortized time the other peers manage the node. We also note that the time a peer manages a node is proportional to the depth of the node in the document tree (since there are fewer paths that travel through a node at a greater depth than a node at a more shallow depth). Thus the root management should change more often than a near-leaf node. This is good because the workload of more shallow nodes in the tree (closer to the root) is more than the workload of deeper nodes. As a result, an in particular if users' editing patters enable a higher degree of caching (via clustered editing patters), the workload in managing the distributed P2P version of the document tree should be balanced among the peers.

6.9. Simulation and Results

The goal of this simulation is to investigate moving from a client-server architecture for a RTCES that implements our hierarchical, dynamic locking with the integration of OT to a P2P architecture for a RTCES that implements our hierarchical, dynamic locking with OT integration. Primarily, we are interested in how this architectural change affects the work load and if message and computation costs may be load balanced among the peers/users within the RTCES. Figure 67 shows the OO model used for the simulation.

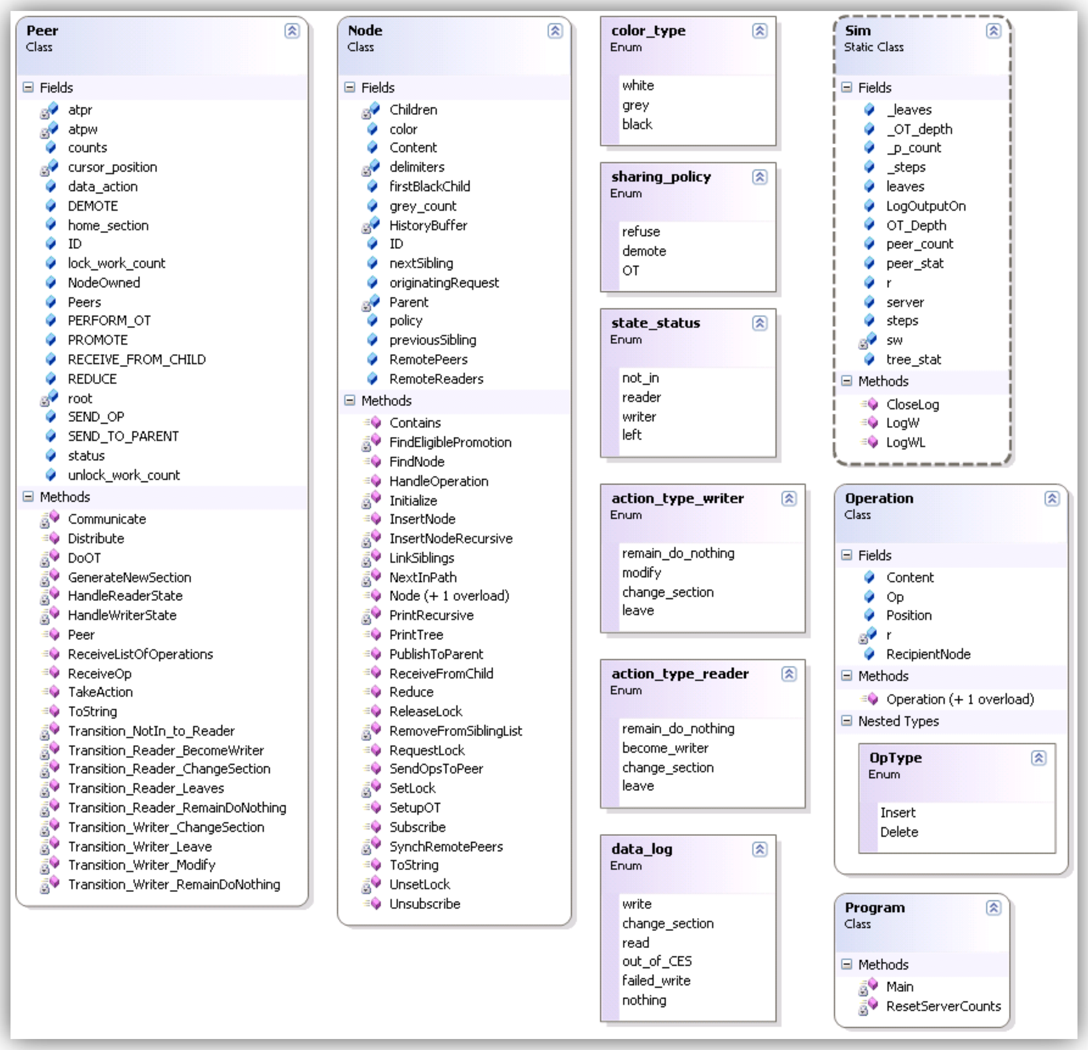


Figure 67: OO Model of the P2P Document Management System

To validate our P2P distributed document management approach, we implemented the model of the node and the OBTAINLOCK and REMOVELOCK algorithms. We modeled three different document trees containing 14, 28, and 56 leaves, respectively. We simulated concurrent users that were either in a reading or writing state; additionally, the users could move to a new section of the document (moving their cursor position), and this new section to which to move was randomly selected. A total of 96 simulation configurations were performed, varying among the three documents and increasing the number of users from 1 to 32.

The results of the 96 simulation runs are shown in Figure 68. Each column denotes a set of peers varying from 1 peer (in simulation runs 1-3) to 32 peers (in simulation runs 94-96). The workload is measured by how many OBTAINLOCK and REMOVELOCK requests were handled on a per-peer basis, thus each point plotted denotes how much work a single peer handled. Note that the y-axis is logarithmic to enable the variance among the peers within the columns to be visible.

If we adopt a first-come policy of node management, then as predicted, one (or a small few) peers are unfairly burdened with the bulk of the document management. Notice the high trend line showing the most burdened peer for each simulation run. When the “most-recent,” balanced approach is adopted, the work is more fairly distributed among all peers. This is corroborated in that while the total work remains the same, the variance among the peers for any simulation run decreases when a balanced approach is adopted (note the increased clustering). Adjacent columns (n, n+1, and n+2 where n is a multiple of 3) denote the different document sizes (14, 28, and 56 leaves); so, for example, simulation 94 contains the 14-leaf document, simulation 95 contains the 28-leaf document, and simulation 96 contains the 56-leaf document. We observe that the total workload decreases when the document size increases. This is intuitive in that if we increase the document size while retaining the same number of peers, then the opportunity for caching increases under our distributed document management model.

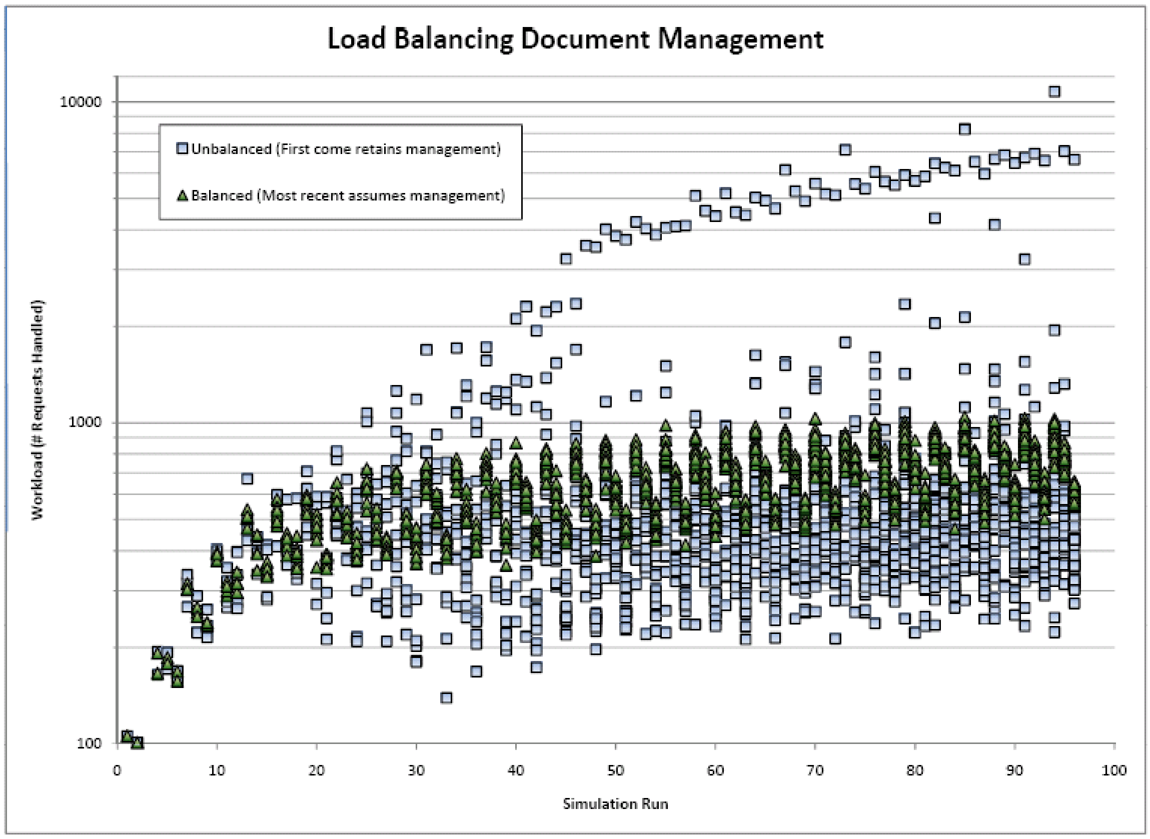


Figure 68: Balancing the Workload of Document Management among Peers

Figure 69 shows how our hierarchical distributed document management approach can reduce the communication costs when compared to a pure OT approach. The topmost three trend lines show how pure OT performs on various document sizes (14, 28, and 56 leaves). The ability to cache changes locally and localize OT to a subset of users sharing the same space within the document dramatically decreases the communication costs of the RTCES. We note that as the collaboration density (the average number of peers per section of the document) increases, the communication also increases; this is as expected since more messages will be sent to maintain consistency when more than one peer shares a section of the document.

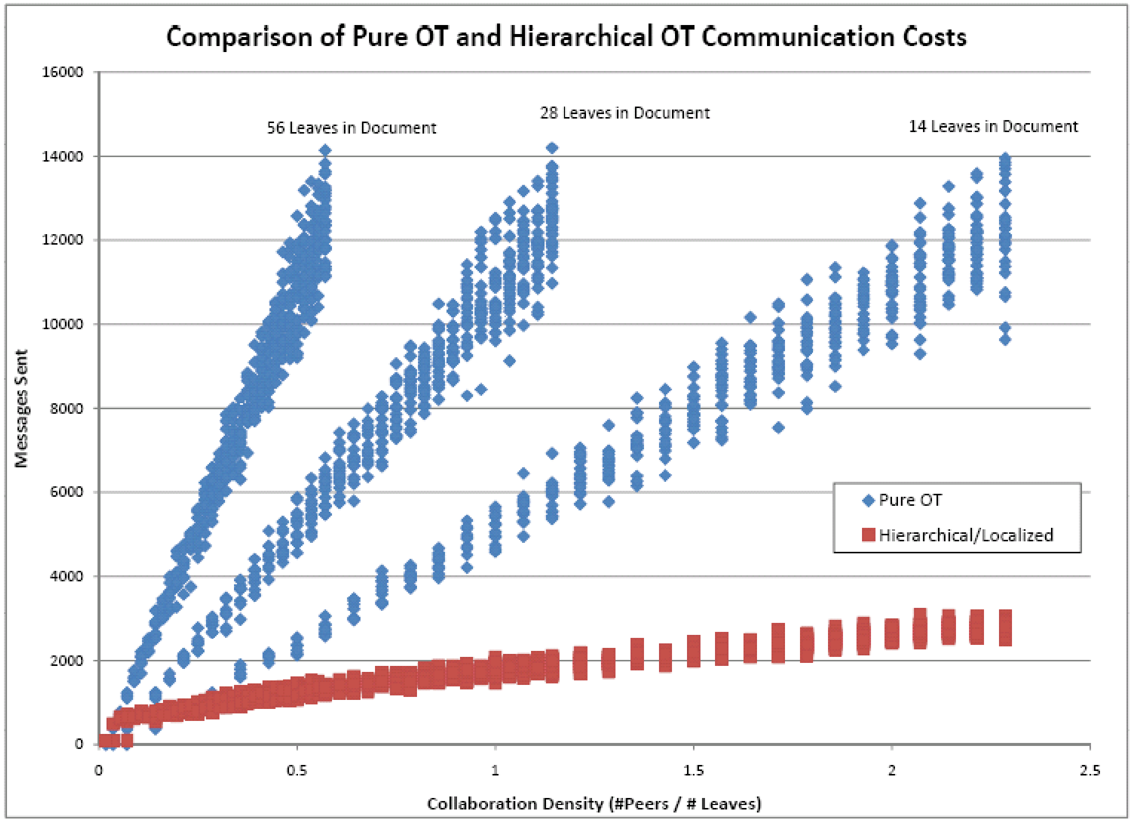


Figure 69: Pure OT vs. Hierarchical OT Communication Costs

6.10. Summary

In this chapter we have shown that our client-server algorithms for dynamic lock management can be extended into a P2P architecture where the workload of handling the lock requests and lock releases can be distributed among the peers within the collaboration. The overall algorithms and data structures are similar to the client-server approach, and we have demonstrated that they are efficient and correct. By utilizing existing efficient location algorithms such as Chord, we are also able to quickly locate the peer who is managing the nodes in the document tree. We have removed a central point of failure at the server and enabled fault tolerance via replication of the top of the tree, and we have shown that the workload is theoretically distributed fairly among the peers. Our empirical results via simulation demonstrate that our P2P approach is scalable and the work of managing the document tree is indeed distributed fairly among the peers.

CHAPTER 7

HIERARCHICAL REDUCTION AND INTENTION PRESERVATION

Now that we have developed client-server and P2P document tree management algorithms and demonstrated how our approach can integrate best-practices of OT, we turn our attention on how we may better manage the changes (operations) performed by users within the RTCES. We note that the cost associated with OT increases as the size of the history buffer increases, so in this chapter we focus on how the size of the history buffers may be reduced throughout the hierarchical document tree. Additionally, we identify opportunities to better achieve intention preservation as the history buffers are propagated up the document tree hierarchy.

Section 7.1 presents the process of reduction of the history buffer and sending these reduced history buffers up the document tree in a pipeline fashion. Section 7.2 presents the modeling of the node to achieve the reduction process. Section 7.3 discusses how our approach creates opportunities to better achieve intention preservation – one of the significant open problems in RTCES research. We then present simulation and results demonstrating how history buffer size can be managed using our reduction process in Section 7.4. We discuss related work in Section 7.5 and provide a summary in Section 7.6

7.1. Reduction

Based upon its structure, a document may be broken up into sections, subsections, paragraphs, sentences, words, etc. If the document being shared is a CAD drawing, it may be broken into layers, objects, etc. If the document is programming source code, it may be broken into classes, components, methods, blocks, etc. Thus, we assume a document tree structure without any preconceived notion of what the sections of the document contain, nor do we require any specific depth/level of decomposition. Our approach works well with a variety of document structures. The document tree consists of internal nodes that represent structure, and all document content resides at leaf nodes, thus users only make changes at leaf nodes within the document tree. Consequently, we initially employ OT at the leaf node and cache changes made by users, only communicating changes to other users that are interested (or currently viewing/editing) the same section. As a result, we minimize the OT computation and communication costs [75][90]. But as changes are made, the history buffers of leaf nodes grow and performance of the OT algorithm degrades.

We agree with Oster [90] who recommends "compression of history buffer" at various key points in time in his Tombstone Transformation Function (TTF). This reduction is appropriate to keep the size of the history buffer from growing too large and degrading the performance of the OT integration algorithm. Many operations made within a section of the document should lend themselves to being consolidated into fewer, larger operations. As an example, assume the user performs the following series of operations on section *v*: *Insert*["This", *v*, 0], *Insert*[" is", *v*, 4], *Insert*[" a", *v*, 7], and *Insert*[" sentence.", *v*, 9]. They may be combined into one: *Insert*["This is a sentence.", *v*, 0]. By reducing many operations into a granular, single operation, the history buffer may be minimized, and a larger, single operation may be relayed to other users; overall, communication cost is reduced by transmitting fewer, longer messages rather than transmitting many, short messages [90][103].

We next address how such “compression/reduction” of the history buffer may be achieved. TTF preserves the *absolute* position of all operations and objects being modified; as a result, operations within the history buffer may be reordered without modifying the result of the operations. Consequently, this enables us to manipulate the operations stored in the original history buffer to result in an equivalent modified history buffer. Let Δv = the history buffer of a section v, v' be the resultant state after performing Δv on v, where and $\Delta v'$ = Reduce(Δv). Since reduction does not change the intention of Δv, $v' = \mathrm{v} + \Delta v = v + \Delta v'$, where + denotes the application (or “replay”) of operations. Thus, we could reorder and reduce the operations while retaining the intention of the original operations. This reordering is essential as our reduction algorithm relies upon the equivalence of an initial history buffer to its reordered set of operations.

As previously noted, users only make changes at leaf nodes within the document tree. Thus we initially employ OT at the leaf node and minimize the OT computation cost. The history buffers of leaf nodes will grow as more changes are made, but it would be advantageous to reduce these and when permissible at certain key times, to consolidate these into fewer operations that retain the intention of the operations performed on this section. Since the history buffer is required to assure total causal ordering in OT algorithms, we cannot reduce the history buffer without knowing that such a reduction will not later inhibit the OT algorithm; consequently, we may only reduce a history buffer (Δv) at node v when

1. A user U_1, who owns v, leaves v and ownership of v is promoted to another user U_2 (see Section 6.4 and Figure 61). Thus, Δv may be reduced because all users have left v and no operations remain that change v.
2. Based upon some event in the CES wherein users wish to accept changes made to a section and all users in v synchronize (using a barrier) such that all copies of v residing at the users in v have converged (i.e., all operations have been replayed at all users in v). This follows a natural divergence-convergence model [28].

All operations contain a position element denoting where in the document the operation occurs. This position information is any ordinal type, but for simplicity and without loss of generality, we assume this to be an integer denoting the operation's position within the section of the document to which the history buffer applies. Further, these integers denote positions relative to each other, so we can compare two operations to see which proceeds and which follows.

Having established that operations' positions are known relative to each other and that operations may be reordered without changing their effect, we express the reduction process as follows:

1. Sort all operations (keyed on position) within the history buffer.
2. Remove all adjacent pairs of Op and ¬Op (since they cancel each other).
3. Combine sets of adjacent insertions and combine sets of adjacent deletions.

These three steps are visualized in Figure 70. History buffer Δv denotes the initial history buffer. 1) shows the history buffer after it has been sorted by position (after step

1). 2) shows the history buffer after removing the Op and ¬Op occurrences (the removed operations are highlighted in red). 3) shows the resultant history buffer after combining adjacent insertions and deletions and demonstrates that a series of adjacent insertions can be combined into a larger insertion and a series of adjacent deletions can be combined into a larger deletion; also note the semantic abstraction from character-based operations to word-based operations at this step.

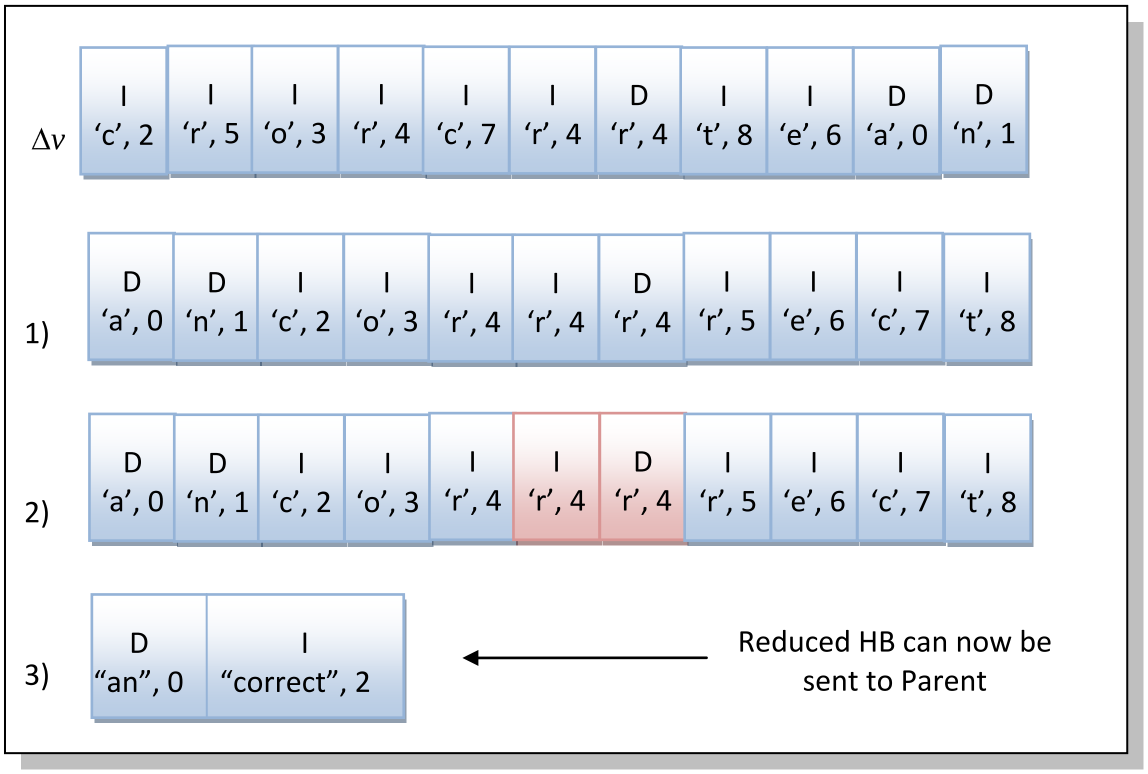


Figure 70: The Reduction of a History Buffer

Assuming an OT algorithm such as TTF that preserves equivalence in reordered operations is utilized, then step 1 does not change the resultant state of the document. Since Op and ¬Op result in no change to the document state, removing pairs of these as done in step 2 does not change the resultant state of the document. In step 3, we combine sets of insert and delete operations into larger granular insert and delete operations that retains the same effect on the document, thus the resultant state of the document is not changed in step 3.

This algorithm is efficient. Step 1 may be realized using any standard linear sorting algorithm in O(*n*) because the keys are bounded by the size of the section; steps 2 and 3 each require one traversal of the set of operations in O(*n*). Thus, the overall efficiency of this reduction algorithm is O(*n*). Considering that the goal is to keep the history buffers small, *n* is expected to be small and the runtime of this reduction algorithm is also reasonable.

Further, since the reduced history buffers are sent up the tree and combined at semantically-higher levels, we may pipeline the reduction. For example, all history buffers at the leaf nodes are reduced and sent to the next level up in the tree; then the history buffer of the parent nodes receive and combined the incoming history buffers from their child nodes. These are reduced and sent higher, etc. At each level, reduction can proceed in parallel and the pipelining realized.

7.2. Hierarchical Reduction

When a reduction occurs, it is useful to transmit these semantically “larger” operations up within the document tree such that these larger operations may be stored in the history buffers of the ancestor nodes. For example, many insertion and deletion of words may be reduced to fewer insertion and deletion of sentences. This process of reduction and transmission up the document tree is demonstrated in Figure 71. In this example, changes made by U_1 to w (Δw) and changes made by U_2 to x (Δx) are reduced to $\Delta w'$ and $\Delta x'$ respectively and transmitted up the document tree to v. Thus v’s history buffer contains the reduced changes denoted by Δ*v*. Later, U_1 makes more changes to *w* (Δ*w*′′) and U_2 makes more changes to *x* (Δ*x*′′).

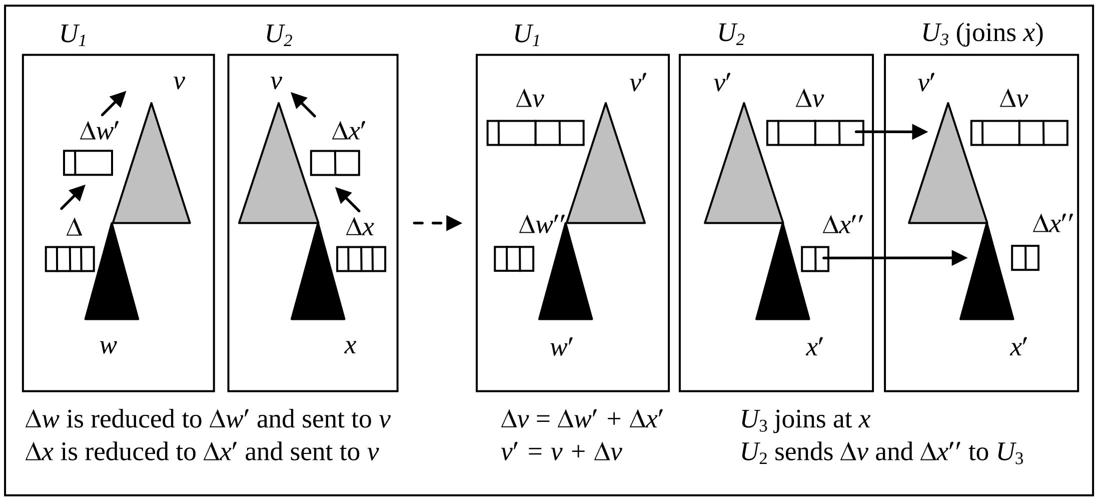


Figure 71: Hierarchical Reduction

The message cost savings of hierarchical reduction is demonstrated when U_3 enters *x* and communicates with U_2 for the latest version of *x*; *x* at U_3 is made current by transmitting Δ*v* and Δ*x*′′ from U_2 and applying these operations on U_3's copy of *x*. Additionally, U_3 has a copy of *w*′ (where *w*' = *w* + Δ*w*), since Δ*w*′ was contained in Δ*v*; Δ*w*′ may be replayed on U_3's copy of *w* such that these copies of *w* are only missing Δ*w*′′. Without hierarchical reduction, all individual changes stored in Δ*v* at U_2 would have to be transmitted and replayed at U_3. As a result of hierarchical reduction, fewer operations must be transmitted and replayed at U_3. In existing OT algorithms, all changes are broadcast to all peers within the system, incurring O(*n*) communication cost per operation. Even if these changes are cached locally, transmitting them in batch to other peers to reduce the overhead cost of small-payload messages incurs a communication cost proportional to the number of operations performed. The computation cost of OT algorithms is also proportional to the number of operations that are passed into the OT engine. We improve upon this by localizing the OT engine to a single node and decreasing the number of operations performed overall through our propagation technique. Since the *Reduce* function aggregates many, smaller operations into fewer, larger operations, fewer operations must be transmitted to other users and run through the OT engine.

Further, since the reduced history buffers are sent up the tree and combined at semantically-higher levels, we may pipeline the reduction. For example, all history buffers at the leaf nodes are reduced and sent to the next level up in the tree; then the history buffer of the parent nodes receive and combined the incoming history buffers from their child nodes. These are reduced and sent higher, etc. At each level, reduction can proceed in parallel in a pipelining fashion.

7.3. Intention Preservation

Intention preservation has been an elusive problem in RTCES for the past decade. While OT achieves convergence and causality preservation, intention preservation is not guaranteed by OT. Thus we turn our attention as to how our approach may address this open problem in RTCES research. We begin by noting that our approach in maintaining a document tree representation of the shared document is superior to the linear representation of the shared document typically employed by OT algorithms; we substantiate this claim by pointing out that semantic knowledge is captured in the structure of the document tree – hierarchy implies structure in that like elements are grouped together, just as this dissertation is grouped into chapters, and all sections within a chapter are logically related, and all subsections within a section are logically related, and all paragraph within a subsection are logically related, etc.

Having presented the reduction algorithm and established that such reduction may occur at times when promotion and demotion occurs as well as when users agree it should occur (at a specified time automatically or at a user-generated synchronization event), we utilize such reduction to better achieve intention preservation.

When reduction occurs, the operations that occurred and are stored within the history buffer at one semantic level are reduced into meta-operations and passed up the document tree to nodes at the next higher semantic level. It is at this point when reduced operations are brought together at a higher semantic level that we have an opportunity to examine the operations to see if a semantic violation occurs and if the combined set of operations creates a problem in ensuring intention preservation. For example, [58] points out that while locally-correct operations achieve the desired results, when combined, the resultant shared document may achieve convergence and causality-preservation, but the combination of the local semantically-correct operations of the two users results in a semantically incorrect document. As an example consider the following as shown in Figure 72 which demonstrates that even when the sites' replicas converge, the semantic intention is not achieved.

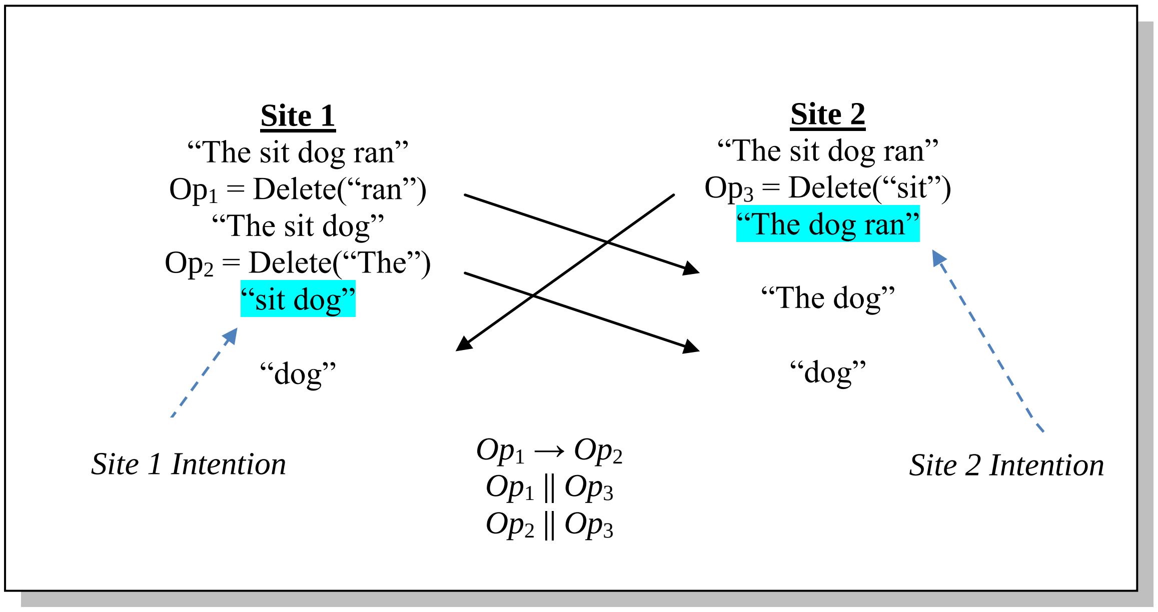


Figure 72: Semantic Intention is Violated

We note that when operations are combined via the hierarchical reduction process, one site's operation(s) should override and the resultant state should be one of the two intended states. In other situations, it could be possible that a subset of the operations should be retained from each site to achieve semantic intention preservation, so it is not exclusively one or the other site's operations that should be retained.

It is precisely at the point of reduction in our algorithm that we can detect such a semantic intention violation. Certainly it is possible to present options to the users and allow the user to resolve the conflict, but in order to do so automatically, semantic knowledge of the content being editing must be defined prior to the operations being performed. While this might seem counter-intuitive (for how can one know the semantic content of the document before the content is authored), [123] presents recent work in rhetoric structure theory (RST) that seeks to establish the structure of the document (referred to as the document narrative or DN) prior to the content of the document being added. This is described as planning out what the document will contain and map the structure of the document prior to allowing collaborating authors to contribute the content of the document. Similar to structured software engineering where the architecture and design are established before the implementation, DN create the overall flow of the document prior to its realization and content completion. RST provides grammatical rules that document must follow, and changes made at different sites might not violate these RST rules locally, but when changes are combined, a RST violation may be detected and dealt with appropriately (either through a priority based scheme, user intervention, or automatically via natural-language processing).

Thus as a result of the reduction process, we enable better intention preservation (an open problem in CES research). Intention preservation is best achieved at a semantically-appropriate level [58][60], and after reduction, changes are propagated up a document tree and accepted or rejected at an appropriate semantic level rather than only at a character level, a limit of existing OT approaches in achieving intention preservation. Consequently, in a scenario in which two users each modify a different word within an incoherent sentence (correcting the semantic problem locally), when these changes are propagated up the tree, we may automatically detect and correct the problem or allow for priority-based or user-intervention correction. Since existing OT algorithms have no semantic/structural knowledge of the document being edited, this opportunity to check for intention preservation has heretofore not existed. Thus our approach of utilizing reduction and propagation of operations up the tree improves the ability to achieve intention preservation.

7.4. Modeling the Peer

Now that the reduction algorithm has been articulated, we integrate it into the nodes within the document tree such that history buffers may be stored at all levels within the document tree, reduced as desired, and propagated up the tree. The components that make up the node model to enable hierarchical reduction are shown in Figure 73.

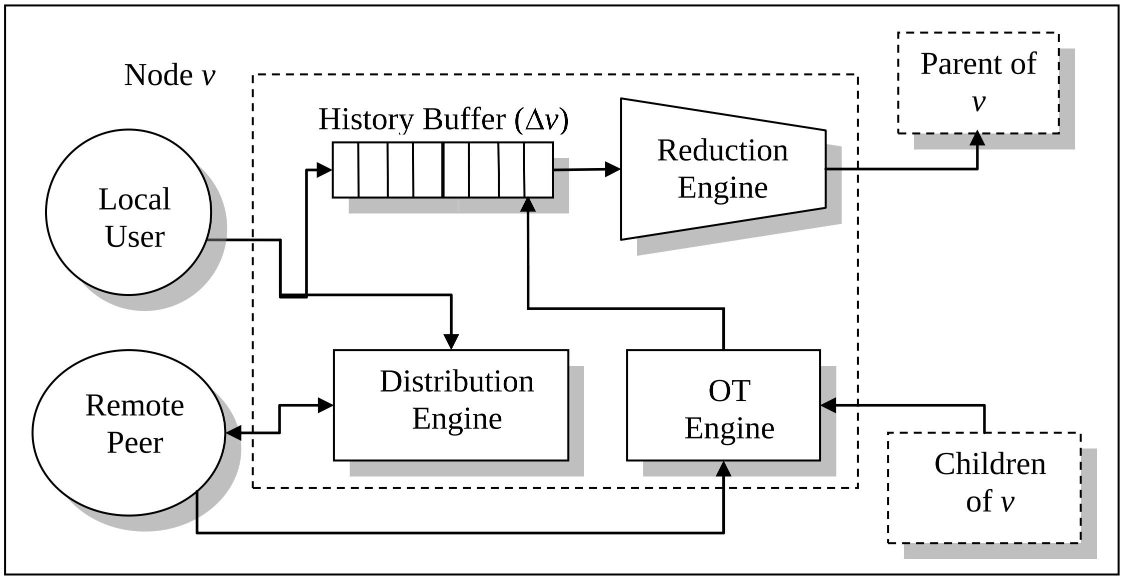


Figure 73: The Components of the Peer

Peers within the RTCES maintain working, cached copies of portions of the document. These portions/sections are represented by nodes within the distributed document tree. In order to correctly process changes being made to the sections of the document, each node must be able to incorporate input from the local user as well as input from other peers. An OT engine is needed to apply the transformations to incoming changes made by remote peers onto the local peer's copy of the section as well as any operations that are sent from children of the node when promotion occurs. When a local user makes a change, this change is stored in the *History Buffer* (HB) to enable OT and ensure total causal ordering of changes [132]; these changes are then sent to the Distribution Engine. It is the responsibility of the *Distribution Engine* (DE) to track which peers are readers and writers of node *v* and need notification when changes are made to *v*; additionally, the DE is responsible for handling requests from peers to join (copy - *v* must be sent to the peer), demote the local user (split - a portion of *v* is sent to the peer and a portion is retained by the local user), and promote (merge - the peer has left/moved and v and a sibling of *v* can be merged together at a common ancestor node). As in existing OT systems, the *OT Engine* is responsible for receiving incoming changes made by a peer and applying the OT algorithms to incorporate the changes made by the peer into the local copy of v; additionally, the OT engine is responsible for incorporating changes that are propagated up the document tree from children of *v*. The *Reduction Engine* is responsible for converting changes made at the level of *v* into meta-changes to be replayed at a higher level in the document tree.

7.5. Simulation and Results

Since the cost of performing OT is dependent upon the size of the history buffer to which it is applied, it is logical to conclude that if the history buffers can be kept small, then the computation cost of performing OT can be kept small. One of the benefits of our reduction algorithm is that when it is performed, the history buffer can be cleared; this is due to the fact that the intention of the operations being reduced are stored higher in the document tree (at nodes semantically higher).

To validate that the reduction algorithm is beneficial in reducing the computation cost of performing OT in a RTCES, we simulated various configurations of document sizes and various numbers of users (increasing the number of users from 1 to 88).

We modeled three different document trees containing 14, 28, and 56 leaves, respectively. We simulated concurrent users that were either in a reading or writing state; additionally, the users could move to a new section of the document (moving their cursor position), and this new section to which to move was randomly selected. The modeling of the user and the document is the same as described in Section 6.9.

But in order to test the benefits of the reduction algorithm to the OT computation costs, it is important to ensure that OT is being performed. Since our dynamic lock management algorithms increase the caching and reduce the necessity of OT, we increased the number of users in the RTCES for this simulation to a maximum of 88; as a result, we achieve collaboration densities (the number of users per leaf in the document tree) to over 6 – which is more than triple than our previous simulation. A total of 264 simulation configurations were performed, varying among the three documents and increasing the number of users from 1 to 88. Additionally, we ran each configuration using no reduction, using minimal reduction only when a promotion or a demotion occurred, and using reduction upon promotion and demotion as well as any time a user entered or left an OT set (the users collaborating within a leaf of the document tree).

The results of the simulation runs are shown in Figure 74. Note that the vertical axis is logarithmic.

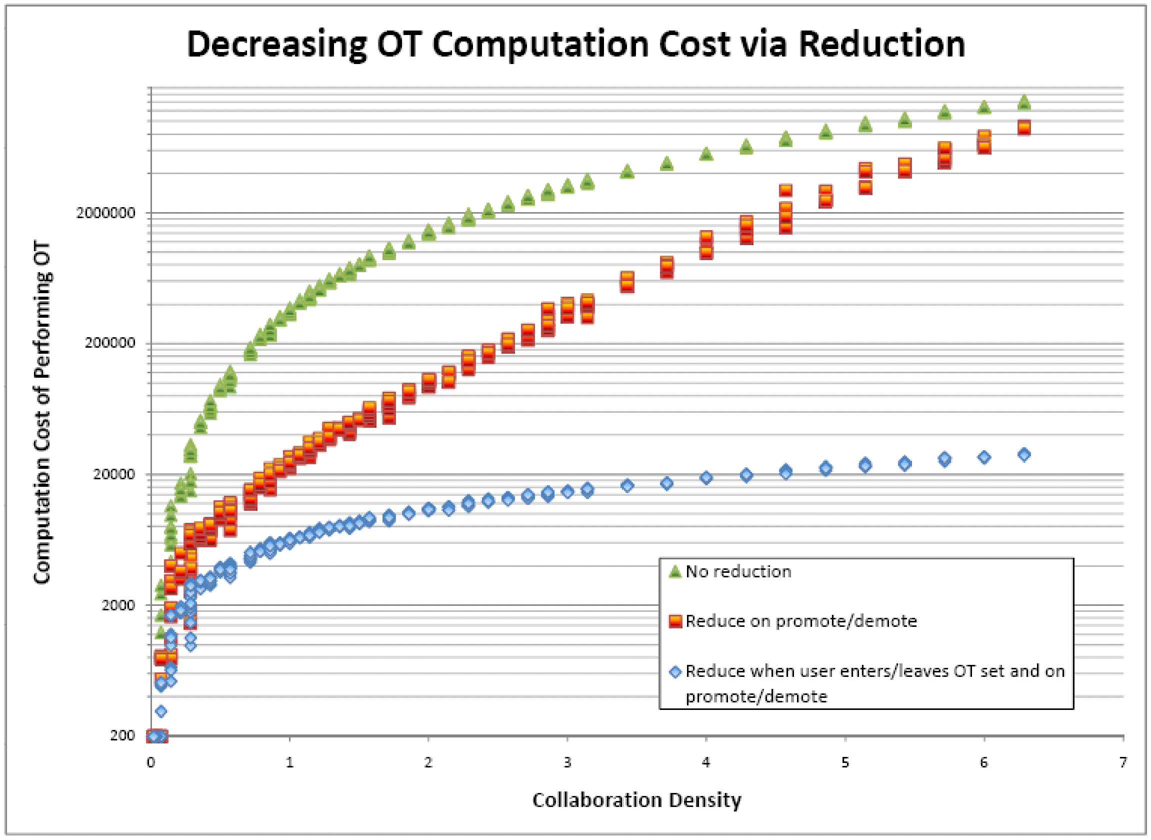


Figure 74: The Reduce Algorithm Decreases OT Computation Costs

Clearly, the cost of performing OT is dependent upon the size of the history buffer to which it is applied. Performing OT where no reduction is applied is most costly. Performing OT with some reduction (when promotion/demotion occurs) is advantageous, but the cost of performing OT is minimal when the reduction algorithm occurs more frequently (upon promotion and demotion and when a user enters or leaves the OT set).

It is interesting to note that while reduction is advantageous to minimize the computation costs of OT, we had to perform it more often that when just promoting/demoting to see the most gains. This is because OT will be performed more often (and thus be more costly) when the collaboration density is higher; if the collaboration density is low, then users are less likely to need to perform OT (since they are less likely to be in the same section at the same time). We found that in such a scenario when collaboration density was higher, promotion and demotion did not occur as frequently; this is intuitive in that with a higher collaboration density, it is less likely that any single user remains in a section and is a candidate for promotion; further, in a high collaboration density environment, most users will have already been demoted to a leaf by previous users' entry into the tree, thus demotion is also not likely. Consequently, we believe it most appropriate to apply reduction when promotion and demotion occurs as well as when a user enters or leaves a shared section (enters or leaves an OT set).

It is important to note that while reduction does decrease the computation cost of OT, the reduction computation cost itself is not significant. As previously defined, the reduction cost is O(n) where n is the size of the history buffer being reduced. This cost is equivalent to performing one operation within the same history buffer; thus if we are willing to incur such a cost for an performing OT on an operation, certainly we are willing to incur this cost for reduction if such a clear overall OT computation cost reduction is achieved.

One disadvantage of performing reduction on an OT set is that all users within the set must perform a 2-phase protocol to synchronize and ensure that no outstanding, non-implemented operations remain unincorporated into the history buffer before it is reduced [91]. This does increase the communication among users within the same section of the document, but the number of users within the same section should be small if our distributed, hierarchical document tree is utilized. Further, this communication cost is quite small relative to the exorbitant cost of broadcasting all operations to all users in a pure OT approach.

7.6. Related Work

[57] discusses managing history buffers in a hierarchical document structure and applying operations at fixed semantic levels within the document (paragraph, sentence, word, character); further, operations are processed from top to bottom, so all operations must flow through the document tree root – posing a significant bottleneck in processing the operations. Rather, our approach is more flexible in supporting operations at any semantic depth and begins the process of managing and applying these operations within the leaf nodes where they occur. From there, reduction occurs and the reduced set of operations (that are meaningful at a higher level semantically) are published up the tree in a pipeline fashion.

[125] discusses the ability to keep some operations private and publish others, which is similar to our work is that local changes can be made and unmade without any other user being made aware of the changes – similar to the process of removing pairs of Op and ¬Op during the reduction process since no one need be made aware of these self negating operations.

The adoption of maintaining semantically-aware history buffers is gaining increased attention in the RTCES research community. [57] utilizes a hierarchical structure to maintain history buffers and applies OT algorithms at different levels within the structure (see Figure 75).

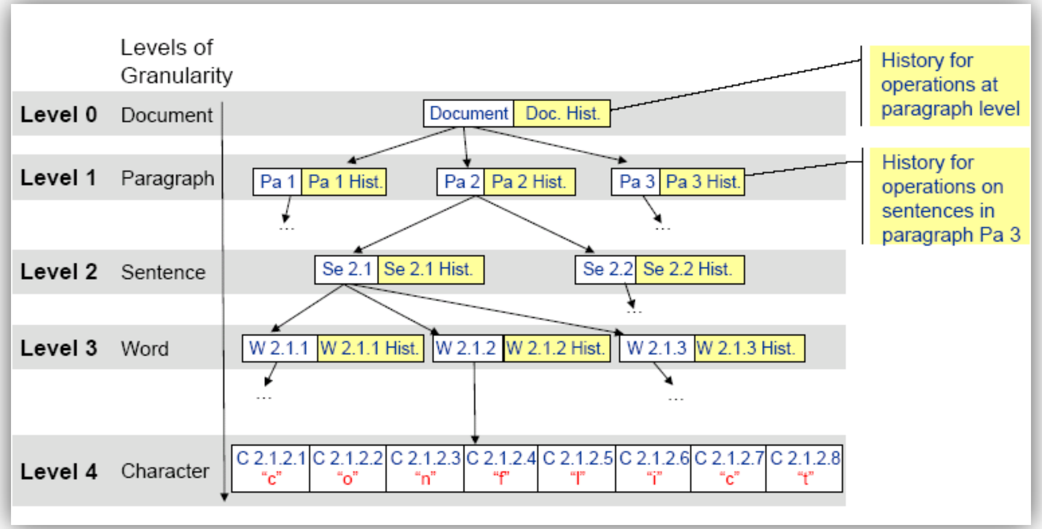


Figure 75: A Hierarchical View of History Buffers [57]

To justify the need for and the potential benefit of our reduction-based approach, it was shown in the document edit profiling research of Papadapoulou [94] that there can be a high amount of operations that nullify each other (such as performing an operations and then performing the inverse of the operations – i.e., performing a DO operations and then immediately performing an UNDO operation). The researchers found that marking such operations as contributions is not necessarily appropriate given the net effect is essentially no operations performed (no contribution to the collaboration), thus it could be beneficial to reduce/remove such combinations to better capture a higher-order view of the document edit profile. This is directly related to and supports our removal of operations that nullify each other; in our reduction algorithm, step 2 removes such pairs of Op and ¬Op as they have no net effect on the document state. Consequently, these non-contributions are removed, and visualizations such as Papadapoulou's that employ our approach of reduction would better display accurate contributions.

Recently, [131] presented their most current work in expanding the capabilities of OT by creating an algorithm that maintains the context of an operation when the OT algorithm is applied utilizing their Context-based Operational Transformation (COT) algorithm. The COT algorithm utilizes a context vector (which is defined by a set of operations) that specifies the context under which an operation is performed. While this approach simplifies solutions to existing CCI problems, it does not solve the intention preservation problem – as semantic knowledge is required to solve this open problem in RTCES research.

7.7. Summary

In this chapter we have shown that our P2P algorithms for distributed document and dynamic lock management can be extended to include hierarchical reduction of history buffers at each node and at varying depths within the document tree. This reduction algorithm is successful in decreasing the size of the history buffers and propagating operations up in the document tree to higher semantic levels. Additionally, we identify the point at which history buffers are merged together hierarchically (at these higher semantic levels) as appropriate points in the RTCES at which intention preservation may be examined as possibly failing; it is at these points that intention preservation violations may occur (and thus we can query the user as to how to resolve the violation and/or automate the violation correction). Our empirical results via simulation demonstrate that our hierarchical reduction approach is viable in reducing the computation cost of performing localized OT. Now that we have successfully developed our theoretical RTCES contributions, we focus the next chapter on implementing prototypes that utilize our approaches.

CHAPTER 8

PROTOTYPE SYSTEMS

System test and performance evaluation are essential in a system development to ensure the system/algorithms under development will not cause major problems when deployed in the real field and used by real users. This is especially important for distributed systems, such as RTCES that has a large number of potential users. Unfortunately, the user-oriented nature of the system prohibits extensive testing and performance evaluation using real users. In this chapter, we follow a stepwise simulation-based design process to test/evaluate the system and algorithms under development. This stepwise design process is motivated by [55] that develops a simulation-based design process to enable smooth transitions between different design stages. It aims to support systematic and cost-efficient testing and evaluation for the distributed collaborative editing systems concerned in this chapter.

This chapter discusses how our simulation design process has allowed us to first move beyond simulating client and server to begin the progress to a functional implementation of both client and server technologies – better achieving an efficient implementation of our algorithms and ideas based upon our empirical simulation results.

8.1. Simulation-based Software Architectural Design Process

The stepwise simulation-based design process includes three steps as shown in Figure 76. In the first step (a), both the server and clients are modeled as DEVS models; clients may have different profiles based on knowledge extracted from real user behavior extracted by analyzing change log files of document repositories. We apply a fast simulation approach wherein events advance the system clock and the simulation completes as fast as possible. At this stage, different configurations (such as varying the number of clients and/or client behavior patterns) can be easily setup, and multiple runs of the simulation may be quickly executed. A key advantage to this approach is that it is very flexible, and we are able to quickly get results without the need to fully implement a research server; this allows for testing and evaluation in the very early stages of the architectural design process. In the second step (b), the server is coded and fully implemented and run on a dedicated computer; simulated client models interact with the server through the network. The key advantage is that there is still flexibility for configuring the tests on the client side, such as having a large number of client models; this is especially cost efficient as no real users are involved and we can scale the tests beyond current RTCES testing user levels. In the last step (c), real users use the client editors to interact with the real server and we collect measurement data. At this stage, we are able to achieve high fidelity measurement of data because this consists of real users and the real server.

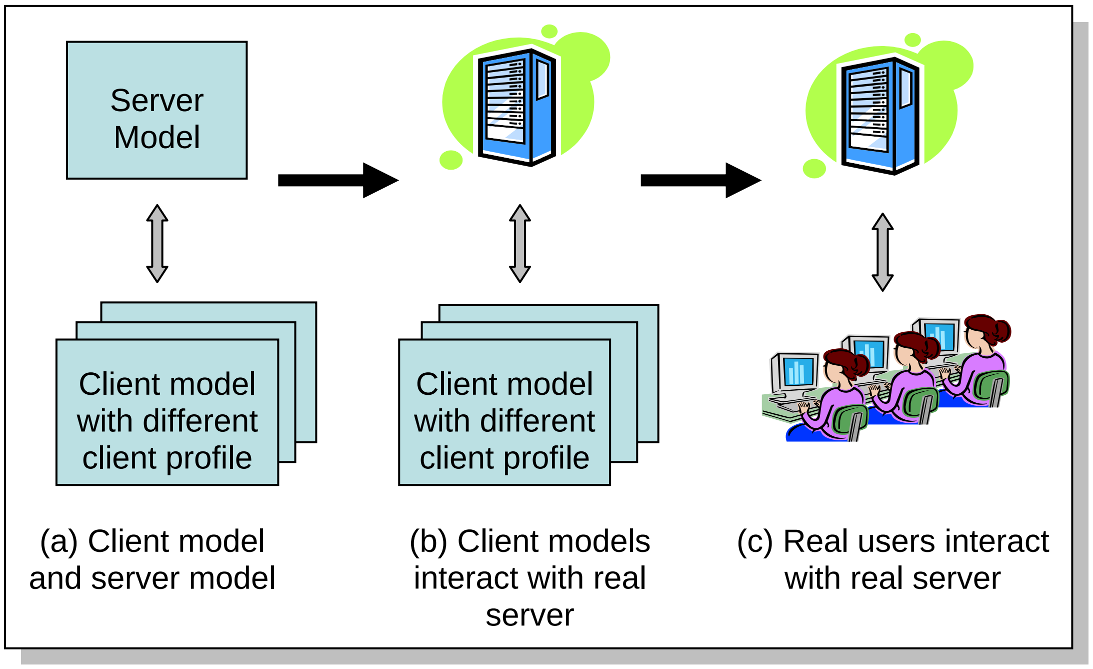

Server Model
Client model with different client profile
Client model with different client profile
(a) Client model and server model
(b) Client models interact with real server
(c) Real users interact with real server

Figure 76: Simulation-Driven Design Process

8.2. Replacing Models with Actual Components

As presented in Chapters 4 through 7, our client and server algorithms effectively support RTCES while minimizing communication and computation costs. We would like to move from simulating each (as we have done in the past) to replacing the server and then replacing the client such that in the end we have fully implemented technologies to support RTCES. This is the natural progression of the simulation-based software architectural design process – moving from the models to the actual implementations.

To realize this goal, we first focus on the server. Porting the algorithms written in the simulation to an actual Web service is straightforward in that the code must be removed from the models in response to external events of the model to being in response to client service calls. There is thus a one-to-one mapping of model event handler for an external transition function in the DEVS model to an event handler for a Web service API method invocation. The only extension needed to a traditional Web service is that we had to make the service state-based so that the document state would be preserved from call to call; this was trivial in that at the beginning of each method call, a *LoadState* function could be invoked by the service to deserialize the document tree state, and at the end of each method call, a *SaveState* function could be invoked by the service to serialize the document tree state.

On the client side, we then replaced the client models with an implementation of a client editor that supported the reading and writing of a document that also connected to the server-side Web service API. When the user moves the cursor, the user's position within the document tree is updated on the server; when the user edits (modifies) a section within the document, concomitant lock request and change messages are generated and sent to the server (and potentially other users in the same section). Lock promotion and demotion messages are sent to clients as needed to ensure each client knows what section(s) he owns.

8.3. Implementing the Server

Having modeled both the clients and the server, we turn our attention to the implementation of the algorithms on the server. We implement the server so that it can be used in a real-world RTCES, but before employing it in a real-world scenario, we would like to validate that our simulation results in modeling the server accurately reflects the real performance that may be achieved when the server is fully implemented. In this scenario, we keep the client machine as previously modeled. The simulated server machine is removed and we add a model called *OutConnection* that sends and receives messages to and from the real server using Web services invocations. The network is then connected to this new *OutConnection* model instead of the previous server machine model.

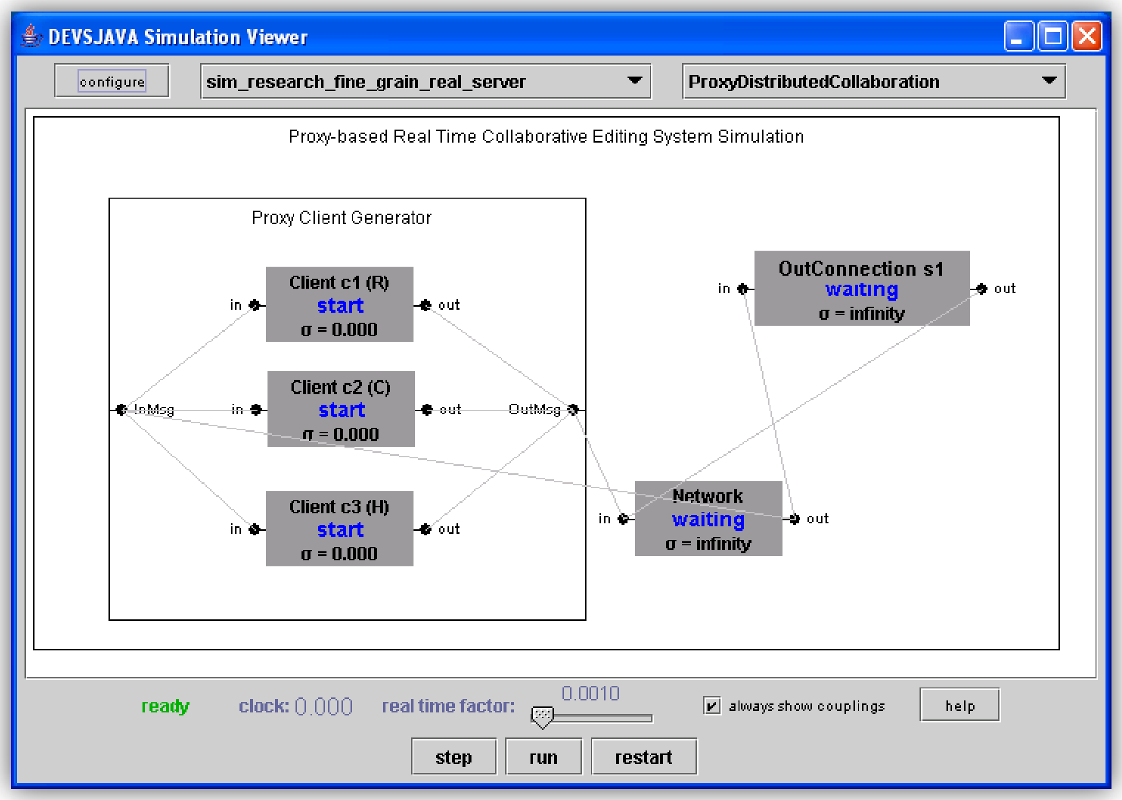
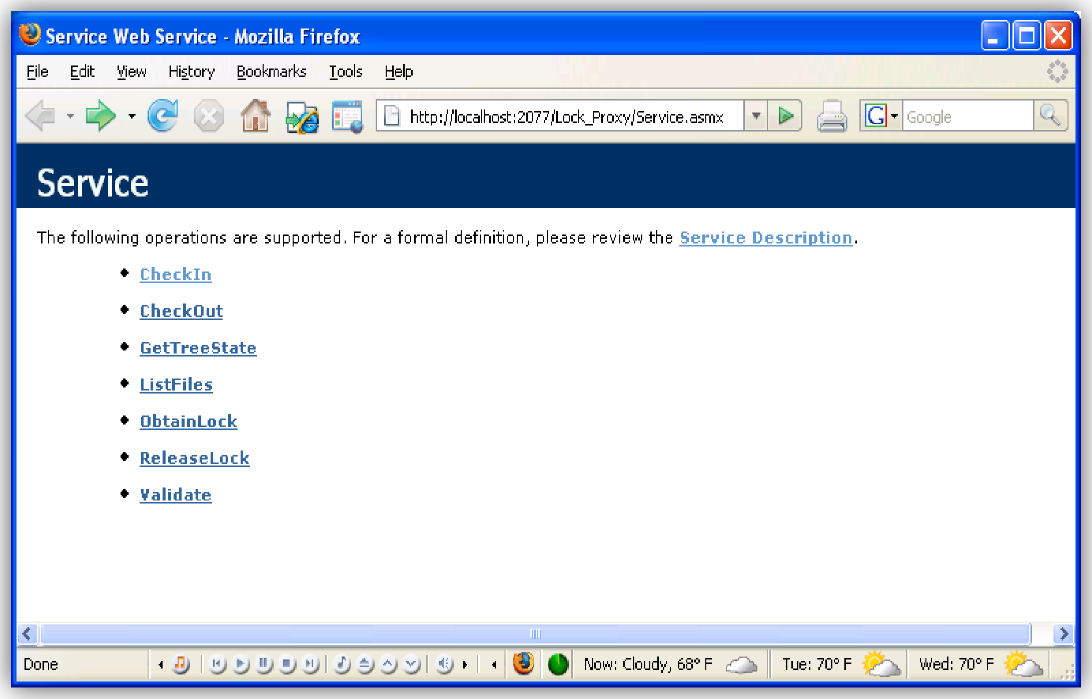


Figure 77: Simulation Connection to Real Server via the *OutConnection* Model

No other RTCES research has been able to test their algorithms under a large-scale scenario with more than a handful of clients. Certainly others have measured performance of their algorithms with a large set of operations (see [66] for a recent example), but OT algorithmic studies focus on how quickly the algorithms may run and the storage capacities required; to date, no RTCES has been systematically tested with a large number of clients, as it is difficult to bring together so many users necessary for such a study. The impact of messages across the network has not been adequately measured in RTCES research, thus we address this cost by simulating a large number of clients connected to a real-world implementation of our server technology. As a result, we are able to determine how our system's performance scales as the number of clients increases.


Service Web Service - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:2077/Lock_Proxy/Service.asmx
Google
Done
Now: Cloudy, 68° F
Tue: 70° F
Wed: 70° F

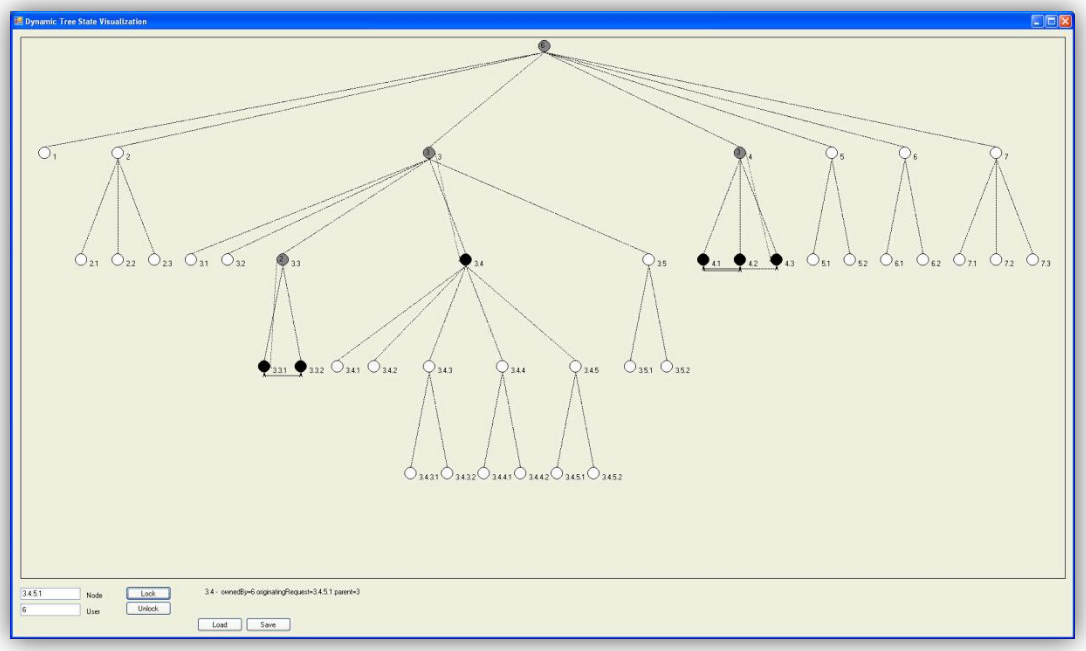
Service

The following operations are supported. For a formal definition, please review the Service Description.

- CheckIn
- CheckOut
- GetTreeState
- ListFiles
- ObtainLock
- ReleaseLock
- Validate

Figure 78: Web Service Implementation of Server API in ASP.NET

We have also developed visualization tools that display the document tree in a graphical view and display the state of each node and references among the nodes. This tool was originally developed to assist in verifying the correctness of our algorithm in seeing how various actions of users affected the document tree state (i.e., visualizing locks being promoted and demoted). A snapshot of the visualization of the document tree state is shown in Figure 79.



3.4.5.1 Node

6 User

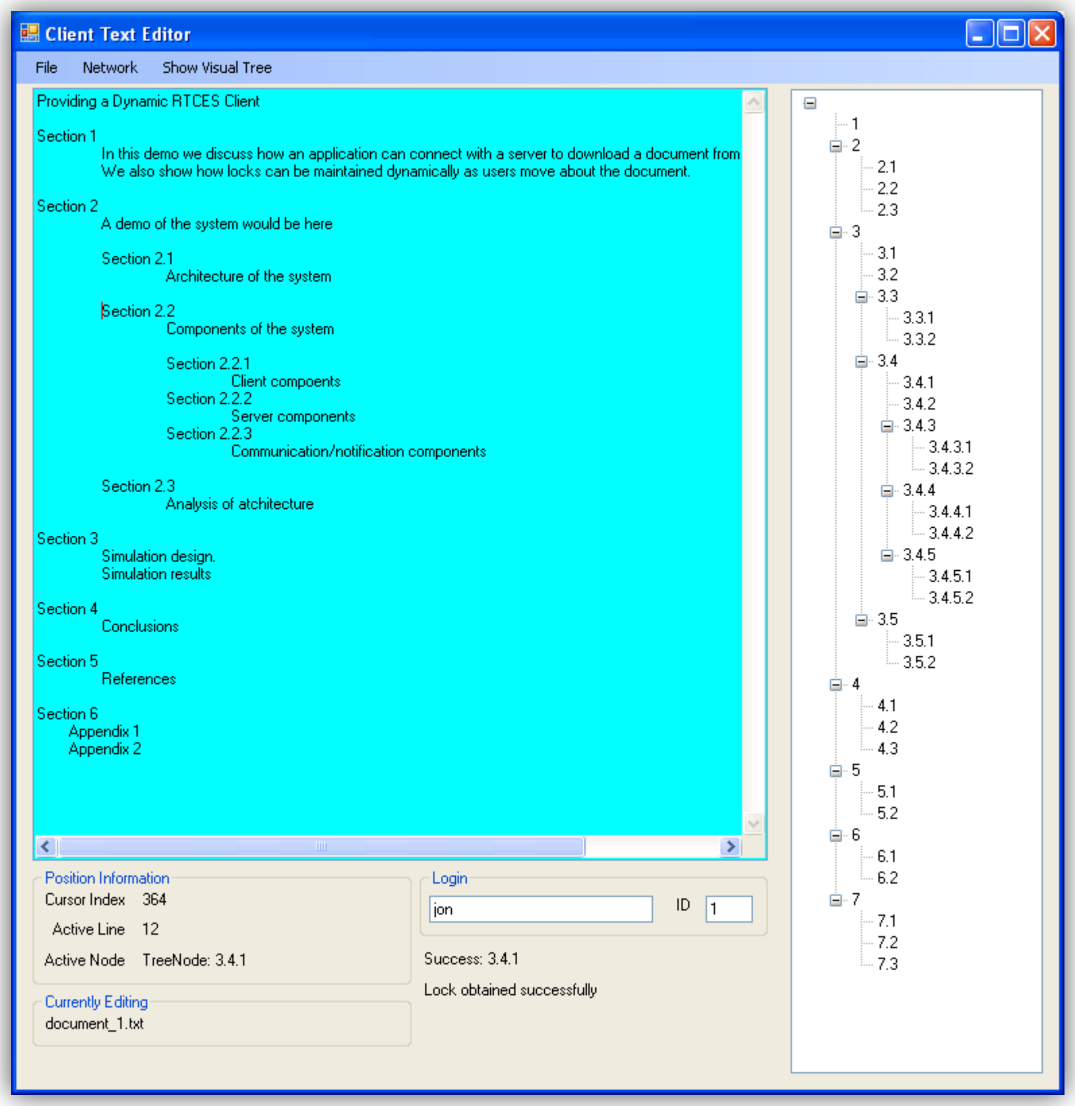
3.4 - ownedBy=6 originatingRequest=3.4.5.1 parent=3

Figure 79: Implementation of Visualizing the Document Tree State

Any client editing tool that can connect to a Web service API can make use of our client-server approach to document tree management and hook into the server technologies developed.

8.4. Implementing the Client

After the Web-services based server is implemented, we began development of a client application that connects to the server and allows multiple users to edit a shared document. The cursor position within the editor is tracked, and movement within the document automatically sends lock request and release messages to the server; as a result, clients are able to modify the shared document, and changes may be cached until a demotion message is received or the user leaves the space of the document that he owns. A preliminary version of this client editor application has been developed and is displayed in Figure 80. The dominant window (left) is the document's content, and the tree on the right shows the structure of the document tree based upon the document's content; the lower region shows state information such as in which section the cursor resides and displays messages from the server. The right-hand treeview control shows the structure of the document tree based upon the document content in the main editing window.



```
Providing a Dynamic RTCES Client

Section 1
        In this demo we discuss how an application can connect with a server to download a document from
        We also show how locks can be maintained dynamically as users move about the document.

Section 2
        A demo of the system would be here

        Section 2.1
                Architecture of the system

        Section 2.2
                Components of the system

                Section 2.2.1
                        Client compoents
                Section 2.2.2
                        Server components
                Section 2.2.3
                        Communication/notification components

        Section 2.3
                Analysis of atchitecture

Section 3
        Simulation design.
        Simulation results

Section 4
        Conclusions

Section 5
        References

Section 6
    Appendix 1
    Appendix 2
```

Cursor Index 364
Active Line 12
Active Node TreeNode: 3.4.1

jon
ID 1

Success: 3.4.1
Lock obtained successfully

document_1.txt

```
1
2
    2.1
    2.2
    2.3
3
    3.1
    3.2
    3.3
        3.3.1
        3.3.2
    3.4
        3.4.1
        3.4.2
        3.4.3
            3.4.3.1
            3.4.3.2
        3.4.4
            3.4.4.1
            3.4.4.2
        3.4.5
            3.4.5.1
            3.4.5.2
    3.5
        3.5.1
        3.5.2
4
    4.1
    4.2
    4.3
5
    5.1
    5.2
6
    6.1
    6.2
7
    7.1
    7.2
    7.3
```

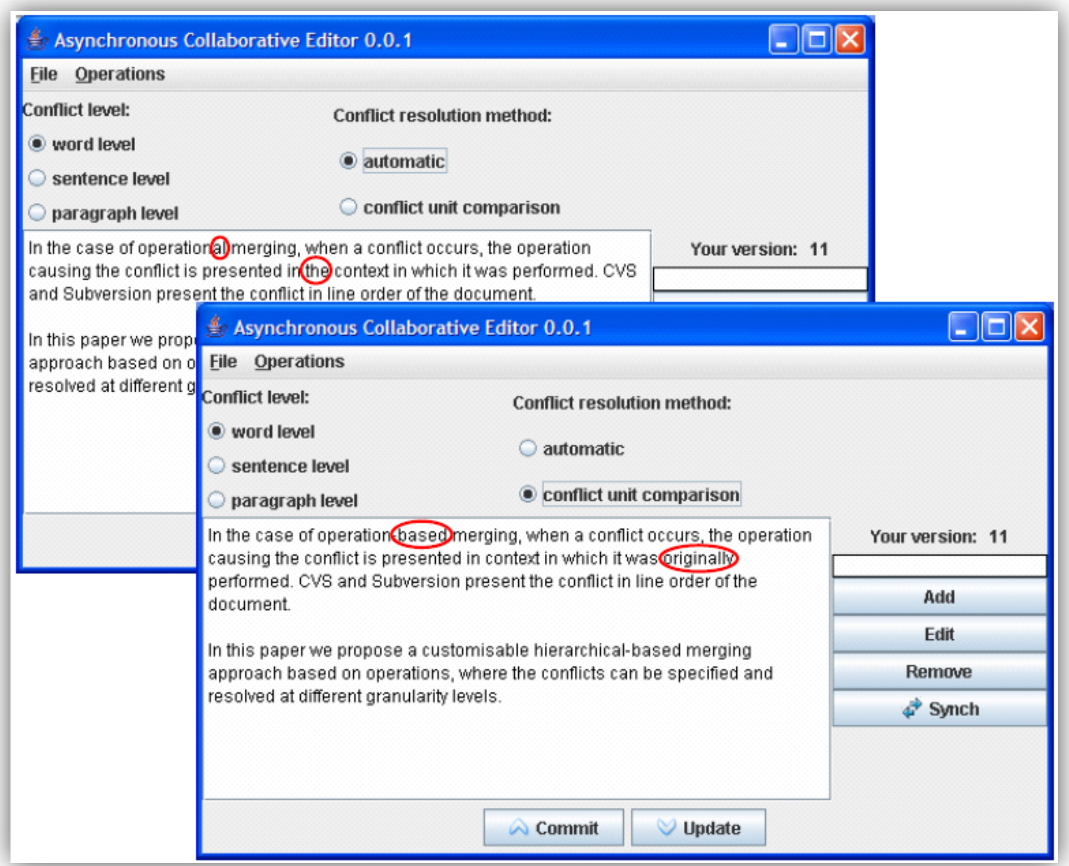
Figure 80: The Implementation of the Client Editor

While simple in nature, this client editing too demonstrates that any existing client technology can be extended (via hooks or other extension technology) to connect to our server API, or a new client tool can be developed to connect to the Web service API for document tree and dynamic lock management.

8.5. Discussion and Related Work

There have been many other systems that have implemented prototypes of RTCES editors [12], and these have been used to examine the efficiencies and correctness of various RTCES algorithms – primarily focused on OT-based algorithms to achieve CCI. Our approach as presented in this chapter has not been so much on creating new RTCES client and server technologies but rather focused on proving the viability of our preceding theoretical work in developing dynamic lock management algorithms to reduce communication and computation costs within a collaborative editing environment. This has been achieved using the simulation-based architectural design process – moving from simulated client and server models to implementations of a client and a server that validates our theoretical work.

Other recent, notable work in the area of prototypes of hierarchical management of document structures within RTCES include the work of Ignat [58] in allowing users to adopt merging of shared document content at a word, sentence, or paragraph level. This adjustable conflict resolution approach is demonstrated in Figure 81.



Conflict level: [✓ word level] sentence level paragraph level

Conflict resolution method: [✓ automatic] conflict unit comparison

Your version: 11

In the case of operational merging, when a conflict occurs, the operation causing the conflict is presented in the context in which it was performed. CVS and Subversion present the conflict in line order of the document.

In this paper we prop
approach based on o
resolved at different g

Conflict level: [✓ word level] sentence level paragraph level

Conflict resolution method: automatic [✓ conflict unit comparison]

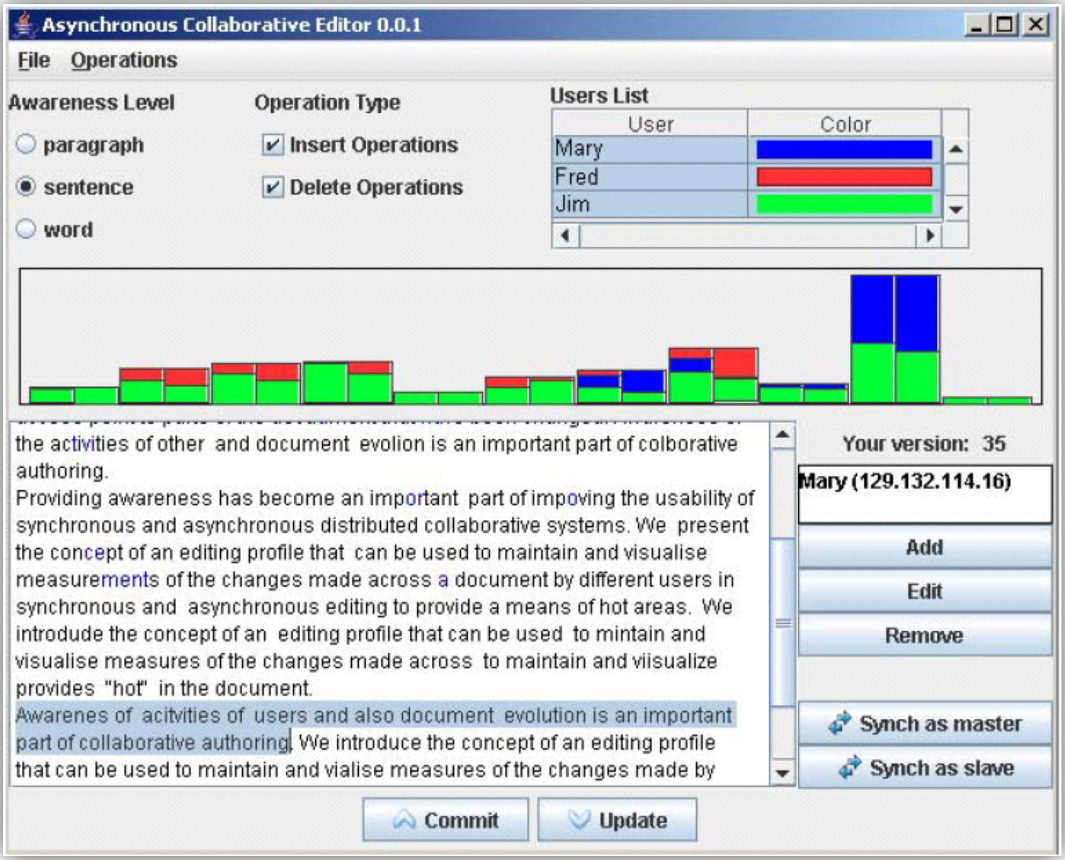
Your version: 11

In the case of operation based merging, when a conflict occurs, the operation causing the conflict is presented in context in which it was originally performed. CVS and Subversion present the conflict in line order of the document.

In this paper we propose a customisable hierarchical-based merging approach based on operations, where the conflicts can be specified and resolved at different granularity levels.

Figure 81: Adjustable Conflict Resolution [58]

Additionally, the work of [94] created visualizations (profiles) of changes made at various levels within a shared document – visualizing the changes at a word, sentence, and paragraph level – to provide meta-views of the changes that had been made to a shared document over time (see Figure 82); this interface provides an overview of the activity of other users with respect to the number and locality of changes within a text document.



Awareness Level

- [ ] paragraph
- [✓] sentence
- [ ] word

Operation Type

- [✓] Insert Operations
- [✓] Delete Operations

Users List

User	Color
Mary	🟦
Fred	🟥
Jim	🟩

the activities of other and document evolion is an important part of colborative authoring.
Providing awareness has become an important part of impoving the usability of synchronous and asynchronous distributed collaborative systems. We present the concept of an editing profile that can be used to maintain and visualise measurements of the changes made across a document by different users in synchronous and asynchronous editing to provide a means of hot areas. We introdude the concept of an editing profile that can be used to mintain and visualise measures of the changes made across to maintain and viisualize provides "hot" in the document.
Awarenes of acitvities of users and also document evolution is an important part of collaborative authoring. We introduce the concept of an editing profile that can be used to maintain and vialise measures of the changes made by

Your version: 35

Mary (129.132.114.16)

Figure 82: CES Document Profiling [94]

Both of these systems demonstrate that addressing the semantic structure of a document and how such semantic structure can enhance RTCES is an active area of research that offers potential and is currently being implemented in prototype systems.

The recent work of [139] shows that prototype systems are also useful in visualizing and managing the various operation scenarios employed in testing OT algorithms. Their time-space diagram (TSD) visualization tool allows a user to construct and manipulate operational scenarios (such as which operations are concurrent and which are causally-related) to see if CCI is achieved using various OT techniques.

8.6. Summary

Having developed an open systems based architecture to support a variety of client and server technologies within a RTCES, and having developed algorithms that support hierarchical locking that integrates existing best practices from OT-based research, we have further developed prototype client and server technologies that demonstrate the validity of our approach to supporting RTCES. Both our client and server prototypes presented within this chapter show that our approach is applicable to supporting scalable RTCES that minimize communication and computation costs.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Computing affords opportunities to enhance communication and collaboration; with the proper user-centric tools, many users can work together to solve ever more complex problems facing the world today. Inter- and intra-collaborations among researchers and business are ever increasing as ever more complex problems require interdisciplinary foci. Productivity software tools and other computing technologies are increasingly supportive of collaboration among multiple users, and as the pace of research and business increases, there will be an increase in the need for and the opportunities to support synchronous collaboration and editing of shared documents.

This research began by examining assumptions that the RTCES research community has not yet fully addressed. In doing so, we have begun to explore areas of RTCES that could be improved to be more scalable in supporting larger documents and larger communities of users. By focusing on intelligently caching changes and enabling dynamic hierarchical locking, we retain the highly responsive interactions that users expect with their local document editing tools. By focusing on integration of existing best practices with the OT research community, we leverage years of research to ensure consistency among replicas of the shared document. And by adopting an open systems approach, we support existing client and server technologies and leverage years of users' preferences and knowledge base.

We have shown that our dynamic locking algorithms are effective and efficient. By caching changes and selective multicasting among local writers, we have reduced communication and computation costs over existing OT broadcast schemes. And by distributing the document state among peers (P2P), we have avoided single server bottleneck latency and starvation.

9.1. A Systematic View of Real-time Collaborative Editing Systems

The CSCW and RTCES research communities have a rich history of algorithm and systems development. Ever increasing and effective techniques to achieve CCI have blossomed from the RTCES community within the past decade, and there shows much promise for the future of this field. The focus of this research has been to extend such promising research into a broader scope by integrating a systematic view of RTCES that includes an inclusive architecture, users' document replica state management (and thus caching), and communication and computation cost reduction. We believe that in looking at the larger picture of the system as a whole, new opportunities for improvements within the field of RTCES have emerged. Like an impressionist painting, certainly each brushstroke is vital and contributes to the whole picture; but by stepping back and viewing the problem from a systematic perspective, we have been able to see patterns of opportunity such as overall communication and computation efficiencies and opportunities for better intention preservation that heretofore have been hidden as the community's focus has been on paying attention to specific individual areas of RTCES research. We are pleased that our approach does not stand in opposition to or compete with the RTCES community's best practices, but rather integrate together with existing best practices of OT research in supporting an overall better system for supporting collaborative editing among multiple users.

In particular, we have achieved the following results:

1. An open systems architecture whereby exiting client technologies may connect with existing server technologies in supporting RTCES. Our approach uses a subscription model and Web services API to enable legacy and preferred technologies to be extended to support collaboration on shared documents in real time. We have empirically validated that the communication costs associated with our architectural approach are reasonable.
2. Algorithms and data structures that enable dynamic hierarchical locking of a shared document via a document tree such that users' changes may be cached when possible and selectively broadcast when multiple users are within the same section of the shared document. As a result of our approach, communication and computation costs are reduced when compared to an OT-only approach.
3. Integration of best-practices within the OT research community such that the CCI model is better achieved within localized subsets of the total client set and subsections of the shared document. Our results validate that we can provide concurrent access to all sections of the document to all users while still reducing communication and computation costs. Further, since we leverage semantic structure of the document, we are better poised to achieve intention preservation among users.
4. An extension of our client-server approach to dynamic, hierarchical lock management and integrated OT techniques into a P2P approach that distributes the document and lock state management among all users within the system. This P2P extension avoids a single point of failure and bottleneck at the server while improving reliability.

5. Preliminary, prototype implementations of both the client and the server technologies that validate our theoretical approach is viable and easily supported in actual, usable tools. These tools demonstrate that our algorithms can be integrated into existing applications or introduced into new applications to be built that support RTCES.

9.2. Future Work

Having developed a preliminary set of algorithms and approaches in support of RTCES, we look to how this work may be extended into the future.

Given that our algorithms are deadlock free, we could place the document tree on a multiprocessor machine and thread out the processing to avoid latency/starvation. While the focus of this research did not include this line of exploration, it would be interesting to see how our algorithms could be parallelized onto multiprocessor machines to achieve better real-time performance of handling the clients' requests.

It is our hope that our approach to supporting real-time collaboration may be applied within the distributed national and global research and business communities, and it is our intention to extend our research presented herein to facilitate collaboration among researchers. Since the main benefit of our approach is scalability of the number of users that can collaborate, it is logical that a large-scale research and development project would be well served by integrating our methods.

BIBLIOGRAPHY

[1] Arregui, D., Pacull, F., Willamowski, J.: “Yaka: Document notification and delivery across heterogeneous document repositories”. In: Proc. of CRIWG'01, Darmstadt, Germany (2001)

[2] 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.

[3] 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.

[4] Bharadwaj, V. and Reddy, Y. V. R., “A Framework to Support Collaboration in Heterogeneous Environments”, *SIGGROUP Bulletin*, (24) 3, Dec. 2003, pp. 103-116.

[5] Booch, G. Collaborative Development Environments. *Dr. Dobbs Journal*, Feb. 2007, pp. 10.

[6] Borghoff U. and Teege G. *Application of Collaborative Editing to Software-Engineering Projects*. ACM SIGSOFT, 18(3), pp. 56-64, July 1993.

[7] Borland JBuilder. http://www.borland.com/jbuilder.

[8] Bulgannawar, S. and Vaidya, N. A Distributed K-mutual Exclusion Algorithm. International Conference on Distributed Computing Systems, pp. 153-160, 1995.

[9] 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.

[10] Cederqvist, P. “Version Management with CVS”, Available from info@signum.se, 1993.

[11] Chawathe Y., McCanne S., and Brewer E. *RMX: Reliable Multicast in Heterogeneous Networks*. In Proc. IEEE INFOCOM, March 2000.

[12] Chen, D. A Survey of Real-Time Collaborative Editing Systems. In Proceedings of the Eighth International Workshop on Collaborative Editing Systems. ACM CSCW 2006, Banff, Canada. November 4, 2006.

[13] Cheng L. et al. *Jazz: A Collaborative Application Development Environment*. In Proceedings of OOPSLA'03, Anaheim CA, 102-103, 2003.

[14] Cheng L. et al. *Building Collaboration into IDEs*. ACM Queue. December/January 2003-2004. pp. 40-50.

[15] Cheng L. et al. Social Software Development Environments: Collaboration Within the Development. *Dr. Dobb’s Journal*. Feb. 2007. pp. 49-54.

[16] Chu-Carroll, M. C. and Sprenkle, S. “Coven: brewing better collaboration through software configuration management”, Procs. 8th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering: twenty-first century applications, Nov. 06-10, 2000, San Diego, p.88-97.

[17] Chu-Carroll, M. C., Wright, J, and Shields, D. Supporting Aggregation in Fine Grain Software Configuration Management. SIGSOFT 2002/FSE-10, pp. 99-108. November 18-22, 2002, Charleston, SC, USA.

[18] Chung., G and Dewan, P., "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.

[19] Conradi, R. and Westfechtel, B. *Version Models for Software Configuration Management*. ACM Computing Surveys, vol. 30, no. 2, pp. 232-282, June 1998.

[20] Davis, A. H., Sun, C., and Lu, J. Generalizing Operational Transformation to the Standard General Markup Language. Proceedings of CSCW 2002, New Orleans, Louisiana, USA. November 16-20. pgs. 58-67.

[21] Dekel, U. and Ross, S. 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.

[22] Dewan, P, "Architectures for Collaborative Applications", Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 169-193.

[23] Dourish, P., "Software Infrastructures", Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 195-219.

[24] Dourish, P. and Bellotti, V., Awareness and Coordination in Shared Workspaces, in Proceedings of the Conference on Computer Supported Cooperative Work (CSCW '92), R. Kraut, Ed. Toronto, Ontario, Canada: ACM Press, 1992, pp. 107-114.

[25] Drury, J. *Developing Heuristics for Synchronous Collaborative Systems*. In Proceedings of CHI'2001, pp. 447-448, March/April 2001.

[26] Ebersbach, A. et al. Wiki: Web Collaboration. Springer-Verlag. Berlin Heidelberg. October, 2005. pp. 9-32, 51-62.

[27] Eclipse. http://www.eclipse.org.

[28] Edwards, W. K. "Flexible Conflict Detection and Management In Collaborative Applications", Procs. 10th ACM Symp. on User Interface Softw. and Tech. (UIST'97). Banff, Canada. Oct. 14-17, 1997.

[29] Ellis, C. A. and Gibbs, S. J. 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.

[30] Eßmann, B; Funke, H: Providing Peer-to-Peer Features to Existing Client-Server CSCW Systems. In: Chen, Chi-Sheng; Filipe, Joaquim; Seruca, Isabel; Cordeiro, José editor. : Proceedings of the 7th International Conference On Enterprise Information Systems (ICEIS 2005), volume 4, S. 271-274, Miami, FL, USA, 24 - 28 May 2005 INSTICC

[31] Estublier, J. Defining and Supporting Concurrent Engineering Policies in SCM. Proceedings of the Tenth International Workshop on Software Configuration Management, 2001.

[32] Everitt, K. M, Klemmer, S. R., Lee, R., and Landay, J. A. 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.

[33] Feiler, P.H. Configuration management models in commercial environments. TechnicalReport SEI-91-TR-07, Software Engineering Institute, Carnegie Mellon University, 1991.

[34] Fu, S., Tzeng, N., and Li, Z. *Empirical Evaluation of Distributed Mutual Exclusion Algorithms*. International Parallel Processing Symposium '97. 1997.

[35] 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.

[36] Glance, N. et al, *Collaborative Document Monitoring*. In Proceedings of GROUP'01, Boulder CO, pp. 171-178, September 2001.

[37] Google Docs and Spreadsheets. http://docs.google.com/

[38] Greenberg, S. and Roseman, M., “Groupware Toolkits for Synchronous Work”, Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 135-168.

[39] 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.

[40] Grinter, R. E. Using a configuration management tool to coordinate software development. In Conference on Organizational Computing Systems, pages 168–177, 1995.

[41] Grinter, R. E. Recomposition: Putting It All Back Together Again. In Proceedings of CSCW’98, Seattle, Washington, USA, 1998. pgs 393-402.

[42] Groove Networks. http://www.groove.net.

[43] Grudin, J. *CSCW Introduction*. Communications of the ACM, vol. 34, no. 12, pp. 30-34, December 1991.

[44] Grudin, J. Cscw: History and focus. IEEE Computer, 27(5):19–27, 1994.

[45] 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.

[46] Gutwin, C. and Greenberg, S., “The Importance of Awareness for Team Cognition in Distributed Collaboration”, *Team Cognition: Understanding the Factors that Drive Process and Performance*, APA Press, Washington, pp. 177-201.

[47] Handley, M. and Crowcroft, J. Network Text Editor (NTE): A scalable text editor for the MBone, Procs. ACM SIGCOMM'97, pp. 197-208, Cannes, France, Aug 1997.

[48] Hofte, G. H. T., Working Apart Together: Foundations for Component Groupware, Telematica Instituut, The Netherlands, ISBN 90-75176-14-7, 1998.

[49] Handley, M. and Crowcroft, J. Network Text Editor (NTE): A scalable text editor for the MBone, Proceedings of ACM SIGCOMM'97, pp. 197-208, Canne France, Aug 1997.

[50] 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.

[51] Harrison W. H., Ossher H., and Sweeney P. F. *Coordinating Concurrent Development*. In Proceedings of CSCW'90, 157-168, October 1990.

[52] Herbsleb, J. D., Mockus, A., Finholt, T. A., and Grinter, R. E., 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

[53] Herbsleb, J. D. and Grinter, R. E. Architectures, coordination, and distance: Conway's law and beyond. IEEE Software, pages 63–70, 1999.

[54] 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.

[55] Hu, X. and Zeigler, B.P., "Model Continuity in the Design of Dynamic Distributed Real-Time Systems", *IEEE Transactions On Systems, Man And Cybernetics— Part A: Systems And Humans*, 35: 6, pp. 867- 878, November, 2005.

[56] Ignat, C., and Norrie, M. Draw-Together: Graphical Editor for Collaborative Drawing. In Proceedings of CSCW 2006, Banff, Canada, 2006, pp. 269-278.

[57] Ignat, C., and Norrie, M., Flexible Definition and Resolution of Conflicts through Multi-level Editing, Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing, Atlanta, Nov., 2006.

[58] Ignat, C-L., and Norrie, M. C., Flexible Merging of Hierarchical Documents, Procs of the Seventh Intl Workshop on Collaborative Editing, GROUP'05, Sanibel Island, Florida, Nov., 2005

[59] Ignat, C. and Norrie, M.C. Customizable collaborative editor relying on treeOPT algorithm. In Proc. of the European Conf. of Computer-supported Cooperative Work, pages 315-324, Sept. 2003.

[60] Ignat, C., and Norrie, M. Tree-based model algorithm for maintaining consistency in real-time collaborative editing systems, Procs. of the Fourth Intl. Workshop on Collaborative Editing, Computer Supported Cooperative Work (CSCW 2002), New Orleans, Nov. 2002.

[61] Ignat, C., Norrie, M., and Oster, G. Handling Conflicts through Multi-level Editing in Peer-to-peer Environments. Proceedings of the Eighth International Workshop on Collaborative Editing Systems. ACM CSCW 2006, Banff, Canada. November 4, 2006.

[62] 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.

[63] Kock, M. *The Collaborative Multi-User Editor Project IRIS*, Technical Report TUM-I9524, University of Munich, Aug. 1995.

[64] 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.

[65] Lamport, L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558 – 565, Jul. 1978.

[66] Li, R., and Li, D., A Landmark-Based Transformation Approach to Concurrency Control in Group Editors, GROUP’05, ACM Press, pp. 284-293, Sanibel Island, FL, Nov. 6-9 2005.

[67] Li, D. and Li, R. Preserving Operation Effects Relation in Group Editors. Proceedings of CSCW’04, November 6-10, 2004, Chicago, Illinois. pp. 457-466.

[68] 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.

[69] 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.

[70] Li, D., Zhou, L., and Muntz, R.R., A New Paradigm of User Intention Preservation in Realtime Coollaborative Editing Systems, In Procs. of the Seventh Intl. Conf. on Parallel and Distributed Systems, pp. 401-408, Iwate, Japan, July, 2000.

[71] Lin, Y-J. and Reiss, S.P. Configuration management with logical structures. In Proceedings of the 18th international conference on Software engineering, pages 298–307, Berlin, Germany, 1996. IEEE Computer Society.

[72] Locasto, M. et al. CLAY: Synchronous Collaborative Interactive Environment. The Journal of Computing in Small Colleges, vol. 17, issue 6, pp. 278-281, May 2002.

[73] Magnusson, B. and Asklund, U. 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.

[74] Magnusson, B. and Guerraoui, R. Support for Collaborative Object-Oriented Development. International Symposium on Parallel and Distributed Computing Systems (PDCS'96), Dijon, France, September 1996.

[75] Magnusson, B. 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.

[76] 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.

[77] Manhart, P. 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.

[78] Mens, T. A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering, 28(5), 2002, 449–462.

[79] 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.

[80] Microsoft Net Meeting. http://www.microsoft.com/windows/netmeeting/

[81] Microsoft SharePoint Services http://microsoft.com/windowsserver2003/technologies/sharepoint.

[82] Miles, V. C., McCarthy, J. C., Dix, A. J., Harrison, M. D. and Monk, A. F. Reviewing designs for a synchronous-asynchronous group editing environment. In Computer Supported Collaborative Writing Ed. M. Sharples. Springer-Verlag. 1993. pp. 137-160.

[83] Mills, K. L., “Introduction to the Electronic Symposium on Computer-Supported Cooperative Work”, ACM Computing Surveys, Vol. 31, No. 2, June 1999.

[84] Molli, P., Skaf-Molli, H., Oster, S., and Jourdain, S. Sams: Synchronous, asynchronous, multi-synchronous environments, The Seventh Intl. Conf. on CSCW in Design, Rio de Janeiro, Brazil, September 2002.

[85] Nesterovsky, A. and Nesterovsky, V. SCCBridge. http://www.nesterovsky-bros.com/html/css2/SCCBridge.htm. 2004

[86] Nickson, R. C. *A Taxonomy of Collaborative Applications*. http://hsb.baylor.edu/ramsower/ais.ac.97/papers/nickers.htm.

[87] Norman, D. Collaborative Computing: Collaboration First, Computing Second. Communications of the ACM. Vol 34, No. 12. December 1991. pgs. 88-90.

[88] O'Reilly, C., A Weakly Constrained Approach to Software Change Coordination. ICSE 2004: 66-68

[89] 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.

[90] Oster, G., Molli, P., and Urso, P., Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems, Proceedings of the 2nd

International Conference on Collaborative Computing: Networking, Applications and Worksharing, Atlanta, Nov., 2006.

[91] Oster, G., Urso, P., Molli, P., and Imine, A. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Banff, Alberta, Canada, November 2006. ACM Press.

[92] Osterweil, L. Software processes are software too. In Proceedings of the 9th International Conference on Software Engineering, pages 2–13, Monterey, CA, 1987.

[93] Pacull, F., Sandoz, A., and Schiper, A. "Duplex: A distributed collaborative editing environment in large scale", Procs of the ACM Conf. on Computer-Supported Cooperative Work (CSCW '94), 1994, pp. 165-173.

[94] Papadapoulou, S., Ignat, C., Oster, G., and Norrie, M., Increasing Awareness in Collaborative Authoring through Edit Profiling, Proceedings of the 2nd International Conference on Collaborative Computing: Networking, Applications and Worksharing, Atlanta, Nov., 2006.

[95] Papadapoulou, S. and Norrie, M., Document Profiling to Enhance Collaboration, 8th Intl. Workshop on Collaborative Editing Systems. Banff, Canada, 2006.

[96] Parnas, D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053–1058, 1972.

[97] Perry, D. E.., Siy, H. P. and Votta, L. G., "Parallel Changes in Large Scale Software Development: An Observational Case Study", International Conference on Software Engineering 1998, pgs 251-260.

[98] Perry, D.E., Siy, H. P., and Votta, L. G., Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Transactions on Software Engineering and Methodology, 2001. 10(3): p. 308-337.

[99] Pfleeger, S. L. *Software Engineering: Theory and Practice*. Prentice Hall, New Jersey, NJ, 1998, pgs 1-34.

[100] Prakash, A. "Group Editors", Computer Supported Cooperative Work, Edited by Beaudouin-Lafon, 1999 John Wiley & Sons Ltd, pgs 103-133.

[101] Pressman, R. S. *Software Ebgineering, A Practitioner's Approach: Sixth Edition*. McGraw Hill, Boston, MA, 2005, pgs 596-613 and 739-765.

[102] Preston, J. A. and Prasad, S. K., "Exploring Communication Overhead and Locking Policies in a Peer-to-peer Synchronous Collaborative Editing System", (Poster), ACM Southeast 2005, Kennesaw, GA, 2005.

[103] Preston, J. A. and Prasad, S. K. 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.

[104] Preston, J. A. and Prasad, S. K. "Achieving CCI Efficiently by Combining OT and Dynamic Locking with Lazy Consistency in a Peer-to-Peer CES", 8th Intl. Workshop on Collaborative Editing Systems. Banff, Canada, 2006.

[105] Preston, J. A. and Prasad, S. K. "An Efficient Synchronous Collaborative Editing System Employing Dynamic Locking of Varying Granularity in Generalized Document Trees", Proceedings of the 2nd International Conference on

Collaborative Computing: Networking, Applications and Worksharing, Atlanta, Nov., 2006.

[106] Preston, J. A. and Prasad, S. K., “Synchronous Editing via Web Services: Combining Heterogeneous Client and Server Technologies”, Proceedings of CSCW 2006, Banff, Canada, 2006.

[107] Preston, J. A. and Prasad, S. K., “A Web-Services-based Open-System Architecture for Collaborative Editing Systems”, Fourth International Conference on Cooperative Internet Computing, Hong Kong, China, 2006.

[108] Preston, J. A., Hu, X., and Prasad, S. K. “Simulation-based Architectural Design and Implementation of a Real-time Collaborative Editing System,” Proceedings of the 2007 DEVS Integrative Modeling and Simulation Symposium, Norfolk, VA, 2007.

[109] Qin, X. Delayed Consistency Model for Distributed Interactive Systems with Real-time Continuous Media, Journal of Software, Vol.13, No.6, pp. 1029-39, June, 2002, China.

[110] Qin, X., and Sun, C. Recovery Support for Internet-based Real-Time Collaborative Editing Systems, Proc. Intl. Conf. on Computer Networks and Mobile Computing , Oct. 2001.

[111] Rao, V. N and Kumar, V. Concurrent Access of Priority Queues. IEEE Trans. on Comput.. Vol 37, No 12. pp. 1657-65. 1988.

[112] Raynal, M. and Singhal, M. Logical time: capturing causality in distributed systems. IEEE Computer, 29(2):49 – 56, Feb. 1996.

[113] Ressel, M., Nitsche-Ruhland, D., and Gunzenh¨auser, R. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW'96*, pages 288–297, Boston, Massachusetts, USA, November 1996. ACM Press.

[114] Roh, H, Kim, S., and Lee, J. How to design optimistic operations for peer-to-peer replication. Proceedings of the Joint Conference on Information Sciences 2006. Taiwan. October 2006.

[115] Roth J. and Unger C. An extensible classification model for distribution architectures of synchronous groupware. 4th International Conference on Cooperative Systems. 2000.

[116] Roth, J. and Unger C. *Developing synchronous collaborative applications with TeamComponents*. 4th International Conference on Cooperative Systems. 2000.

[117] Sarma, A. and van der Hoek, A., "A Conflict Detected Earlier is a Conflict Resolved Easier", Proceedings of the 4th Workshop on Open Source Software Engineering, Edinburgh, United Kingdom, May 2004.

[118] Sarma, A., Noroozi, Z. and van der Hoek, A., Palantír: Raising Awareness among Configuration Management Workspaces . In Proceedings of Twenty-Fifth International Conference on Software Engineering, pp 444-454, May 2003, Portland, Oregon.

[119] Sarma, A., "A Survey of Collaborative Tools in Software Development", UCI, ISR Technical Report, UCI-ISR-05-3, March 2005.

[120] Sarma A., Noroozi Z., and van der Hoek A. *Palantír: Raising Awareness among Configuration Management Workspaces*. Proceedings of the 25th international conference on Software engineering, Portland OR, pp. 444-454, May 2003.

[121] Shen, H. and Sun, C. *Flexible Notification for Collaborative Systems*. In Proceedings of CSCW'02, New Orleans Louisiana, pp. 77-86, November 2002.

[122] Shen, H. and Cheong, C. T. CoStarOffice: Towards a Flexible Platform-independent Collaborative Office System. 6th International Workshop on Collaborative Editing Systems. Chicago, IL, USA, November 6, 2004.

[123] de Silva, N. Narratives to Preserve Coherence in Collaborative Writing. Proceedings of the Eighth International Workshop on Collaborative Editing Systems. ACM CSCW 2006, Banff, Canada. November 4, 2006.

[124] de Souza, C. R. B., David, R., and Paul, D., "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.

[125] de Souza, C. et al. How a Good Software Practice Thwarts Collaboration – The Multiple Roles of APIs in Software Development. Proceedings of SIGSOFT’04/FSE-12, Oct. 31-Nov. 6, 2004. Newport Beach, CA. pp. 221-230.

[126] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. *Chord: A scalable peer-to -peer lookup service for internet applications*. In Proceedings of the ACM SIGCOMM Symposium on Communication, Architecture, and Protocols (San Diego, CA, U.S.A., Aug. 2001), ACM SIGCOMM, pp. 149-160.

[127] SubEthaEdit. http://www.codingmonkeys.de/subethaedit/

[128] Subversion. http://www.tigris.org.

[129] Sun, C. and Chen, D. A Multi-version Approach to Conflict Resolution in Distributed Groupware Systems, Procs. of the 20th IEEE Intl. Conf. on Distributed Computing Systems, pp. 316-325, April 10-14, 2000.

[130] Sun, C. and Chen, D. "Consistency Maintenance in Real-Time Collaborative Graphics Editing Systems," *ACM Transactions on Computer-Human Interaction,* vol 9, no 1, March 2002. pgs 1-41.

[131] Sun, D. and Sun, C. Operation Context and Context-based Operational Transformation. In Proceedings of CSCW 2006, Banff, Canada, November 2006. pp. 279-288.

[132] Sun, S. and Ellis, C., "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.

[133] Sun, C., Jia, X., Zhang, Y., and Yang, Y. A Generic Operational Transformation Scheme for Consistency Maintenance in Real-time Cooperative Editing Systems, In Procs. of Intl. ACM SIGGROUP Conf. on Supporting Group Work, pp. 425-434, Phoenix, Nov. ,1997.

[134] Sun, C., Jia, X., Zhang, Y., Yang, Y., and Chen, D. "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.

[135] Sun, C., Jia, X., Zhang, Y., Yang, Y.`` 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.

[136] Sun, C., et al. CoOpenOffice: Converting OpenOffice into a Real-Time Collaborative Office Suite. Proceedings of the Eighth International Workshop on Collaborative Editing Systems. ACM CSCW 2006, Banff, Canada. November 4, 2006.

[137] Sun, C. and Sosič, R. ``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.

[138] Sun, D., Xia, S., Sun, C., and Chen, D. "Operational transformation for collaborative word processing," *Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work,* Nov 6-10, Chicago, IL USA. 2004

[139] Sun, C., Xia, S., Guo, J., and Sun, D., Using Time-Space Diagrams for Testing Real OT Systems, Proceedings of the Eighth International Workshop on Collaborative Editing Systems. ACM CSCW 2006, Banff, Canada. November 4, 2006.

[140] Sun, C., Yang, Y., Zhang, Y., and Chen, D. ``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.

[141] Sun, C., Zhang, Y., Yang, Y., and Chen, D. `` 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.

[142] Sunderam, V. et al. *CCF: Collaborative Computing Frameworks*. SC'98: High Performance Networking and Computing Conference (Orlando, Florida USA). IEEE. 1998.

[143] 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

[144] Tanenbaum, A. S. and van Steen, M. Distributed Systems: Principles and Practices. Prentice Hall, New Jersey, 2002, pp. 291-360.

[145] 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.

[146] 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.

[147] van der Hoek, A., Redmiles, D., Dourish, P., Sarma, A., Filho, R. S., and de Souza, C., “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.

[148] van der Lingen, R. and van der Hoek, A. “Dissecting Configuration Management Policies”, Proc. of the International Conference on Software Engineering Workshops: Software Configuration Management 2001.

[149] Visual Studio. http://msdn2.microsoft.com/en-us/vstudio/default.aspx.

[150] Xia, S., Sun, D., Sun, C., Chen, D. and Shen, H. “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.

[151] Younas, M. and Iqbal, R. “Developing Collaborative Editing Applications using Web Services”, The Fifth International Workshop on Collaborative Editing, ECSCW 2003, Helsinki, Finland, September 15, 2003

[152] Velazquez, M. *A Survey of Distributed Mutual Exclusion Algorithms*. Colorado State University Department of Computer Science Technical Report CS-93-116, September 1993.

[153] 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.

[154] 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.

[155] Walter, J. et al. A K-Mutual Exclusion Algorithm for Wireless Ad Hoc Networks. Principles of Mobile Computing '01. Newport, Rhode Island USA. 2001.

[156] 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.

[157] 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.

[158] Xia, S., Sun, D., Sun, C., Chen, D., and Shi, Y. Supporting Interactive Presentations with CoPowerPoint. 6th International Workshop on Collaborative Editing Systems. Chicago, IL, USA, November 6, 2004.

[159] Yang, Y., Sun, C., Zhang, Y, and Jia, X., Real-Time Cooperative Editing on the Internet, IEEE Internet Computing, pp. 18-25, May/June, 2000.

[160] Younas, M. and Iqbal, R. "Developing Collaborative Editing Applications using Web Services", *Proceedings of the 5th International Workshop on Collaborative Editing*, Helsinki, Finland, September 115, 2003.

[161] Zeigler, B. P, Praehofer, H., and Kim, T. G. Theory of Modeling and Simulation (Second Edition): Integrating Discrete Event and Continuous Complex Dynamic Systems. Academic Press. Amsterdam. 2000.

[162] Zeigler, B. P. and Sarjoughian, H.S. “Introduction to DEVS Modeling & Simulation with JAVA: Developing Component-based Simulation Models”, Technical Document, University of Arizona. 2003.