ENABLING TRULY COLLABORATIVE WRITING ON A COMPUTER

JOHN HAINSWORTH

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

APRIL 2006

© Copyright by John Hainsworth, 2006. All rights reserved.

Abstract

This study explores a new model for computer-based collaborative writing. A prototype application was built to implement this model, and user tests were performed on the application. The new model demands that each user have control of both the inflow of data from others and the outflow of data to others, and that the computer system handle the memory load required to track unmerged changes. The prototype achieves this goal by displaying to the user a persistent peer-to-peer multiversion set for each phrase in the document in which there is a conflict between contributors. This study presents the new model, describes the insights gained by building the prototype application, and presents the results of the user tests.

Acknowledgments

This work was partially supported by award number DGE-9972930 of the IGERT Program in Integrative and Computational Sciences, funded by the the National Science Foundation. I thank them for their support.

My thesis committee have all provided significant help to me. Dr. Andrea LaPaugh introduced me to and guided me through the world of databases and data mining. Dr. David Dobkin provided me with significant insights about how he collaborates. Dr. Larry Peterson and Dr. Brian Kernighan provided many pointers and references. And finally Dr. Perry Cook, my advisor and guide, gave me enough rope to forge ahead into a totally new area, but not quite enough to hang myself.

I particularly thank Dr. Greg Kochanski for his tireless efforts in testing my prototype.

Many other people have helped me throughout this project. Dr. Otto Anshus spent hours at a time talking with me in the early stages of this project, when my ideas were so vague that our conversations had a more impressionistic than concrete flavor. Dr. Charles Halcomb, Dr. Barbara Chaparro, and Dr. Eszter Hargittai were all instrumental in helping me to design and run user tests. I would also like to thank Dr. Amit Chakrabarti my theory consultant, Dr. Adam Finkelstein, and Dr. Ed Felten for helping me along the way.

I am also grateful to my mentors from my summer internships – Dr. Steve Whittaker at AT&T, Dr. Eric Baum and Igor Durdanovic at NEC, and Dr. Annette Adler, Leigh Klotz, David Hirsch, Mitch Garnaat, and Dr. Richard Burton at Xerox – for teaching me how research is done in a laboratory setting.

Finally, I thank my wife Deirdre and my sons Liam and David for encouraging me and giving me the strength to stick with this project to the end.

John Hainsworth

I dedicate this thesis to my wife Deirdre, with love always.

Contents

Abstract iii

List of Figures xi

List of Tables xiii

1 Background **1**

1.1 Problems With the Collaborative Editing Model 2

1.2 Efficiency Considerations 3

1.3 Why Collaborative Writing is Not Done Today 5

1.3.1 Challenges of Writing 6

1.3.2 Challenges of Collaboration 7

1.3.3 Additional Challenges of Collaborative Writing 9

1.4 Versioning 9

1.4.1 Single-Version Systems 10

1.4.2 Time-Sequenced Versioning 10

1.4.3 Accumulative Systems 11

1.4.4 Multiversioning Systems 12

1.5 Other Collaborative Systems 13

1.5.1 Collaborative Meeting Software 14

1.5.2 Accumulative Knowledge Bases 15

1.5.3 Collaborative Hypertext Systems . . . 15
1.5.4 Collaborative Object-Based Graphics Editors . . . 18
1.5.5 Source Code Control Systems . . . 19
1.5.6 Collaborative Editing Systems . . . 24
1.6 Summary . . . 27

2 The *CASTER* Collaborative Writing System 28
2.1 Design Principles . . . 29
2.1.1 Giving the User Control of Collaboration . . . 29
2.2 User Interface Implementation . . . 32
2.2.1 Startup and General Appearance . . . 32
2.2.2 The Add Operation . . . 34
2.2.3 The Replace Operation . . . 36
2.2.4 The Delete Operation . . . 40
2.2.5 The Move Operation . . . 43
2.2.6 Mechanisms for Building Consensus . . . 43
2.3 Implementation . . . 45
2.3.1 Database . . . 45
2.3.2 Overall Configuration . . . 50
2.4 Summary . . . 51

3 User Tests and Results 52
3.1 Procedure for Controlled Tests . . . 53
3.2 Interview Results from Controlled Tests . . . 57
3.3 Observations During Controlled Tests . . . 58
3.3.1 Patterns of Sharing . . . 58

3.3.2 Learning to Use the Chooser Dialog Box 61
3.3.3 Learning Curve Effects 61
3.3.4 Getting Started 61
3.3.5 Narrative Style Versus List Style 63
3.4 Measurements of Inclusivity by Character Counting 63
3.5 Questionnaire Comparison for Controlled Tests 64
3.6 Procedure for Shakeout Test 67
3.7 Results from the Shakeout Test 67
3.8 Procedure for Iterative Trials 69
3.9 Results From Iterative Trials 70
3.9.1 Word Processor Features 70
3.9.2 Use of Color 72
3.9.3 Text Segmentation 77
3.9.4 Design and Implementation Issues 83

4 Future Directions and Conclusions 87

4.1 Further Testing 87
4.2 Improvements to the Collaborative Writing Program 88
4.2.1 Content Subset Browser 88
4.2.2 Feature Refinements Indicated by Testing 89
4.2.3 Alternate Communication Channels 90
4.2.4 Combination With an Outliner 91
4.3 Applying the *CASTER* Paradigm Beyond Collaborative Writing 92
4.3.1 Hierarchical Document Content 92
4.3.2 Data Repositories That Never Achieve Consensus 93

4.3.3 Data Sets Accumulated by Store and Forward 94
4.4 Extensions to the *CASTER* Collaborative Model 96
4.4.1 Bulk Decision-Making 97
4.4.2 Decision Support Information 99
4.5 Summary and Conclusions 101

A Database Design 103

A.1 Database Design in *SQL* 104
A.1.1 Containment Tables 105
A.1.2 Content Tables 105
A.1.3 Building the Display List 107
A.1.4 Controlling the Visibility of Content 108
A.1.5 SQL table definitions 109
A.2 Update Handling 110
A.2.1 Commit Processing 111
A.2.2 Reading the Database Into an Application 113
A.3 Possible Enhancements 115
A.3.1 A Segmentation-Independent Approach 115
A.3.2 Database Merge 118

B Test materials 121

B.1 Questionnaire for Controlled Test 128
B.2 Questionnaire Results for Controlled Test 136
B.3 Questionnaire for Shakeout Test 138
B.4 Questionnaire Results for Shakeout Test 143

Bibliography **144**

List of Figures

2.1 *CASTER* Startup Dialog Box 33

2.2 *CASTER* Text Entry Window 34

2.3 New Text Created by Another User (Bob) 35

2.4 New Text Created by the Current User (Alice) 35

2.5 A Conflict Section 36

2.6 Conflict Resolution Dialog Box 37

2.7 Conflicts Segmented by Individual Characters 38

2.8 Segmentation 38

2.9 A Sequence of Conflict Sections 39

2.10 Repudiation 41

2.11 Modifying Another's Version Repudiates One's Own 42

2.12 Display of Deleted Text 42

2.13 Database Organization for a Short Document 47

2.14 Determination of Relative Newness Relationships 50

3.1 First Writing Task 54

3.2 Second Writing Task 55

3.3 Sample Document Produced During User Tests 55

3.4 Consequences of Delayed Reading 60
3.5 Result of Frequent Reading 60
3.6 Chat Behavior at the Start of a Writing Task 62
3.7 Coloring by Author 76
3.8 Conflict Displayed Side by Side 82
3.9 Conflict Displayed Stacked Up 82
3.10 Conflict Displayed as HTML Table 82

A.1 Containment Structure 106
A.2 Inter-Table Dependencies 112

B.1 Overall Test Description 123
B.2 Consent Form for Controlled Tests 124
B.3 Consent Form for Shakeout Test 125
B.4 Description of Control System 126
B.5 Description of CASTER System 127

List of Tables

1.1 Number of Authors per Paper at Recent ACM CHI Conferences 2

2.1 Database Example Showing Newest Fields 49

3.1 Percentage of Document Contributed by Minority Author 65

3.2 Comparisons of System Quality Based on Questionnaire Responses . . . 66

3.3 Performance Comparison of Display List Datatypes 86

B.1 Questionnaire Data for Controlled Test 137

B.2 Questionnaire Data for Shakeout Test 143

Chapter 1

Background

Collaborative preparation of documents using computer-based tools is common in our society and getting more common. As an example, Table 1.1 shows the average number of authors for technical papers in one technical conference (ACM CHI) for the last ten years. The average number of authors is increasing, and the percentage of single-author papers is decreasing. However, the process by which collaborative documents are created today is not by collaborative writing: rather, this process is better described as non-collaborative writing followed by collaborative editing. More specifically, collaborative "writing" today is a three-step process:

1. The group chooses a primary author.

2. The primary author writes the first draft of the document.

3. The group collaboratively edits the draft document.

The second step of this process, which comprises the actual writing, is not collaborative. The goal of this study is to explore the possibility of making this second step collaborative.

year	number of papers	average author count	percent single author
1995	76	2.64	13.16
1996	67	2.70	19.40
1997	76	2.64	18.42
1998	81	3.06	17.28
1999	78	3.33	12.82
2000	72	3.29	4.17
2001	69	3.39	11.59
2002	61	3.74	8.20
2003	75	3.55	6.67
2004	93	3.32	4.30
Increase per year	0.56	0.11	-1.49

Table 1.1: Number of Authors per Paper at Recent ACM CHI Conferences

1.1 Problems With the Collaborative Editing Model

There are at least two significant problems with the collaborative editing model described above: first, it creates disproportionate work for the primary author; and second, it gives disproportionate power to the primary author.

The primary author of a document must do disproportionate work when the writing is not collaborative, because he or she must do all of the actual writing. Furthermore, this primary author will often have to work under a lot of time pressure, because he or she is on the critical path: any time delays in document completion are totally the responsibility of this author until the first draft is complete. The primary author's workload is sometimes split up by assigning one chapter to each participant for initial writing, but this approach ossifies the document structure before the first word is written and also makes no provision for narrative flow and stylistic consistency.

The more significant problem with having a single draft author is that he or she acquires disproportionate power to determine the final contents of the document. The author of the first draft determines the structure of the document, the ordering of ideas, and the narrative pattern, which in turn dictates the relative importance accorded to those ideas. Furthermore, the group will pressure all editors to suggest as few changes as possible and as short changes as possible, because the amount of work required of the group during editing increases with the number and scope of changes made. Thus an idea whose expression requires rewriting a large document section may be effectively excluded from the process.

This power imbalance can lead to issues of injustice where competing interests are involved: for example, use of a draft real estate contract written by the seller and slanted toward the seller's interests places the buyer at a significant disadvantage in asserting his or her rights. This imbalance can also lead to significantly sub-optimal documents even when all collaborators are working to a common goal: the author of the first draft is not necessarily the best writer, the wisest, or the most knowledgeable. Furthermore, even if the primary author happens to be a reasonable choice, the constraints of the collaborative editing process greatly decrease the potential for the document to best leverage the strengths of all of its contributors.

1.2 Efficiency Considerations

In addition to problems with the quality of the resulting document, the collaborative editing process can be extremely time-consuming, tedious, and expensive. This section presents one case study from the research literature to illustrate how bad this process can get. The case study presented here, *Reaching Consensus on the* Tampa Bay Estuary

Program Interlocal Agreement*: A Perspective*, by Richard Eckenrod[18] describes a collaboratively written contract between government agencies, and provides an example of the effort and expense that can be required to achieve consensus on a collaborative document.

To begin with, this project had one exceptional resource: a champion who single-handedly solved the majority of the problems:

> The individual who championed the agreement not only was instrumental in overcoming obstacles encountered along the way, but also conceived the idea of the agreement, drafted the majority of it, and served as its principal advocate throughout the consensus-building process. His effectiveness as the champion was enhanced by his ability to appreciate the interests of all stakeholders in the process and to conceive compromises that preserved the integrity of the agreement. When obstacles were encountered, he wisely appointed task forces, with differing viewpoints appropriately represented, to work through the problem.[18, page 238]

The champion wrote a majority of the agreement; thus most of the actual writing was definitely not done collaboratively. Nevertheless, the group still needed a considerable array of other professional services to complete its collaborative document. One of the these services was professional facilitators, who were used extensively:

> Identification of stakeholder interests through interviews with individual parties helped identify most significant issues and avoid pitfalls late in the process. Development of a detailed framework for agreement based on the interviews helped organize and focus the discussion on key unresolved issues.[18, pages 236-237]

Further increasing the expense, the process was also highly dependent on attorneys. The report states that some of the sections of the document were drafted during meetings of groups of attorneys – all of whom were thus billing the project for the same time period.

Perhaps the most significant insight from this study is that it presents this entire process as a success story. The central theme of this analysis is that the (somewhat expensive) methods and mechanisms described here were considerably less expensive than they could have been.

1.3 Why Collaborative Writing is Not Done Today

This section will present some of the reasons why computer programs to enable collaborative writing have not been developed before now. First will be an examination of the prevalent assumption in much of the research community that truly collaborative writing is simply not possible. The remainder of the section will explore the challenges first of writing in general, then of collaboration in general, and finally of collaborative writing in particular.

Historically, the collaborative writing problem as described in the first section has been largely ignored by the research community in collaborative systems because this community has somewhat arbitrarily assumed that the problem is insoluble. The following excerpt from a book on collaborative writing clearly states the assumption that is implicit through much of this research literature:

> If we accept a definition of collaborative writing as the activities involved in the production of a document by more than one author, the pre-draft discus-

> sions and post-draft analyses and debate are the collaborative components. [14, page 84]

Because of this blind spot in the research community, the term "collaborative writing" has been co-opted in the literature to refer to the collaborative editing model. The remainder of this dissertation will refer to this approach as "collaborative editing" and will only use the term "collaborative writing" to describe systems that enable collaboration during the composition stage of writing.

1.3.1 Challenges of Writing

Writing is a difficult process. It involves finding words to represent ideas, crafting sentences to express those ideas, and applying both logical and narrative structure to organize those sentences – often all at the same time. And if the writing involves creativity, then the writer must go even a step further. This process requires as close to total concentration as possible. It is a solitary, individual process that works best when free of distractions. Anne Lamott describes the depth and spirit of this endeavor in her book on writing, *Bird by Bird*:

> "You need to trust yourself, especially on the first draft, where amid the anxiety and self-doubt, there should be a real sense of your imagination and your memories walking and woolgathering, tramping the hills, romping all over the place. Trust them. Don't look at your feet to see if you are doing it right. Just dance." [34, page 112]

It is also worth observing here that writing is not modular. A well written document will have a narrative flow, where each part of the document is designed to not only perform its primary purpose but also flow smoothly into the next section. A long document

or document section will also need to have summaries at the beginning and end, and signposts throughout to guide the reader. Changing any section thus may require changes to surrounding sections, and may require changes to introductory and summary sections as well. A well-written document should also have consistent style conventions, meaning that a style change in one section may require a cascade of other changes throughout the document.

Although the challenges of writing outlined here can be adequately addressed for a single author by traditional word processors – indeed, most solo authors today use a word processor for some or all of their writing – these applications are inadequate for multi-author collaboration. Furthermore these challenges of writing greatly complicate the process of collaboration during writing, in ways that will be explored below.

1.3.2 Challenges of Collaboration

Collaboration in general is also a challenging task. A collaborative system must ensure that each collaborator is sufficiently aware of the other collaborators' actions that he or she can effectively coordinate his or her efforts with theirs. The most demanding collaborative applications strive to create a continuous, subconscious awareness of the other participants called "presence". This awareness is the primary factor that determines how efficiently collaborative systems perform, and is established and maintained by a variety of media and techniques, which will be described below in the applications section.

Although preserving presence is important for collaborative systems, the most important key to their effectiveness is their role in helping their collaborators deal with disagreements and conflict. Collaboration is ultimately a process of merging. The starting point is always ideas within individual people's heads, and the end product is some form

of common result of the collaboration, and thus the ultimate effect of the collaboration is always to somehow merge these inputs into the end product.[1] There are two design parameters here: what degree of disagreement can be handled, and what role the collaborative system performs in resolving disagreements.

Collaboration between people who always think exactly the same thing is fairly pointless, because the reason that we put up with the inefficiencies of collaboration is to get the unique or best contributions from multiple participants. However, the intensity of disagreements that must be handled varies by the nature of the collaboration, ranging from the level of trust and gentleness of long-time collaborators to possible hostility, vandalism, and floods of unsolicited advertising in systems that are open to the general public. The most common cases are somewhere between these two extremes. Collaboration typically involves some degree of competition and personal conflict, but generally can rely on some level of external social structure or relationship to ensure that the disagreements do not get out of hand.

For a collaboration to complete, all of the conflicts between the contributions of the collaborators must somehow be resolved. Computer-based systems for different applications can take very different roles in this resolution process. At one extreme, the collaborative system can control the collaboration by dictating how all conflicts are resolved. At the other extreme, the collaborative systems can essentially take no role in conflict resolution, leaving the collaborators to negotiate their own compromises. In between, the system can support various levels of advising the participants and pushing them towards consensus.

[1]Designing a collaborative system to make these merging decisions automatically is a bad idea, regardless of how tempting it looks. Harris and Henderson make this argument effectively in *A Better Mythology for System Design*[26].

1.3.3 Additional Challenges of Collaborative Writing

Writing is not a simple or uniform process, and as a result there may be significant, document-wide disagreement about how the final document should look. These disagreements may take many forms. There may be disagreements in the writing style, as described earlier, including but not limited to notation and word choice, sentence structure, narrative flow, and logical organization. There may also be disagreements about the goals and purpose of the entire document, especially if it is being used for strategy or planning. These overall disagreements can filter down into smaller conflicts, so that there may be little or no agreement among the participants even as to what general types of changes should be considered improvements.

1.4 Versioning

The primary characteristic of a collaborative system that determines how it handles disagreements is its strategy for versioning: specifically, whether or not it allows multiple versions of conflicting sections of the collaborative product, and at what granularity these versions are accessible to the participants. There are four basic versioning approaches found in collaborative systems today. Single version systems allow only one version to exist. Time-sequenced versioning systems maintain older copies of the current version. Accumulative systems never delete anything, and thus keep all versions. Finally, multi-versioning systems present one or more active versions of the content simultaneously to the participants.

1.4.1 Single-Version Systems

Single-version collaborative systems are by far the most common. This approach mirrors how people normally work in non-collaborative settings: When an artist or writer without a computer makes a mistake, he or she paints over or erases it. In computer-based text creation with word processors, the operating system does not clutter up its hard disk with older versions, but rather discards them and keeps only the latest version.[2]

Collaborative systems that only keep a single version can be thought of as "latest only" systems: any conflict is (and must be) resolved by accepting the latest version presented and discarding anything with which it conflicts. Collaboration is achieved by allowing users to take turns accessing the artifact being created. In turn-taking or shared-cursor systems, the entire artifact is locked by the user currently modifying it. In multiple-cursor systems, each user can do whatever he or she wants, wherever he or she wants, at any time. A common compromise between these two extremes is region locking, where each user can lock and modify part of the collaborative artifact – a paragraph, a graphic object, or some similar segment – while still allowing other users to simultaneously modify other regions.

1.4.2 Time-Sequenced Versioning

The most common type of multiple versioning used in collaborative systems in time-sequenced versioning. This approach is so common that it is usually simply called "versioning" in publications of the research community. In a system using time-sequenced versioning, a new version of one component of the collaborative artifact is created when-

[2]The VMS operating system developed by Digital Equipment Corporation in the 1970's kept older versions of all files, and the *plan 9* prototype operating system also had a similar approach. However, no mainline operating system since has followed this model.

ever one of the participants locks that component, makes one or more changes to it, and then releases the lock.

The most widely used collaborative applications with time-sequenced versioning are the *RCS* and *CVS* version control systems, primarily used for the source code of computer programs. Both will be described in more detail in the applications section.

While a single-version system makes the implicit assumption that any change will always be an improvement, a time-sequenced versioning system assumes merely that any change will probably be an improvement. Thus in normal usage the user interface only shows the latest version of each module, but tools are available to access older versions in the relatively rare situations when they are needed.

A sequence of versions enables history tracking, which can occasionally be useful, but the most common use of versioning is to allow changes to be undone. Such undo operations when done directly using the version database are limited in two ways. First, all or none of the changes between a newer version and an older version must be undone. Second, versions can only be uncreated in the reverse order in which they were created. To work outside of either of these constraints – to undo parts of a version or to undo a version out of order – will require some manual text modification. Some time-sequenced versioning systems have tools to make complex undo operations easier, but these tools themselves can be difficult to learn, understand, and use.

1.4.3 Accumulative Systems

An accumulative system is essentially a system without a delete function – it displays all contributions from all users to all other users. The most widely used system of this type is the Usenet newsgroup system, in which anybody with Internet access can contribute a message to any open newsgroup. Such a system can be thought of as "all versions"

versioning, because the only way to change something in an accumulative system is to make a copy of the "old" version, make your changes to it, and write the "new" version into the system next to the old one. Since every older "version" is displayed, even if all participants want it gone, an accumulative system is a very noisy and distracting way to attempt to write.

Most accumulative systems support threading, which is implemented by allowing each author to submit his or her contribution as a reply to somebody else's contribution. A thread is formed by a contribution, a reply to it, a reply to that reply, and so on. Threads can branch if there are multiple replies to one contribution. Display interfaces in such systems usually automatically detect threads, organize them, and display them as tree structures. Threading might seem to naturally support time-sequenced versioning, but in practice most replies are comments about the previous contribution rather than replacements of it.

1.4.4 Multiversioning Systems

Multiversioning is a term that appears to have first been proposed by Sun, Chen *et. al.*[49]. Multiversioning refers to versions that are alternatives – i.e., only one of them should appear in the final product – but are considered to be equally valid "peers" until some process chooses one of them at some later time.

The first application for which Sun and Chen used multiversioning was for support infrastructure rather than for a user interface. They built a multi-site collaborative editor called *REDUCE*[48, 49] that maintained one common view at all sites. To maintain good responsiveness while running over potentially slow network links, this system immediately reflected all text changes locally, before they had a chance to reach the central database. This approach meant that the central database might receive conflicting and

apparently simultaneous text modification commands from multiple sites – multiversions – in which case it arbitrarily picked one version and broadcast it to all the sites, along with appropriate undo commands for rejected versions. This approach occasionally caused some contributor's change to disappear a few seconds after it was made. This behavior occurred only rarely because the granularity of the versioned segments was small, and therefore was tolerated by the users.

Making multiversions visible within a user interface raises some additional challenges. For display, the granularity of the versions – the size of the smallest segment of the collaborative artifact that can be represented as multiple alternative versions – is crucial to the success of the interface. The segments must not be too big or too small, and the segments must feel natural to the user.

Allowing multiple versions to be present simultaneously creates an additional problem: how these versions can be merged into a finally collaborative product. As described earlier, collaboration is ultimately a process of merging multiple inputs into a final product, and therefore allowing multiversions to persist in a collaborative artifact is in essence a mechanism for delaying part of the collaboration. This mechanism in turn creates a requirement for some other mechanism to complete the merge by choosing one of each set of multiversion for the final product.

1.5 Other Collaborative Systems

Before considering the implementation of a new collaborative writing system, it is instructive to examine other collaborative systems designed for other tasks, to understand both how their tasks differ from collaborative writing and how their implementation techniques exploit these differences. This survey will consider collaborative meeting

support software, accumulative knowledge bases, collaborative hypertext systems, object based graphics editors, and source control systems for computer programming. It will then conclude with a survey of systems to support collaborative editing, the closest currently available substitute for collaborative writing.

1.5.1 Collaborative Meeting Software

Computer and communications technologies have long been used to bring distant people together for meetings by providing video, audio, and other links between the participants' sites, and studies have shown such a linkage to be effective by itself[52]. However, computers have also been used to facilitate the business of meetings by providing temporary common work areas for the collaborative use of the meeting attendees. One common meeting support tool is shared electronic whiteboard systems such as *Tivoli*[40]. These systems usually support only a single version of the display: new writing overlays or erases old writing, just like on a real whiteboard or bulletin board, respectively. The single-version approach is not a problem with such systems because the artifact created is temporary: the principal product of the meeting is the consensus of the participants. The whiteboard contents occasionally will be archived but usually will be simply erased when the meeting is concluded. Thus the participants need not be concerned with what is written, because it will probably never be seen again and they will certainly not be held responsible for its content.

Decision support systems such as *SIBYL*[35] and *gIBIS*[12] provide another type of meeting support. Again, conflict resolution is not a problem because the permanent output of the system is only the final decision, not the information supporting it.

1.5.2 Accumulative Knowledge Bases

Accumulative knowledge bases, at their purest, are simply implementations of accumulative databases, as described above. Such systems never need to deal with conflicting versions, but they pay a price in lack of coherence. This lack of coherence is not a problem in such systems because the users do not expect coherence.

An example of such a purely accumulative system is the Usenet news program, mentioned earlier. In this system, anybody with an Internet connection can post a message to a newsgroup, and every other reader of the newsgroup will see that message. Although Usenet news servers eventually delete old messages, many websites archive the entire newsgroup hierarchy, so all messages are essentially available forever. The Usenet system worked extremely well in the 1980's – before the World Wide Web, Usenet was the preferred place to find current technical information – but lately its newsgroups have become so cluttered with advertisements and posts by inexperienced users that today few people can afford the time and effort read an entire newsgroup. Instead, more people are relying on text searching to find useful information in newsgroups[42]. Similar systems on smaller scales, including Lotus Notes and various members-only discussion groups on the World Wide Web, are able to control their volume of content by limiting membership and ejecting uncooperative members. Such systems must rely on two factors to work: posters who sincerely attempt to play by the rules, and readers who are tolerant of a bit of disorder and confusion.

1.5.3 Collaborative Hypertext Systems

Hypertext systems are a newer type of mostly accumulative knowledge base that has been gaining popularity within the last few years. In a collaborative hypertext system the

participants collaboratively create a hierarchical structure and attach notes or articles of the nodes of the hierarchy.

Another widely cited early collaborative hypertext writing systems is *CoAUTHOR*[24], which used region locking in individual hypertext nodes.

Anja and Jörg Haake in 1993 created a collaborative hypertext system called *CoVer*[21], which supported multiversions (which they called Multi-state OBjectS, or MOBS) for leaf nodes in the hierarchy tree. The *CoVer* user interface allows one to browse the versions in a MOB object and compare or merge any two such versions, but does not support the display of multiversion objects inline within the context of surrounding material.[3]

However, hypertext systems can also support collaboration fairly well without multiversions. The relative modularity of their link and node structure (as opposed to a monolithic narrative document) helps collaboration by reducing the chance of collisions between possibly conflicting contributions.

Currently one of the most popular classes of collaborative hypertext systems are those that create and manage wikis. A wiki contains two types of objects, hierarchy nodes and content (or leaf) nodes. Any wiki user can create, delete, or replace any object of either type. The document residing at any point in the hypertext structure can consist of a single leaf node, although in most wikis each hypertext document can be divided into separate modifiable sections, each represented internally by its own leaf node.

Probably the largest active wiki is Wikipedia[4], a collaboratively built encyclopedia. The Wikipedia writing system was designed with the articles following a traditional single-version editing model. Wikipedia also developed a set of flexible and scalable

[3] This system was built on an earlier project called *SEPIA*[23] in which each node could be modified in a shared-cursor session.

moderation policies and procedures to place limits on version conflicts. These policies include:

- The “three revert rule”, which forbids any author from undoing another author’s changes more than three times in a 24-hour period,
- A procedure by which administrators can be recommended by non-administrators and appointed by a vote of administrators (so power can be distributed while still emanating from the founders), and
- A procedure by which administrators can suspend or ban disruptive authors.

In practice, disagreements about the hierarchy do not seem to be a problem with Wikipedia users. This is probably true because cross-references are allowed and thus the hierarchy can be redundant, and also because most participants agree about what an encyclopedia structure should look like. The collaboration on articles did not work out so well. The initial expectation of the Wikipedia designers was probably that contention would be rare because a small group of people putting together an entire encyclopedia would have plenty of articles to write without bothering to argue over details. This approach worked well early on, but as the Wikipedia grew more comprehensive it attracted a new generation of contributors who write high volumes of material, rewrite others’ articles much more aggressively, and are much less thin-skinned about having their own articles edited or rewritten by others. Aggressive collaborators even contributed a new term to the language of collaboration – “revert wars” – to describe the situation where two people with an intractable disagreement about an article repeatedly replace the other author’s version of the article with their own version. Many of the original pioneers of Wikipedia got very frustrated and offended by the new, rougher, less respectful collabora-

tors, and quit Wikipedia entirely. The following are a few direct quotes from the "Missing Wikipedians" web page in Wikipedia (keyword WP:MW) expressing this frustration:

> "It took me way too long to realize the underlying facts about the way Wikipedia works. In this libertarian anarchy, any process is only as functional as it's most dysfunctional participant. What that means is that in too many areas the inmates are in charge of the asylum." - GK

> "I've gotten tired of the English Wikipedia. It's gotten so big that I would have to spent most of my time reverting bad edits to pages on my watchlist, and that's not fun. I'll probably be back, but not until a stable branch is launched." - Dori

> "Fine. I give up. Delete everything I've ever written. There appears to be some personal thing here where if you're not part of the clique, you're not welcome. Goodbye." - Corvus13

Perhaps the most important lessons from Wikipedia are that its type of collaboration is a niche solution, not suitable for everybody, and that both its limitations and its niche were not apparent until after months of intensive use.

1.5.4 Collaborative Object-Based Graphics Editors

Object-based graphics editors have been a fruitful proving ground for collaborative research prototypes. Many such systems appear to have worked quite well, and have not been turned into products largely because there is little demand for collaborative graphics editing. Collaborative graphics editors work well because individual graphics objects provide convenient and totally independent segments for simultaneous editing

by multiple users. Sun and Chen developed such a graphics editor[11] on top of the *REDUCE* multi-site synchronization engine described earlier on page 12. Sun and Chen also developed a graphics editor that allowed out-of-sequence undo of previous editing operations[47]. Stewart *et. al.*[46] and Bier and Freeman[7] also developed multi-cursor graphics editors, where multiple users could draw simultaneously, and these systems worked well in user tests.

The *Tivoli* team at Xerox PARC also developed a two-site electronic whiteboard system[38] that implemented true multiversions. Here the modularity of graphic objects was exploited to build a user interface: in two-site *Tivoli*, editing an existing object caused the original object to be moved aside (and sometimes shrunk) and a new copy of the object produced for editing.

1.5.5 Source Code Control Systems

One of the most widely used classes of collaborative computer applications is source code control systems used in the development of computer programs. These systems have caught on for two reasons: first, there is a great need for collaboration in computer programming, and second because computer program source code lends itself well to collaboration.

There is a need for collaboration in programming because some programs – most notably operating systems – must be large, and therefore require a large number of people working on them. Furthermore, adding people to a project increases the communication and coordination (i.e., collaboration) overhead, so much so that at a certain point adding people to a project can slow it down, as described by Frederick Brooks in his famous essay *The Mythical Man Month*[10].

Writing computer program source code works better with collaborative tools than does writing narrative text. There are many reasons for this, including the following:

- **Order does not matter**: Reordering an entire module is never necessary – the compiler does not care. The same is not true of writing.

- **Style does not matter**: Programmers care about the clarity of presentation in source code only in support of later debugging and maintenance, and the audience for code reading is limited to programmers. Writing usually must be accessible to a wide range of audiences, placing much greater emphasis on style.

- **Modularity**: Programmers rarely work on the same feature simultaneously. Each programmer works on a small set of functions which he or she has been assigned (or, in free software, those in which he or she has an interest).

  Although all computer program code must be reachable through a relatively small programmer interface or user interface (which precludes programmers working and testing totally alone), in a well-designed program most of the source code to implement each feature is relatively independent from the code to implement other features. The implementations of two or more features rarely share lines of code, and furthermore they are often segregated into separate source code files as well.

  On the other hand, writing is not modular in many ways, as described earlier on page 6.

- **Shared goals**: Programmers generally agree on the overall goal of a program before they start coding together. With the exception of experimental prototypes, most major programming projects start with a planning stage where the desired function of the program is clearly specified.

In contrast, collaborating writers may disagree about things as fundamental as the purpose of the document.

- **Consensus as to what constitutes an improvement**: Programmers benefit from a shared, industry-wide consensus of what constitutes an improvement: if a change adds a feature or fixes a bug, few programmers would argue that such a change is not an improvement. In the absence of deliberate vandalism, most programmers' changes do vastly more good than damage, and therefore are usually accepted by default by all other project members.

These characteristics of programming allow for straightforward version merging mechanisms. To merge changes to different module files, the user can simply collect the latest version of every file. To merge changes to different lines in the same file, the system expresses one of the changes as an ordered sequence of line modifications (add, delete, or replace) and applies all of these line modifications to a version of the file containing the other change. The only remaining case, where two participants simultaneously change the same line of source code, is extremely rare, so it is considered acceptable if the tool handles this case poorly.

The most universally available version control system – and the one upon which most others are built – is named *RCS*[4] (which stands for Revision Control System) and was developed by Tichy in the 1970's [51]. In the *RCS* system, a versions file is created corresponding to each source code file in the program. This versions file contains all versions of the described source file. It contains an exact copy of the latest version of the described file, interspersed as necessary with line modification commands which can be

[4] *CVS*, which is probably the most popular version control system, is one of the systems built upon the concepts of *RCS*.

applied as an executable script to generate any previous version of the file. *RCS* is thus a time-sequenced versioning system.

The *RCS* content modification user interface contains two basic functions: check out and check in. When a user checks out a file, *RCS* gives the user a writable copy of the desired version of the file (usually the latest version) and records what version of the file the user chose to check out. The user changes the file, and then checks in a new version of it containing his or her modifications. When a file is checked in, the *RCS* system makes the checked-in file into the latest version, determines the set of line modification commands required to generate the checked-out version from the checked-in version, and then appropriately arranges these and all the existing line modification commands so that all other versions can be created.

If two users simultaneously make changes to the same line of code in the same version of the same source file, the *RCS* system accepts the change made by the first user who checks in and then rejects the change made by the second user when he or she checks in. The rejection takes the form of a re-checked out, non-functional version of the file containing conflict sections. This second user must fix the problem manually by modifying the file and then checking it in again. This merge interface is hard to use, but is tolerated by users on many projects because it occurs very rarely. Most significantly, there exist open source projects, within the Linux project and others, with thousands of developers per product who tolerate these merging inconveniences. On the other hand, many projects using *RCS*, especially in a corporate setting, have historically avoided this merging interface entirely by disallowing multiple simultaneous checkouts of the same source code file. For example, the author worked on five programming projects using source control at Digital Equipment Corporation in the 1990's, and only one allowed multiple simultaneous checkouts of source code files.

Since this merge interface is a possible candidate for other collaborative systems, it is instructive to examine some of what it wrong with it. It has many problems, including unfairness, uncontrolled risk, and unpredictability. This merge system is unfair because it arbitrarily places the burden of merging on the second person to check in, regardless of who that person is. If this person is the programmer most overloaded with critical tasks, he or she may waste a lot of time researching and merging with changes done by other programmers with much more spare time. On the other hand, the merge may be very risky if the second person to check in is a novice programmer who does not understand the change made by the other person. This merge system can also add a lot of unpredictability to the development process. While most check-ins take a matter of seconds, a merge may require time-consuming analysis to understand the other person's change. Since many organizations expect programmers to check in their changes at the end of the day for an overnight build, these merges may end up happening under severe time pressure, with subsequent increases in error rate. Thus the *RCS* merge interface is not a good model for conflict handling in more demanding applications.

RCS Version Comparison Tools

Because versions in *RCS* are stored as deviations from a reference version, writing version comparison tools is relatively straightforward. *RCS* version files are segmented by source code lines, so displaying conflict sections side-by-side is very natural – however, it requires a wide screen. Some users prefer displaying change sections stacked up. Many systems offer both options. Other systems such as the ediff feature in emacs show two synchronized windows displaying the alternative versions. All of these interfaces work well for comparing two versions, but do not scale as well to comparing three or more versions.

CVS Branching

CVS, or the Concurrent Version System[3], is a system that offers all of the functionality of *RCS*, and in addition allows variant branches. Each variant branch has its own latest version, which can be checked out and checked in independently of the "main" branch or any other branch. *CVS* also offers what is called a "3-way merge" to merge the latest version of a variant branch back into the main branch. In this case, the latest versions on each branch are treated as peers in the user interface – none is necessarily latest or best – and thus they are equivalent to multiversions.

1.5.6 Collaborative Editing Systems

Collaborative editing systems assist with that part of the process of document preparation that occurs after a draft of the entire document is in place.

Collaborative editing is an easier task to manage than collaborative writing. It need not involve creativity, and thus need not provide the freedom from distraction required for writing.

The remainder of this section will describe three approaches to computer-based collaborative editing systems: the "low tech" method using word processing applications and electronic mail, the "Track Changes" feature in Microsoft Word, and "shared cursor" systems for text creation and modification. These systems can all be used to implement the collaborative editing model described earlier: non-collaborative writing followed by collaborative editing.

Word Processors, Email, and Meetings

The most common way that collaborative editing is done today does not use specialized tools, but relies instead on general-purpose word processing applications and communications channels. First, the primary author writes a first draft of the document and sends it to the entire collaborative group via electronic mail. Group members then send back either plain-text lists of suggested changes or new versions of the document with their suggested changes incorporated. For large projects, one or more editing cycles may be done through a face-to-face meeting of all participants, with one person recording the edits desired by the group and integrating them into the final document.

Microsoft Word "Track Changes"

The Microsoft Word program from Microsoft Corporation contains a feature in its "Tools" menu called "Track Changes". Once this feature is turned on, all new text typed will be shown in red, and all text deleted will be shown in red with a strikeout line through it. Further versions can be created, and their changes will be represented by colors other than red. The same menu also contains a "Compare Documents" feature that allows the user to represent the differences between two documents in this colored and strikeout format. The text produced by this interface can get confusing to read if it is extensively edited because changes are recorded letter-by-letter, sometimes with no complete words or phrases to provide a readable context. On the other hand, this can be a useful interface for experienced and capable users. One group of administrators at Princeton University[15] uses this "Track Changes" feature to prepare sensitive documents such as contracts or job offer letters, often having up to five people successively

editing the same document, each using a different color. In this case the document history is important, so it is vital that the system record who made each change, when, and why.[5]

Shared Cursor Systems

A number of research groups have built prototype collaborative text editors using a single-version "shared cursor" approach. In these systems the users share a common or duplicated view of the document. As in normal writing every change is permanent, including deletions. Some systems use a "turn taking" approach, where the entire document is locked by one person currently changing it, and all other users must wait. At the other extreme, multiple cursor systems allow all participants to make changes simultaneously, with no locking. In between these extremes, some systems allow each user to lock a region, usually a paragraph, until that user's modifications are complete.

The most extensive such prototype was the *QUILT* system, [19, 36] developed at Bellcore in the early 1980's. This system was developed over at least a seven year period, and included many features. It supported all three types of locking described above. It also included other communications channels as part of the application: video, audio, annotations, and an early implementation of a chat window. The system achieved some success as a collaborative editor, but its intent was never to support collaborative writing without extra help:

> Only if a collaboration has a firm social footing can we expect Quilt to be a positive element in the attempts to write collaboratively.[19, page 36]

[5] It is also sometimes important for political and legal reasons that the document history of a contract *not* be available to the recipient. For this reason, these documents were exported to PDF format before being sent.

Another such system named *SubEthaEdit*[2][6] is a multi-cursor document editor built for the Apple Macintosh. This system also displays each author's contributions in a different color, with an author key always available onscreen.

1.6 Summary

Although many computer tools have been developed to help people with the process of assembling and refining documents, few allow collaboration while actually writing the text of a document. Those that could possibly support collaboration during writing, the shared cursor systems, have only been successfully used for collaborative editing of text that was written non-collaboratively. The next chapter will describe a collaborative model that supports collaboration during creative work and a collaborative writing system built using that model.

[6]formerly *Hydra*[6]

Chapter 2

The *CASTER* Collaborative Writing System

This chapter describes a set of design principles for a system to support true collaborative writing, and a prototype computer program that was built according to these principles. Both the model and the prototype program are named CASTER.[1] This chapter is divided into three sections. The first section develops the design requirements for a collaborative writing system. The second section describes the user interface of the prototype and discusses how it implements the design principles. The final section describes implementation strategies and details of the prototype that are not visible through the user interface.

[1] *CASTER* originally stood for Collaborative Author-Specific Text Editor with Repudiation. This acronym is obsolete because the *CASTER* writing system is emphatically *not* merely a text editor. Thus *CASTER* is currently merely a name for this project.

2.1 Design Principles

The primary design requirement for a truly collaborative writing system is that it must give the user control of the collaboration. This requirement can be deduced logically from the discussion in the previous chapter. To do so, one starts by considering the essentially opposite requirements of the two components of collaborative writing, collaboration and writing. Writing requires focus and concentration. Collaboration requires awareness of and engagement with the activities of other, which largely destroys focus and concentration. To get around this problem, collaboration must be temporarily shut off when the user is actually writing new material. Furthermore, if a user can be returned to collaborating mode by any action of anybody else, then this user must continually remain prepared to deal with unexpected interruptions – which is just about as bad for concentration as dealing with the interruptions themselves. Therefore each user must be able to both turn off collaboration and prevent others from turning it back on. [2]

2.1.1 Giving the User Control of Collaboration

Collaboration is enabled by communication, and communication occurs in two directions. This section will describe how and why the user can and should be given control of communications, and thus collaboration, in each direction.

One side of controlling communication with one's collaborators is having the ability to write in privacy, without others being able to see one's work until one judges it ready for viewing. While writing in public may not be a distraction in and of itself,

[2] Some members of the research community have arbitrarily assumed that the goal of giving the user full control of collaboration is unattainable. Hewitt and Gilbert[27], believe that interruptions are inevitable and unavoidable in collaborative work. Neuwirth et. al.[39] agree that the closeness of collaborative coupling is a parameter that varies widely between individuals and between groups, but they argue that the application can at best offer a few hard-coded modes from which the user can choose.

it is indirectly distracting because the writer must be concerned about writing anything embarrassing, impolitic, or otherwise unsuitable, even temporarily, for fear that others will see it. Fortunately, a private-writing feature is easy to implement: all that is needed is to delay sending messages to the central server until the writer chooses to send them. It is also easily understood when implemented in the user interface: The **Save** command from most traditional applications will be replaced by a **Share** command, which conceptually does the same thing.

The other side of controlling communication with one's collaborators is to have control of the incoming information from other collaborators. Controlling this communication requires two things: control over interruptions providing the information and control over the process of dealing with that information.

It is difficult to concentrate on a creative process like writing when one can be interrupted at any instant. If another contributor can share new material at any time, and collaborative presence would dictate that other participants must be informed at that time, then the only way these unpredictable interruptions can be avoided is to temporarily suspend collaborative presence. To avoid shutting off collaboration entirely, these interruption messages must eventually be delivered, and the only way to make the eventual time of delivery predictable is to place message delivery under the control of each participant. This requirement can be implemented relatively simply, by adding a **Read New** function in the user interface which will read all new changes from the central database whenever it is pressed. The information that a change has occurred should be made immediately available to the user in an unobtrusive fashion: but the specifics of the actual change should not be supplied until the user requests them.

The more difficult part of controlling incoming communications is keeping track of what has changed. The next paragraphs will demonstrate why this is necessary by

exploring the problems that result if the user must manually keep track of changes, and then describing what types of features are necessary to prevent these problems.

If a collaborative writing system does not show each user all of the changes made by other users, then each user must either understand and evaluate all changes by others immediately when they are presented or remember them for later processing. Neither option is good: both entail distractions that will encroach on the concentration needed for writing. If the user must process all changes immediately as they are presented, then the **Read New** command will entail an unpredictable and possibly large merging task every time it is used – much like the conflict resolution in *RCS*, except that it will happen often. This threat of an unbounded merging task will cause users to either "self-interrupt" by reading new changes often enough to keep the merging workload low (and thus never achieving concentration) or to dread integrating and avoid it as long as possible, essentially killing collaboration. Expecting the user to remember these changes for later processing is not much better: in this case, the user's memory load will provide a continuous background "worrying" distraction that will interfere with concentrating on writing.

The only way to avoid requiring the user to pay attention to changes in a collaborative writing system is for the system to remember all of these changes. The user must be able to view the changes in context, which means with other nearby changes as well as within the surrounding text. The user must also be able to conveniently choose among or further modify such changed sections.

The creators of the *sam* word processor[41] in the *plan 9* operating system (among others) assert that there are four types of text modification operations: add, replace, move, and delete. However, a functional interface for text creation and modification can be built with only two of these operations. In the design of this interface, only the add

and replace operations were directly implemented. A move operation is simulated by a delete operation followed by an add operation, and a delete operation is simulated by a replacement with nothing. Remembering an add operation requires marking the new text so that the user knows that it is waiting for consideration. Remembering a replace operation requires not only marking the new text, but also displaying the old, replaced text with the new text for comparison.

2.2 User Interface Implementation

To evaluate the feasibility of the design principles presented in the previous section, a prototype collaborative writing system called *CASTER* was constructed following these principles. This section presents this user interface and describes how it conforms to these design principles.

The first subsection shows the startup procedure and general appearance of the user interface.

The next four subsections describe the principal mechanisms for implementing the design principles: the implementation of the individual text modification operations add, replace, delete, and move. The last section discusses mechanisms by which the interface facilitates resolving disagreements and building a final single consensus document.

2.2.1 Startup and General Appearance

At startup the *CASTER* collaborative writing program presents a dialog box analogous to the **Open** and **New** dialog boxes in word processors, as shown in Figure 2.1. Although the five selectable items in this dialog box are necessary for the CASTER program, three can usually be specified by default values. The specific behaviors of this dialog box are not

necessary to implement the *CASTER* collaborative writing model, but rather were chosen to balance ease of use for novice users against the need for flexibility in user testing.

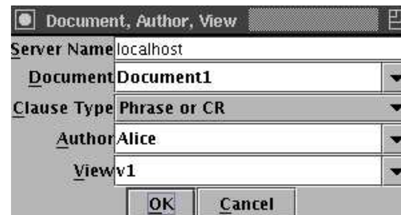

Figure 2.1: *CASTER* Startup Dialog Box

The five selectable items are:

- **Server name**: The ability to specify a non-local server address is necessary when two or more collaborators are using different computers. This field rarely needs to be entered: it can be set by an environment variable, and it defaults to “localhost”.

- **Document name**: This field is conceptually analogous to a filename. The user can either select an existing document from a pulldown menu or type the name of a new or existing document. This field has no default value.

- **Clause type**: This field allows the user to determine how the document will be segmented for the purpose of displaying conflict sections, which will be described below. It defaults to a segmentation strategy that has been empirically determined to work best for a document consisting of paragraphs of text.

- **Author name**: Each user must choose an author name to identify him- or herself when modifying a document. The user can either select an existing author name from a pulldown menu or directly type a new or existing author name. This field has no default value.

- **View name**: Each author may create multiple views of the same document. This feature is useful for making sense of large documents, but is rarely needed for short documents. The user can either select an existing view name from a pulldown menu or directly type a new or existing view name. This field defaults to “v1”.

In common usage, novice users will generally need to set only the document name and author name fields. The interface encourages this usage by its use of keyboard focus. The dialog box first appears with keyboard focus in the document name field. Pressing the “Enter” key in the document name field advances keyboard focus to the author field. Pressing the “Enter” key in the author name field advances keyboard focus to the OK button. The other three fields are still accessible via navigation with the “Tab” key or explicit selection with a pointing device.

Once the document is opened, a text entry window appears that looks much like a traditional word processor, as shown in Figure 2.2. The following sections describe the specialized ways in which *CASTER* handles traditional text modification operations.

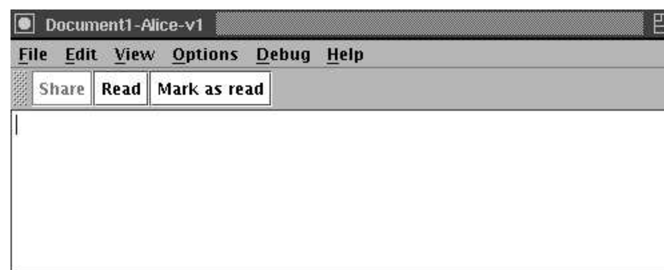

Figure 2.2: *CASTER* Text Entry Window

2.2.2 The Add Operation

The only requirement for the system to remember add operations is to mark any new text. The approach chosen for *CASTER* is to mark all new text created by others with a

yellow background, simulating the common process of using a yellow highlighter pen to mark changes in a paper document, as shown in Figure 2.3. The *CASTER* user interface implements a function called **Mark as Read**, which turns the background white for any yellow marked text in the current text selection. The current text selection is chosen in exactly the same way it is done for the cut and paste operations in most existing word processors and widgets. *CASTER* uses another common highlighter pen color, a light green, to mark new text created by the user of the program, as shown in Figure 2.4. The background in all of these green sections is changed to white when the user uses the **Share** function to write those changes to the central database.

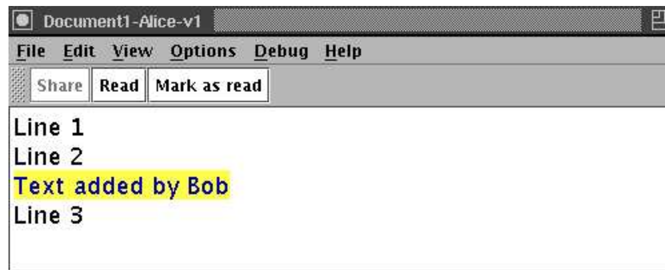

Figure 2.3: New Text Created by Another User (Bob)

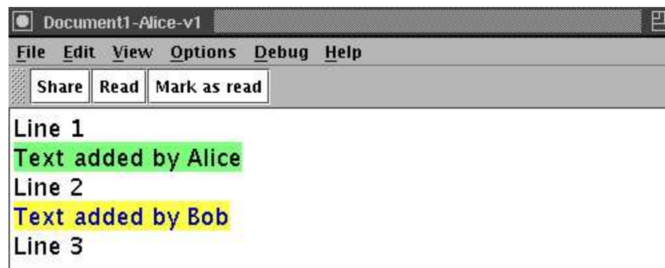

Figure 2.4: New Text Created by the Current User (Alice)

2.2.3 The Replace Operation

The user interface for implementing the replace operation was the greatest user interface challenge for *CASTER*. The old version of any text must be displayed, which is relatively straightforward at the expense of some screen real estate.[3] The bigger challenge is making the interface seem natural to users of both pen on paper and traditional word processors who are conditioned to expect old versions to be immediately hidden or destroyed.

The mechanics of the interface for handling text modifications requires support for displaying conflicting versions, choosing among them, or further modifying them. In *CASTER*, conflicting versions of a text segment are presented side by side, with delimiters around and between them, as shown in Figure 2.5. The delimiters have a red background to make them stand out. The user can choose between these versions by double-clicking on a conflict section. Double-clicking on a conflict section brings up a dialog box showing all active versions, as shown in Figure 2.6. The user can then select one of the displayed versions. Alternately the user can modify a version directly by clicking anywhere in a conflict section and then typing.

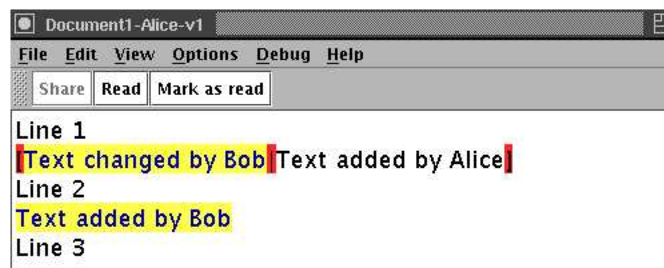

Figure 2.5: A Conflict Section

[3]The strikeout approach used in Microsoft Word was considered but rejected because it does not provide a clear cue to the user that this is a conflict section (between keeping and deleting the text). One possible project listed in the "Future Work" section is to evaluate this approach.

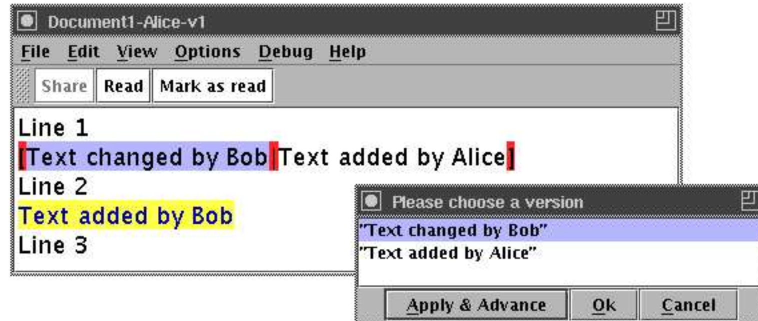

Figure 2.6: Conflict Resolution Dialog Box

The mechanics of this interface do not address two vital semantic questions: how the text should be segmented to create conflict sections, and which versions should be displayed within those sections.

Text Segmentation

To display conflict sections, the text must be broken into segments. These segments must provide sufficient context for comparing versions while not unduly interrupting the flow of the document. There are two design parameters for the size of these segments: how small the smallest sections will be, and how big the biggest sections will be.

Preliminary testing indicated that the smallest size of conflict sections should probably not be individual letters: the hypothetical example in Figure 2.7 shows how confusing this can get for even the simple modification of changing a sentence from plural to singular. The design choice for *CASTER* was to segment by phrase, where a phrase begins with text and ends with punctuation. Later testing showed that conflict sections containing both text and carriage returns were hard to understand, and that therefore each sequence of carriage returns should be its own segment. Figure 2.8 shows an example of this segmentation. Note in the figure that the segments containing carriage returns

contain no visible text. During conflict resolution, however, the version-choice dialog box represents each carriage return with the text "<BR>".

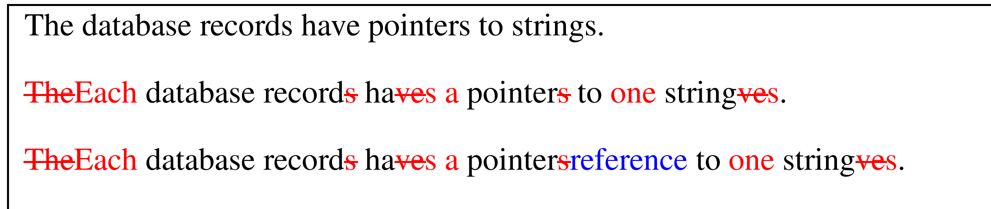

Figure 2.7: Conflicts Segmented by Individual Characters

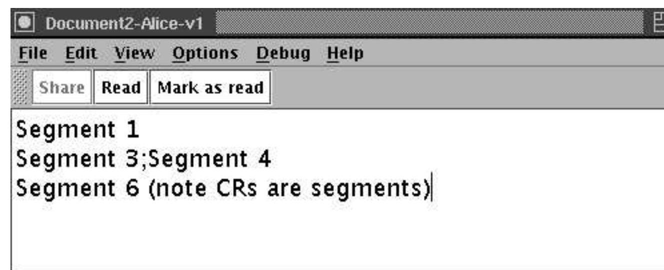

Figure 2.8: Segmentation

The design question concerning the maximum size of a conflict segment is whether multiple phrases should be combined into a single large conflict segment, or each phrase should be represented as its own conflict segment. The disadvantage of large conflict sections is that they may extend over multiple pages, making it difficult for the user to compare and choose. The disadvantage of using smaller segments is that a large text modification may create a long sequence of conflict sections, as shown in Figure 2.9. The design choice in the *CASTER* system was to limit each segment in a conflict section to a single phrase. To deal with possible long sequences of conflict sections, an **Apply and Advance** command was added to the conflict resolution dialog box, as shown in Figure 2.6. This command applies the current version choice and advances both the main text window and the contents of the conflict resolution dialog box to the next conflict section in the document. If possible, this update procedure will pre-select a sensible

default version in the new conflict section: this pre-selected version will be the one written by the same author as the version selected in the previous conflict section. Thus either accepting or rejecting all of a multi-section change can be accomplished by making a decision in the first conflict section and then repeatedly selecting **Apply and Advance**.

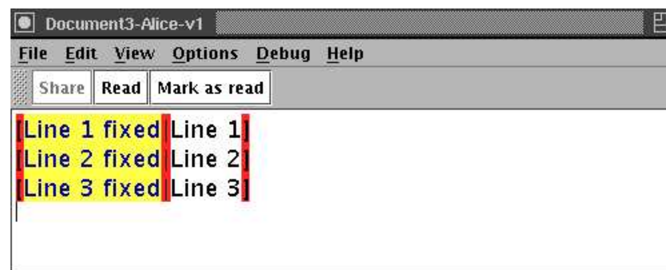

Figure 2.9: A Sequence of Conflict Sections

Choosing Which Versions to Show

Displaying multiple versions of the same text segment is distracting for editing. This distraction does not extend to the initial writing process, because the writer knows that he or she can deal with it later. For editing, however, multiversions make the text harder to read and understand as a unit. There is evidence that a shared view causes difficulties[45], but these difficulties are not insurmountable. On the other hand, allowing the user to eliminate something that he or she disagrees with from his or her private view effectively shuts off collaboration in that region, perhaps leading to significant divergence between participants' views, and a difficult and complicated merge process in the future.

Fortunately one mechanism exists to keep the number of versions manageable, which we call repudiation. If one participant modifies a segment that he or she also created, the system labels the old version of the segment repudiated, and eliminates it from display in the views of all participants. This mechanism is based on the idea that each author is implicitly the advocate for what he or she wrote, and thus a segment discarded by its

own author has no advocate. An author can also repudiate a version of a segment by choosing someone else's version for it. For example, if one participant writes a sentence containing a misspelled word, a second participant corrects the spelling, and the first participant chooses the corrected version of that segment, then the version containing the misspelled word is classified as repudiated by the *CASTER* program and removed from all participants' views. Figure 2.10 shows this process.

One consequence of the repudiation mechanism is that each participant is allowed to own at most one version in any conflict section. This leads to an unusual behavior when typing in a conflict section where the user already owns one version. If this user types into another user's version in that conflict section, then the first user's previous version disappears and is replaced by a modified copy of the version that was edited. An example of this is shown in Figure 2.11. The first illustration shows one version created by the current user and one created by another user. The second illustration shows the result of typing a single letter ("X") into the version created by the other user: The version previously created by the current user disappears and is replaced by the version created by the other user with the "X" added.

2.2.4 The Delete Operation

A change that deletes an entire segment is represented as a conflict section in which one version is an empty string. This empty string is represented by the placeholder string "<deleted>" in the conflict resolution dialog box, but it is represented by nothing in the main text, as shown in Figure 2.12.[4]

[4] Note that the *CASTER* program does not allow a user to delete a text segment within the conflict resolution dialog box unless some user has previously deleted that segment. This approach is logically consistent – this dialog box only allows a user to choose something that somebody else has already done. However, a user may entirely delete a segment that has never before been deleted simply by deleting all the content of any version within the conflict section.

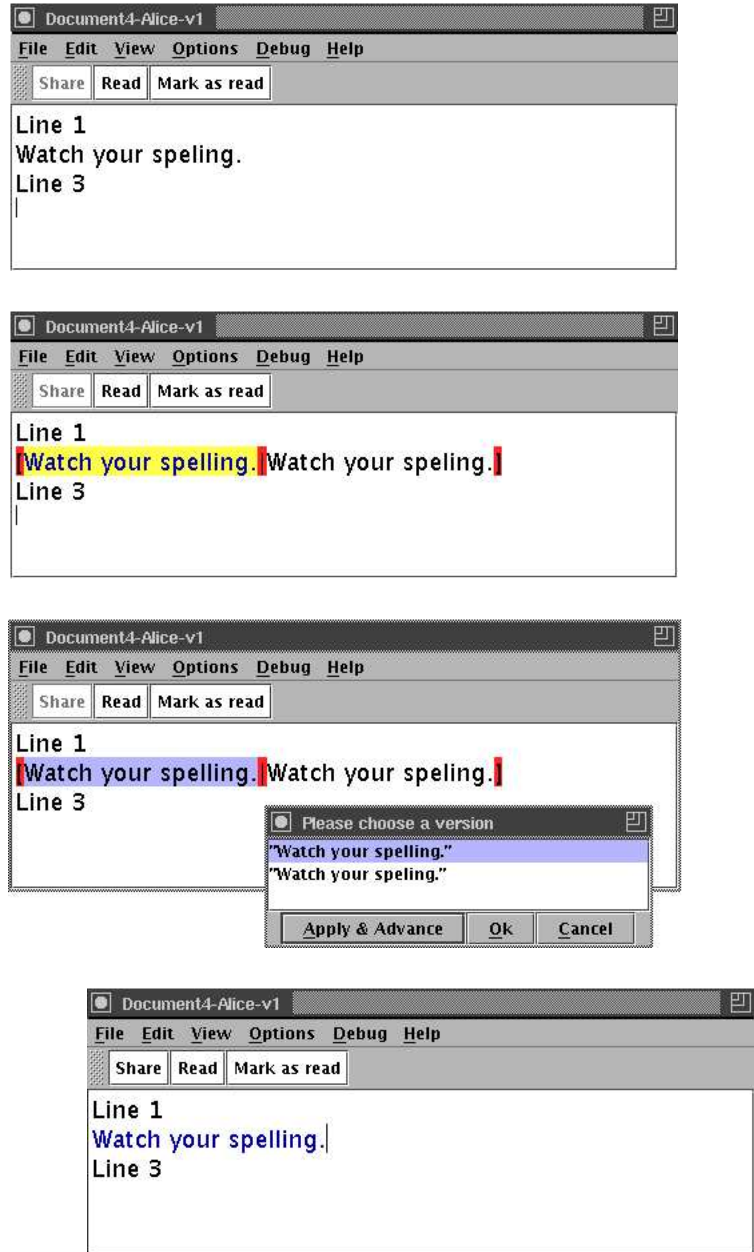

Figure 2.10: Repudiation

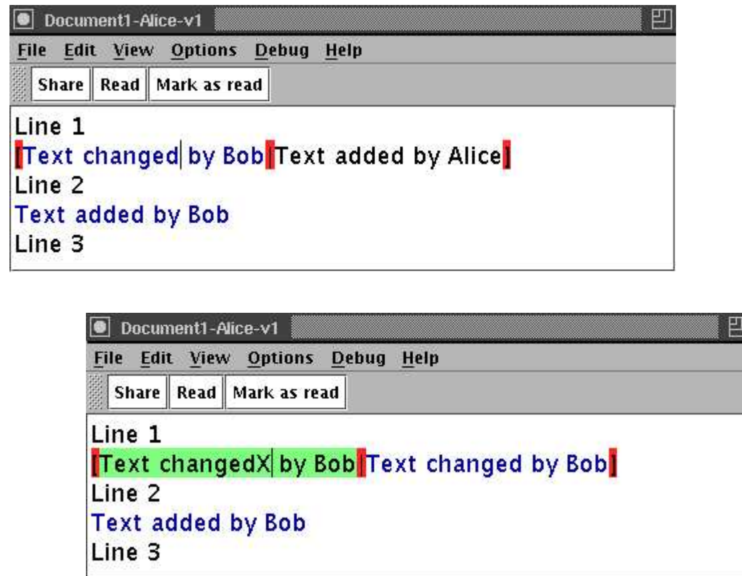

Figure 2.11: Modifying Another's Version Repudiates One's Own

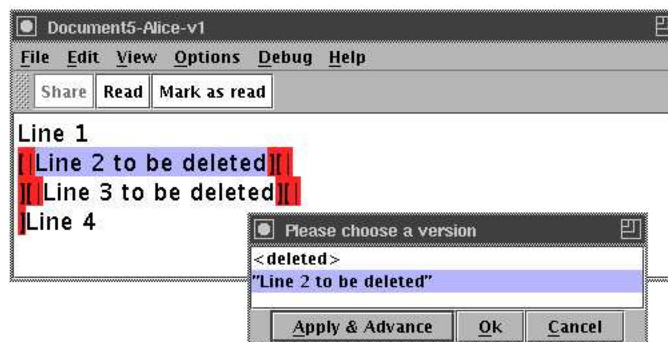

Figure 2.12: Display of Deleted Text

Deletion of a large region of text can result in a long sequence of consecutive conflict sections, which can be handled efficiently by the **Apply and Advance** feature in the conflict resolution dialog box, as described earlier.

2.2.5 The Move Operation

A design decision was made for the *CASTER* system not to implement an atomic move operation. Moving text in *CASTER* must be done by deleting it from one place and inserting it into another. This decision was made because the user interface complexity required to visualize a block of text in two or more contexts would have made the entire program more difficult to learn and use, and this feature was not deemed to be worth the additional complexity. (The file format used by the central *CASTER* database server has supported multiple locations for each segment from the beginning, but this feature was never used).

2.2.6 Mechanisms for Building Consensus

The ultimate goal of collaboration is to achieve a consensus result. In the case of collaborative writing, this result is one consensus version of the document being written. This merging of viewpoint is the final step of most collaborations. This goal conflicts with our design requirement that each participant have control over his or her own creations. The only mechanism that is compatible with user control for eliminating a conflicting version is repudiation, in which a version can be rejected by the same user who created (and thus controls) it. Thus if one wants to achieve a consensus document in the face of one or more intractable conflict sections – sections where no author can convince any other author to

change his or her mind – one must somehow compel one or more authors to repudiate their contributions, against their will.

Fortunately, the final merging process for collaboration need not consist of browbeating the participants into recanting their opinions. Merging does require someone with the power to override others' decisions, but this overriding is done simply by ignoring some of the viewpoints presented, without the authors' consent. Stripped to its essentials, consensus document building involves resolving all conflicts of opinion in the document, while doing no other writing or editing.

This process can be done within *CASTER* using a private view. *CASTER* can make such merging very efficient: the group doing the merging work need only discuss the conflict sections in the document, and the discussion of each conflict section can be limited to comparing the versions present in that conflict section. Specifically, the final merge process can be organized as follows:

- Each participant creates a private view of the document with which he or she is satisfied.
- The entire collaborative writing group meets (or joins in communication) to generate a merged document.
- The moderator or leader creates a new, "fresh" private view of the document, in which no personal choices have been made. This view shows every conflict which could not be resolved by the authors of the candidate versions in its conflict section.
- The group considers these conflict sections in order of occurrence, and the leader chooses one favored version from each using whatever decision-making mechanisms are consistent with the participants' organization: voting, persuasion, social pressure, fiat, etc.

- The contents of this new private view are taken to be the final merged document.

2.3 Implementation

This section will describe aspects of the design and implementation of the *CASTER* prototype collaborative writing system that are not visible in the user interface.

2.3.1 Database

This section will highlight features of the *CASTER* database that directly support the collaborative features described above. The database format is more fully presented in Appendix A.

Accumulative Format

The data format in which *CASTER* documents are stored is basically a sequence of text modification operations, much like that of the *sam*[41] word processor in the *plan 9* operating system. The database format is purely accumulative: nothing is ever deleted from the database, and the responsibility for hiding repudiated or otherwise hidden content lies with the application. The format is designed so that new data can always be stored by appending data to a single data file, with no forward references. Although the data is stored that way currently in the *CASTER* prototype, the data can also be stored in a relational database format – and was stored that way in early versions of the prototype. In the following explanations we will use the relational database model to describe the features because the terminology and concepts are more universally known.

Support for Multiversions

The document contents are stored in three record types: *Slot*, *AtomVersion*, and *SiteVersion*. Each clause in the document is represented by a *Slot*, but the *Slot* record does not contain any text data. The text data is instead stored in another type of record called an *AtomVersion*, which is linked to the *Slot*. When a user modifies a phrase, a new version must be created for that phrase. This new version of the text is stored in a new *AtomVersion* record, and this new record is linked to the same *Slot* record as the original *AtomVersion*. Thus *AtomVersion* records containing the text of every version that may appear in a conflict section – and possibly many versions that will remain hidden – will all be linked to one *Slot*. Each *SiteVersion* record represents a position in the document at which a text segment might appear, and this position is specified by a link to another *SiteVersion* record called its anchor. This *SiteVersion* record also links to a *Slot* record to indicate what text will be shown at that position. Thus the document can be thought of as an ordered list of *SiteVersion* records, each pointing to a *Slot* that describes what will be shown at that position. The purpose of *SiteVersion* records is to support move and copy operations, in which the same text segment can appear in more than one location. The *CASTER* prototype writing system does not support move or copy operations through its user interface, so the *Slot* and *SiteVersion* records in a *CASTER* document are always in one-to-one correspondence.

Figure 2.13 shows a sample document and the conceptual organization of its underlying database. For the purpose of this illustration, support is assumed for a copy-text feature not found in the *CASTER* prototype. Additionally, the connections between *SiteVersion* records are simplified in this diagram. Appendix A contains a full specification of the exact database format.

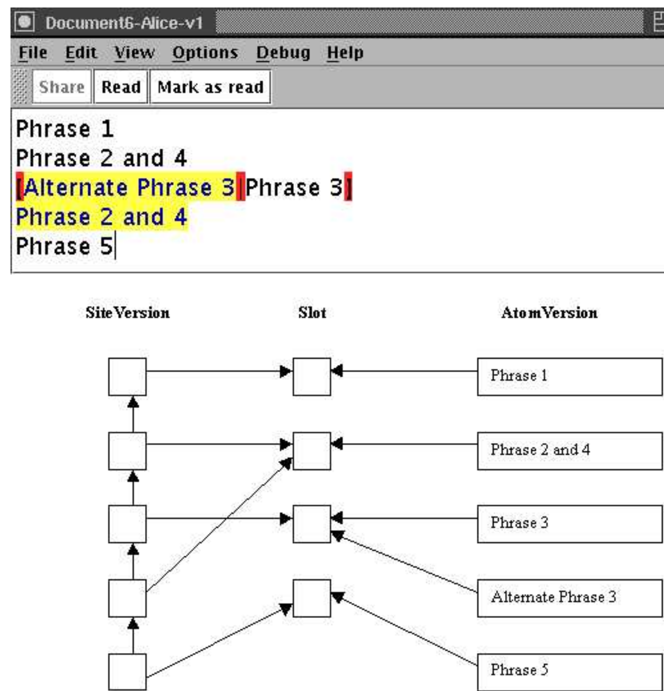

Figure 2.13: Database Organization for a Short Document

Support for Choices

The *CASTER* program allows each of its users to choose a preferred version in any conflict section. This choice implicitly rejects all other versions visible in the conflict section at that time. Such a choice is stored in the *CASTER* database as an *AtomChoice* record with a reference to the chosen version and a *choiceFlag* set to **Choose**. Such a choice could more conveniently have been stored instead as a set of *AtomChoice* records, one for each visible version that was not chosen, with a *choiceFlag* set to **Reject**. However, the commands to accept one version and to reject every other version are semantically different, and a user interface that supported both **Reject Version** and **Choose Version** commands might need to preserve this distinction. Although the *CASTER* prototype does not support a **Reject Version** command, its **Choose Version** function creates a single

AtomChoice record with a *choiceFlag* of **Choose** to preserve this distinction for possible future use.

The Problem of Newness

In the previous section we observed that each *AtomChoice* record in *CASTER* implicitly rejects one or more competing *AtomVersion* records. The nature of the *CASTER* model also dictates that any new *AtomVersion* can override other *AtomVersion* records that share its *Slot*: more precisely, each *AtomVersion* implies an *AtomChoice* for itself. Since *CASTER* thus heavily uses implicit rejection in its choice records, it must have a systematic way of figuring out exactly which *AtomVersion* records are rejected, and which are not rejected, by any *AtomChoice* that it encounters. From a user interface standpoint, this set of *AtomVersion* records can be stated clearly: the versions affected by any new choice or modification are the versions that were known within the author's view at the time that that author made the change. A new *AtomVersion* will cause all of these competing versions to be rejected, and a new *AtomChoice* record will cause all but one of them to be rejected. The remainder of this discussion will describe any *AtomVersion* or *AtomChoice* record as being "newer" than this set of competing known versions.

Figuring out this sort of newness is not straightforward in the database. Although each database record has an ID number corresponding to its write order in the database, storing newness is not as simple as storing write order. The problem with simple write ordering is that if two people change the same part of the document simultaneously (meaning that neither saw the other's changes while making their own changes) then the system must not consider either change to be newer than the other. If either of these changes were to be considered newer, then the author of the 'newer' change would never see the 'older' change. To correctly figure out such relative newness, another field must be stored in the

database. In the *CASTER* database, each *AtomVersion* or *AtomChoice* record includes a pointer to the most recently written *AtomVersion* (if any) that both shares its *Slot* and was present when it was written.

The following sequence of operations by multiple users provides an example of how these algorithms operate:

- Author A types a phrase
- All authors share and read new
- Author B modifies the phrase
- Author A modifies the phrase
- All authors share and read new
- Author C modifies the phrase
- All authors share and read new
- Author D modifies the phrase
- All authors share

This sequence of operations creates the database shown in Table 2.1.

Record ID	newest	text
1	NULL	original version of phrase, created by A
2	1	Author B's edited version of the phrase
3	1	Author A's edited version of the phrase
4	2	Author C's edited version of the phrase
5	4	Author D's edited version of the phrase

Table 2.1: Database Example Showing Newest Fields

The partial orderings that can be established in this case are shown in Figure 2.14. In this figure, a solid arrow means "is newer than" according to the above definition and a dotted arrow means simply "is written later than". The rule for determining relative

newness in this graph is as follows: Version A is newer than Version B if one solid arrow followed by one or more dotted arrows can be traversed from A to B. Note that the relative newness of Versions 2, 3, and 4 cannot be established, although Version 5 is provably newer than all other versions.

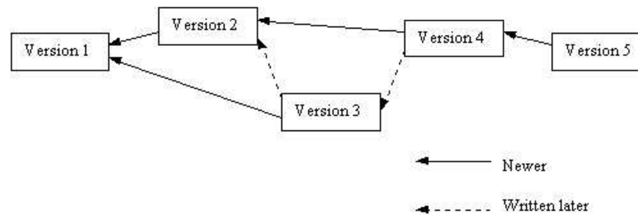

Figure 2.14: Determination of Relative Newness Relationships

2.3.2 Overall Configuration

The *CASTER* prototype collaborative writing program consists of two executable images: a client that provides a user interface for one author and a server that handles all reading and writing of data. Only one server should be running at any time, but any number of clients may be run simultaneously. Both the database server and client writing programs are written in the Java programming language, and were packaged as self-executing JAR (Java ARchive) files for ease of copying between machines.[5] They were developed primarily under the Linux operating system. The software runs on both the Linux and Windows operating systems, but user tests were performed with both server and clients running on Linux as well.

Communications between the server and the client were performed using a simple socket interface, with all messages being sent in plain text in both directions. The client-

[5]The program contains 17040 lines, including both source code and comments. It also contains a total of 7292 semicolons. The JAR file size is 380512 bytes.

server communication protocol is completely program-specific: no standard higher-level protocols were used.

2.4 Summary

This chapter has presented a model for collaboration in which each participant can achieve the concentration necessary for creative work by controlling his or her collaborative linkage to the other participants. This chapter has also described how model has then been used to build a computer program to facilitate collaborative writing. The next chapter will present the results of user tests run on the collaborative writing application and outline a number of possible areas for further research.

Chapter 3

User Tests and Results

Computer applications that involve either creative processes or collaboration are difficult to test, and systems involving both of these factors are still more difficult to test[20]. In this project a variety of test approaches were used, generating both quantitative and qualitative data.

Three sets of user tests were run on the *CASTER* Collaborative writing prototype program: a controlled user test, a shakeout test in preparation for the controlled test, and an earlier long-term iterative test that largely shaped the user interface. For each test, one section will describe the procedure and then one or more sections will describe the results from that test.

The quantitative information that came out of these tests consisted of counts of characters generated by each author in the controlled tests and statistics from questionnaires filled out by the participants in both the controlled and shakeout tests. The qualitative information included observations of users during both the shakeout and controlled tests, interviews with the participants at the end of each controlled test, and email feedback from the initial iterative trials. This qualitative information included a large number of

observations about how and when the user interface features succeeded or failed. Also included in this information were performance considerations for the program, personal reactions to the task and the program, and some insights into the nature of the writing process itself.

3.1 Procedure for Controlled Tests

The object of the controlled tests on the *CASTER* Collaborative writing prototype was to determine whether or not the *CASTER* model aided in collaborative writing. For comparison purposes, a special version of the *CASTER* collaborative writing program was developed where the user was not given control over the collaboration. This control version of the program automatically resolved all conflicts in favor of the latest content. In this control version of the *CASTER* prototype, everything that each user typed was immediately shared with all other participants.[1] In short, the control version of the program functioned like the shared cursor systems described in chapter 1 on page 26.

The tasks for the controlled tests, shown in Figure 3.1 and Figure 3.2, were deliberately chosen both to provoke conflicts and to force the participants to deal with those conflicts using the computer program. The reasons these tasks were chosen are as follows:

- The questions concerned the environment in which the participants live every day, maximizing the likelihood that they would have strong opinions.

[1] The difference in the immediacy of sharing – the *CASTER* program shares content on demand and the control program shares all content immediately – may appear to be a second parameter that was varied within this test. However, this difference was dictated entirely by the nature of the conflict resolution mechanisms. Immediate sharing in the *CASTER* system would be bad because it would introduce unnecessary, unpredictable interruptions, thus defeating the fundamental goals of the system. On the other hand, on-demand sharing in the control system would make its already serious problem (of un-trackable changes) significantly worse.

- The questions concerned controversial and nuanced issues upon which there was strong disagreement on campus, decreasing the likelihood that the participants' opinions would be exactly the same.

- The tasks could not easily be segmented into pieces that the participants could write non-collaboratively.

- The tasks intentionally did not suggest any format for the entire document. Thus part of the task was to determine the layout of the final document, a subtask which cannot be divided up.

- The time pressure imposed by the 30 minute time limit precluded the more relaxed collaboration method of passing drafts back and forth.

These tasks achieved their goals well: both participants were engaged, active, and interested during all of the tests. Figure 3.3 shows a sample document produced during these tests.

Problem A.

Imagine that you and your collaborator both work in the Dean's office at Wichita State University. You have been asked to propose a policy outlining the conditions under which University personnel can or cannot enter the dormitory rooms of students.

Please write this proposed policy jointly with your collaborator.

Figure 3.1: First Writing Task

The tests themselves were each done with two participants. Each pair received a short introduction to one system, worked for 30 minutes on a collaborative writing task, received a short introduction to the other system, and then worked for another 30 minutes

Problem B.

Imagine that you and your collaborator both work in the Dean's office at Wichita State University. You have been asked to propose a policy outlining the conditions under which alcoholic beverages will or will not be permitted on campus.

Please write this proposed policy jointly with your collaborator.

Figure 3.2: Second Writing Task

Wichita State University Alcoholic Breverage Policy:
Policies must be followed by all studenst and staffs in WSU
1) No Alcoholic beverages are allowed on campus .
2) Alcoholic beverages are allowed on campus on special occasion under restriction.
3) As Drunken and Driving is strictly probited by law ,the students may be checked for some drunken and drive test if suspected.
4) If any Student disobeys these policies they will be a warning for the first time and if its repeated again they will be suspended from the semester .
5)The alcohol content of all the drinks will be checked before it is sold in the Rhatigan student center to ensure the ALCOHOLIC BEVERAGE POLICY.
6) The alcoholic beverages can cause problems to who drink and also people around them, so strictly should be prohibited while driving.
7) Age limit should be there for students for alcohol consumption.
8) All the students should co-operate with the officials to make things working and life easier for people who drink as well as people around. Wichit

Figure 3.3: Sample Document Produced During User Tests

on another collaborative writing task. The participants filled out questionnaires before, during, and after the tests. The primary purpose of the questionnaires was to determine the participants' background and stimulate conversation about test experiences. However, the questionnaire also included a block of 13 questions that were asked twice, once for each system. The questionnaires and writing task statements are shown in Appendix B, along with histogram tables of the answers to the questions selected by the participants. The entire test process for one pair of subjects took approximately two hours. The tester observed the subjects while they performed the tests, and interviewed each pair together after the tests were completed. The materials given to the participants during these tests are presented in Appendix B.

The tests described above were done in blocks of four to control for two variables: learning effects and task difficulty. One test in each block used the first of two writing tasks and used the *CASTER* system before the control system. The second test in the block used the second writing task and also used *CASTER* before the control system. The remaining two tests in each block were similar to the first two except that they used the control system before the *CASTER* system. Two complete blocks of controlled tests were run: a total of eight two-hour tests involving 16 test subjects.

All of the tests were performed at Wichita State University. The participants were mostly students in the departments of Computer Science and Psychology. All of the test participants had written at least one five page paper in their lifetimes, but only a minority had written ten or more such papers. All were passable typists and comfortable using computers. The test procedures were approved by the university's Internal Review Board, and every test participant signed a consent form.

3.2 Interview Results from Controlled Tests

All of the participants reported that they enjoyed the tests. Both the *CASTER* system and the control system were sufficiently usable for the participants to complete the assigned tasks, and the test subjects liked the novelty both of the tasks and of the computer interfaces.

Of the eight pairs of participants in the controlled tests, six pairs preferred the *CASTER* system, one pair was split – one participant preferred the *CASTER* system and one liked them equally – and the remaining pair both preferred the control system.

Of those test subjects who preferred the *CASTER* system, the most common first impression that they expressed about the *CASTER* system was that it was less disruptive, less distracting, and less pressuring – in other words, they first noticed and appreciated the delayed reading feature. When questioned further, however, they expressed some disagreement about whether delayed sharing or delayed reading was more important. Although there was no consensus about which of these two features is more important, there were advocates for each feature, and their opinions were often quite strong. These results suggest that building a system with either one of these features, even if the other could not be included, would still be worthwhile.

The pair of test subjects who preferred the control system to the *CASTER* collaborative writing system was asked some additional questions. Although they had never worked together before, they completed their tasks quickly and easily, and agreed that their work flowed well in both systems. This pair of participants found that they thought similarly, agreed most of the time, and coordinated well: one participant stated, “I knew what she was going to write about.” They preferred the control system’s immediate exchange of ideas because its immediacy allowed them to work faster. When questioned

further, they stated that they thought that they might have preferred the *CASTER* system if there had been more participants, and that the *CASTER* interface might make a good training interface for others. This idea of a training interface was discussed with the participants of all remaining tests, but it did not resonate with any of them. They all said that they would prefer to use the *CASTER* interface always, and that they would never reach an experience level where they wanted to "graduate" to the control system's immediate-exchange interface.

3.3 Observations During Controlled Tests

The test administrator observed the test participants during each writing exercise, listening to their comments and questions and helping them with interface problems. The following sections will describe problems, usage patterns, and behaviors that were observed. Most of the interface problems were due to interface features that performed in ways that were non-obvious or unexpected by the participants.

3.3.1 Patterns of Sharing

When they first started writing, participants testing the *CASTER* interface often forgot to either share their contributions or read the contributions of their partner. After having to remind every participant to share and read during the first few tests, the test administrator modified the initial instructions for the *CASTER* interface to specifically stress that they should do this. Unfortunately, reminder mechanisms within the program such as blinking the Share and Read buttons would have been sufficiently distracting to violate the design requirements, so the best available solution to this problem is probably to find a way to

stress these features in the instructions. As the ideas from this interface are integrated into consumer products, a good tutorial of these features will be vital.

After the first few tests, the test administrator recommended to every participant that he or she always share and read at the same time. The participants consistently did this, and rarely invoked either of these functions without invoking the other. This seems to be a useful usage pattern, and might make a good default for the system: perhaps **Share and Read New** should be one command, and the separate Share and Read commands could be made somewhat less accessible.

Once they got into the habit sharing, however, the participants quickly figured out that there were disadvantages to sharing too infrequently: if one worked for too long without sharing, the chance of doing the same thing twice increased, and sorting out too many interwoven changes could get confusing. Figures 3.4 and 3.5 show a hypothetical example of this situation: in Figure 3.4 the author has typed in a paragraph before checking with his coauthor, and when merging has discovered that the coauthor typed essentially the same thing. All of the information is available to merge the two author's versions, but the process will be tedious: whoever does the merge must merge his or her ideas into the text created by the other author and then delete his or her own version. Merging is much simpler if an author frequently reads the other author's contributions, as shown in Figure 3.5. In this case this author can interleave his or her new ideas with those of the other author, with much less merging required. The participants therefore developed a habit of sharing once for every sentence typed, and sometimes once for every phrase. This behavior was necessary because of the exceptional intensity of collaboration required by the design of the writing tasks, discussed earlier.

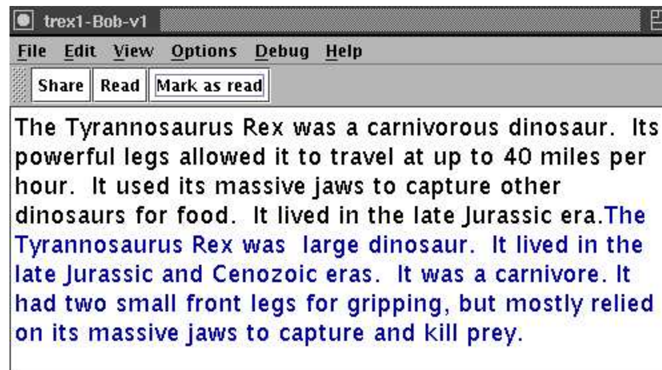

Figure 3.4: Consequences of Delayed Reading

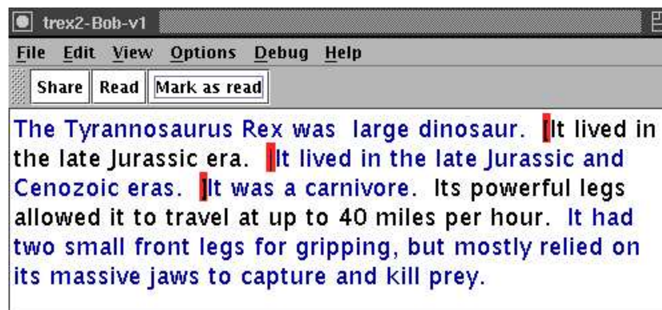

Figure 3.5: Result of Frequent Reading

3.3.2 Learning to Use the Chooser Dialog Box

The use of a dialog box for conflict resolution was not obvious: most participants did not think to double-click on conflict sections even when this feature was described beforehand. Once the participants had seen it demonstrated once, however, they found it very natural to use.

3.3.3 Learning Curve Effects

The two problems described in the preceding paragraphs were both specific to the *CASTER* interface: they presented no problems in the control system, because it did not support those features. More generally, the *CASTER* interface was more complex to learn than the control system because the instructions for the control system were almost a strict subset of those for the *CASTER* system. This increased complexity is shown by the relative lengths of the instruction sheets for each interface given to the participants, shown in Appendix B. The tasks in these tests were sufficiently short in duration that the participants spent a significant portion of their work time learning the interfaces, and this extra complexity meant that in a comparative test the *CASTER* system was operating at a slight disadvantage.

3.3.4 Getting Started

Test participants were isolated in separate rooms and required to communicate only through the collaborative writing program. As a result, most groups started out by using the text entry window as a chat window. Figure 3.6 shows an example of this usage. The participants quickly figured out, however, that the best way to discuss the task was

Do u get my msgs?
hellooooo

Yes i do get ur messages
good
what do u propose
personnel can or cannot enter?

I propose they cannot enter without the permission of the student
yeah. that's write

nobody can enter their personal lives unless it is essential

so what would be the policy say?

WOW u r doing great

If they enter they have to have a written document of the student
.......giving the permission to enter

Figure 3.6: Chat Behavior at the Start of a Writing Task

to simply write something and see how their coauthor reacted. The time pressure of the 30-minute test also pushed them away from chatting and into writing.

3.3.5 Narrative Style Versus List Style

The writing tasks both required the participants to write a policy. Such a policy can be written either in narrative form or as a bulleted or numbered list, and some participant groups chose each approach. Participant groups who used list formats spent a lot of time rearranging the list items. This behavior suggests that for more list-oriented tasks a move function may be worth the trouble of the additional interface complexity. It also suggests that a collaborative outline generator would be helpful to generate the skeleton of the document prior to collaborative writing.[2]

3.4 Measurements of Inclusivity by Character Counting

The controlled tests generated a total of 16 collaboratively written documents: one from the control system and one from the *CASTER* system for each pair of test subjects. A quantitative measure of the inclusivity, or balance of participation, for each system was generated by calculating the percentage of the characters in each document that were written by the minority author. For this metric, the best possible inclusivity score is 50%, and the worst possible inclusivity score is 0%.

The analysis started with the computation of the number of characters in the final document that were generated by each author. To accumulate these counts, each clause that was originally written by one author was credited wholly to that author. Each clause that was generated by modifying another clause was assumed to have been a modification of that older clause residing in the same slot that was most similar to the new clause.

[2]This collaborative outliner may be an easier task using existing technology: evidence exists that some older attempts at collaborative writing systems work best in list-oriented tasks[50]

The changed characters in the new clause were then credited to the new author, and the ownership credits for the remaining characters were inherited from the older clause. In the *CASTER* system the contents of conflict sections were amortized: the ownership credits for the contents of each conflict section still pending at the end of the test session were divided by the number of visible versions in that conflict section.

The results of these calculations are shown in Table 3.1. The inclusivity metrics for the documents generated with the *CASTER* program were on average 4% higher than those for the documents generated by the control program. The testing factor for the hypothesis that the *CASTER* system significantly improves inclusivity, based on these tests, is Z=0.7149 [29, page 313], where a Z of 2.0 would correspond with a 95% confidence factor. Based on these results, the probability that the *CASTER* program significantly increases inclusivity is 76%.

Based on these data, the hypothesis-testing factor for the hypothesis that the second system tested by each pair was easier to use (i.e., that the learning effect between tests significantly affects performance) was 0.5653, corresponding to a probability of 72%. The hypothesis-testing factor for the hypothesis that the first writing task was easier than the second writing task was 0.5127, corresponding to a probability of 69%.

3.5 Questionnaire Comparison for Controlled Tests

The questionnaires that were given to the participants during the controlled tests contain one block of 13 questions that were asked identically about both the *CASTER* system and the control system. A simple numerical analysis was done on these test results to determine whether the *CASTER* system or the control system performed better according to the criteria asked by those questions. The results of this analysis are shown in Table 3.2.

Test number	Control System	CASTER system	Improvement
1	33%	39%	+6%
2	37%	20%	-17%
3	46%	50%	+4%
4	46%	38%	-8%
5	31%	41%	+10%
6	30%	48%	+18%
7	15%	46%	+31%
8	49%	41%	-8%
Average	36%	41%	+4%
Standard Deviation	11%	9%	
Z0			0.7149

Table 3.1: Percentage of Document Contributed by Minority Author

The questionnaires from which these questions were extracted are shown in Appendix B. The procedure for performing numerical comparisons was as follows:

1. Consideration was limited to the 13 questions that were asked identically for each system.

2. Consideration was further limited to the ten questions where there were clearly identifiable "best" and "worst" responses. Sometimes the "best" response was the leftmost one, and sometimes it was the rightmost one.[3] The table identifies which response was considered "best" for each question.

3. A numerical score was generated for each question response. The worst possible response for each question was assigned a value of 0, and the other responses were assigned values that counted up their distances from the worst response (1, 2, 3, etc.). Thus for each question where the leftmost response was the worst, the score for each answer was the position of the response counting from the left. For each

[3]In a well-designed questionnaire, approximately half of the questions should be reversed in this way. If the "best" answer is always to the left or always the right, then the respondents get in the habit of choosing answers on the left or right side, thus skewing the test results.

question where the rightmost response was worst, this scoring was reversed: the score for each answer was the position of the response counting from the right.

4. Table 3.2. shows three values for each question computed from these scores. The first value is the average score for each question, averaged over all respondents and both systems. The second value is the average difference in score for each question, computed by subracting the control system's score from the CASTER system's score, averaged over all respondents. The third column is the total difference in score for each question, which is the previous column multiplied by the number of respondents.

Quest #	**Gist of Question**	**"best" response**	**Avg Score**	**Avg Diff**	**Total Diff**
1	we agreed/disagreed	left	2.4/3	-0.19	-3
2	confident I was heard	left	2.4/3	-0.13	-2
3	did (not) feel time pressure	right	2.1/3	-0.31	-5
4	felt (un)worried as I wrote	left	2.5/3	+0.38	+6
5	did (not) edit myself	right	2.2/3	+0.25	+4
6	easy/hard to understand other's changes	left	1.9/3	+0.06	+1
7	confident other understood your changes	left	2.8/4	-0.56	-9
8	who had bigger role	-	-	-	-
9	did/didn't "agree to disagree"	right	2.6/3	+0.06	+1
10	other wrote stuff that surprised you	-	-	-	-
11	other wrote stuff you hadn't thought of	-	-	-	-
12	was this document complete?	right	1.6/3	-0.44	-7
13	was this experience pleasant?	right	2.8/4	+0.06	+1
	Total for All Questions				-13
	Average for All Questions		73%	-0.08	-1

Table 3.2: Comparisons of System Quality Based on Questionnaire Responses

These results do not significantly indicate that either system is better or worse. The averages over both systems show that overall the respondents liked both systems. The average differences show that the respondents liked both systems about equally according to these measures. The total differences show how few response shifts (moving an answer

one position to the left or right) were responsible for these variations. Over all the questions, the *CASTER* system scored worse on average by a factor of 0.08 point, which corresponds to one question in twelve being scored one choice lower. Because these questions focus primarily on factors that would be influenced by the expertise of the system user, these data primarily indicate that the advantages of the *CASTER* system are approximately balanced by the increased difficulty of learning the *CASTER* system in the factors described above.

3.6 Procedure for Shakeout Test

The shakeout test was the first "live" user test performed on the *CASTER* collaborative writing prototype. This test was run using 33 subjects simultaneously for approximately two hours. The intended purpose of this test was to ensure that the program was ready for the controlled tests, but many interesting observations came out of this test as well. The test was free-form: the tester explained the program to the group, allowed the users to work or play with it however they wished, and watched and talked to them during and after the test. Data were collected using a preliminary version of the questionnaire used for the controlled tests, and the documents generated were also examined.

3.7 Results from the Shakeout Test

The informal nature of the shakeout test led to a number of discoveries related to how the program worked with inexperienced users. Much of what worked with an expert user whose usage had evolved with the system did not work so well with beginners. Two significant areas for improvement were found: the need for direct modification of text

within conflict sections and the need for better terminology. Finally, there were some surprises in the usage patterns chosen by the participants.

At the time of this test, the only operation that a user could perform on a conflict section was to resolve the conflict. To modify one of a set of conflicting versions, the user was required to select a preferred version and then modify it after the conflict section disappeared. This "resolve first" approach led to a well-ordered task flow that was acceptable to the experienced programmers who had hitherto tested the system. However, essentially all of the beginner users in this test wanted to be able to position the cursor anywhere within any version, whether conflicting or not, and start typing. This feature was added for the later controlled tests, and worked well.

Another advantage of modifiable conflict sections is that it enables use of a common view instead of private views. It was observed that private views caused document coherence to deteriorate rapidly, and thus a shared common view mode was essentially required for the controlled tests. Developers of other collaborative systems[23] have argued that both common and private views are necessary. In a common shared view, the only way to resolve a conflict is to repudiate your own version. Thus with the "resolve first" approach, an intractable conflict was not modifiable by anybody. Allowing direct modification of conflict sections solved this problem.

The shakeout test exposed some terminology problems. First and foremost, all icons in the toolbar had to be replaced by text, because icons for the novel operations in this application were invariably more misleading than helpful. Beyond that, the database term **Commit** was removed from the interface and replaced with the term **Share**, which is also used throughout this dissertation. Similarly, the database-oriented term **Update** was replaced with **Read New**, and the term **Accept** was replaced with **Mark as Read**. The questionnaire was also fine-tuned based on feedback from the shakeout test.

The usage patterns of the system also held some surprises. Most significantly, the investigator considerably underestimated the degree to which users, even beginners, would find collaborative writing to be fun. Some of the documents created by these beginners were exploratory, containing phrases like "What does this do?" However, at least one ad hoc group wrote a collaborative story, and one user in one of the two test labs experimented with mildly vandalizing a document created by users in the other lab.

3.8 Procedure for Iterative Trials

Learning to use the *CASTER* collaborative writing program was anticipated (correctly) to be a doubly difficult process: users had to learn not only an entirely new interface but also a totally new approach to writing. Many iterations of the user interface were necessary to lower this barrier to learning sufficiently so that the system would be even accessible to non-expert users. To develop the interface to this point, the author and one other computer-proficient user developed and tuned the interface over a period of five months using iterative trials. The author's partner in this effort was an experienced professional researcher.[4] This testing partner occasionally tested the system with one or more associates willing to experiment with the system with him. These iterative trials were conducted by sending a copy of the program to the tester, having him install and run it, and receiving comments back by electronic mail or telephone. The results of these iterations were a large list of suggestions for changes to the user interface, most of which were adopted. These iterative trials were the biggest formative factor in the user interface for the *CASTER* collaborative writing prototype program: aside from the features directly

[4]This test partner for the iterative tests was Dr. Gregory P. Kochanski, then at AT&T Research and currently at Oxford University.

dictated by the design requirements, most of the design decisions for this user interface were based on empirical results from these iterative trials.

3.9 Results From Iterative Trials

The user interface for the *CASTER* prototype started with the core functions required to satisfy the design requirements, as described earlier. Beyond this core, most of the other features of the system evolved extensively by trial and error through iterative testing. The discoveries made in these tests were many and varied, but the most significant user interface findings can be grouped into three areas: features analogous to those in a traditional word processor, usage of color, and issues concerned with text segmentation. This section presents these findings and concludes with a few observations about the development and runtime platforms.

3.9.1 Word Processor Features

Early in the iterative testing an understanding was developed that seems obvious in retrospect: that because non-collaborative writing is done with word processors, user will find it easier to do collaborative writing if the system looks and feels as much as possible like a single-user word processor. Duplication of all of the features of a word processor was not feasible, not only because of the prohibitive amount of development effort but also because of the complications that such features would introduce into the relatively short duration controlled tests. However, user testing identified a minimal necessary set of these features, and also showed ways of mapping some of the collaborative features of the *CASTER* prototype to features commonly found in word processors.

One area that benefited from the emulation of word processors was the startup process. Creation of a *CASTER* session requires four pieces of information: a document name, a clause type or segmentation rule if the document is new (discussed in greater detail below), an author name, and a view name. This information quadruple together is roughly analogous to the filename in a single-user word processor. The first design iterations allowed selection of these fields directly in the frame displaying the text, but it was quickly discovered that a separate dialog box analogous to a file selection dialog was much easier to use, as shown in Figure 2.1 on page 33. Furthermore, all of these fields other than the document name confused most users. Although the author field selection is absolutely necessary and unavoidable, the clause type and view fields can be, and were, set to default values so that the user did not have to deal with them. The clause type, which appears in the dialog box immediately below the document name, is set in a new document to a default value that works fairly well for the most common types of document. The view field for a new author is set by the program to a default name, and the view field for an existing author with a single defined view (the most common case for beginning users) is set to the name of that view. These fields not only are defaulted, but they are also skipped in normal workflow. When the user selects a document name and presses the **Enter** key, cursor focus is moved past the defaulted clause type to the author selection field. Similarly, when the user selects an author name and presses the **Enter** key, cursor focus is moved past the view field to the OK button. Thus users can – and do – use the interface without understanding or caring what the clause type and view fields do.

The dialog box described above is removed before the frame displaying the text is presented, as a file selection box in a word processor would be. Thus a separate function is needed to open a different view. This function was implemented as a **New Window...**

button in the **File** menu, similar to that found in most web browsers. This button presents a new version of the dialog box described above.

For creating and modifying large documents, a **Find Next** dialog box was determined to be necessary. This box, however, was never used by any participant in either the shakeout tests or the controlled tests.

The *CASTER* system handles plain text, and thus serves as a good front end for text-based document production systems such as *TEX*[31]. To achieve this type of integration, import and export functions were found to be necessary. A checkpoint file, written to a filename based on the document name every time text is shared, was found to be extremely useful for document generation controlled from a makefile. Users never wanted to generate final documents from text source with unresolved conflicts, so no effort was deemed necessary on rendering conflict sections cleanly – any conflict sections were just included verbatim in the exported document, delimiters and all. A set of conflict delimiters that turned each conflict section into an HTML table was tried, but in user tests proved to be not worth the trouble to use.

3.9.2 Use of Color

The *CASTER* collaborative writing program makes extensive use of color in its user interface. This section describes what was learned through iterative testing about color use. The first subsection will describe how the colors were chosen, and the second subsection will describe which properties were (or were not) usefully represented by these colors.

Color Choice

The first question to ask when choosing colors is whether or not color should be used at all. Ben Shneiderman's first rule of thumb in color use is to "Use color conservatively"[44]. Other available appearance cues for text include blinking, underlining, size, typeface, bolding, and italics. Unfortunately, none of these other cues proved suitable for *CASTER*. Blinking is too intrusive and distracting for writing. Underlined text segments proved during testing to be surprisingly difficult to identify quickly and accurately. The remaining appearance cues listed above are all traditionally made available to the user for the presentation of text in documents, and thus should not be overlaid with program-specific meanings in a writing application.

During testing two methods of marking text with color were tried: coloring the text itself, generally referred to as "foreground color", and coloring the background around the text. Both were eventually used in the interface, but for different purposes.

A foreground color provides a subtle cue: the user notices it if he or she is specifically looking for it, but may overlook it when considering the document as a whole. Although lighter colors are more noticeable, they were discovered to be significantly less readable because of their reduced contrast. Therefore, only fairly dark foreground colors were used.

Background colors provide a much more noticeable cue. Text with even a very pastel background color was discovered to stand out extremely well against the surrounding text with white background. Contrast for readability is not an issue here as it was with foreground colors – rather, background colors must be kept very pastel to avoid an overwhelming amount of visual noise.

Testing also showed that there was significant value to be added by using the most common colors of highlighter pens to exploit the pen and paper metaphor. Specifi-

cally, the interface uses a bright yellow background, corresponding to the most common highlighter pen color, and a light green background, corresponding to another common highlighter pen color. Pink, the third common highlighter color, was not used so that the color red could be used for other visual cues. The gray levels of the yellow and green backgrounds are sufficiently different from each other that they should be distinguishable by a color-blind user; however, if color-blind users were common then making one of these backgrounds have a fine pattern would make them more distinguishable for these users.

Applications of Color

The *CASTER* interface went through many design iterations before it was determined how to use color most effectively to advance the process of collaborative writing.

In early iterations several attempts were made to present the data model to users, based on a mistaken assumption that users would need to understand the data model to effectively use the system.

The earliest trials gave text a bright pink background if it was in an active conflict and a pastel pink background if it was in a resolved conflict. This approach was a complete, unquestionable failure: coloring of already-marked conflict sections is redundant, and nobody cared about conflicts once they were resolved.

The next trial used a unique background color to distinguish each of the visibility classes used to determine what is shown or hidden in a view. These visibility classes are:

- repudiated by this view (hidden by default)
- repudiated by another view (hidden by default)
- rejected by this view (hidden by default)

- explicitly chosen from a conflict section by this view (visible by default)
- accepted (marked as read) by this view (visible by default)
- new and not yet acknowledged by this view (visible by default)

This color coding was coupled with a feature that allowed the user to individually show or hide each of these classes of text. This text class interface was created for a number of reasons, all of which were shown during testing to be largely wrong:

- It was thought that users would be confused by the disappearance of versions in conflict sections, and would need a tutorial on text classes to understand what was happening. Testing showed that the behavior of conflict sections made intuitive sense to the users, even those who knew nothing about how or why it was done.
- A user who deleted the wrong thing by mistake would need to turn on the visibility of the “Rejected” or “Repudiated by this view” text classes to get the deleted thing back. Testing showed that the undo feature performed the same function and was much more natural to use. Furthermore, users of word processors are conditioned to retype things that they accidentally delete, so users experienced with single-user word processors would not even bother to look for such a recovery feature.
- A user who changed his or her mind about something he or she had deleted and then committed could go back and find it. Testing showed that in essentially all cases the user would prefer to just retype the deleted text.
- If one user repudiated text that another user had accepted (marked as read), then that second user might wish to retrieve the original text. This potential problem was never encountered by any actual users during testing, but might arise more often in

documents with many authors. These features were left in the interface as options for advanced users, but were so rarely used that they cannot be considered to have been tested.

Another color scheme that was tried was to alternate the colors of the clauses, as a method for teaching the user about the clause divisions used for conflict sections in the document. This feature was useful to the developer for debugging, but testing showed that users did not care about the clause structure of the document.

The color scheme for text that was finally settled upon was to represent the text created by the current view in black and to use a unique color for text created by each other view, as shown in Figure 3.7. The colors used were relatively dark foreground colors to make this cue subtle. User feedback was that this coloring was nice but not necessary. Although information about the view that created a clause should be available on demand, it need not be immediately shown in the user interface. For future tests with large numbers of users, a simpler scheme might be preferable in which text created by the active view is shown in black and text created by *any* other view is shown in one other color.

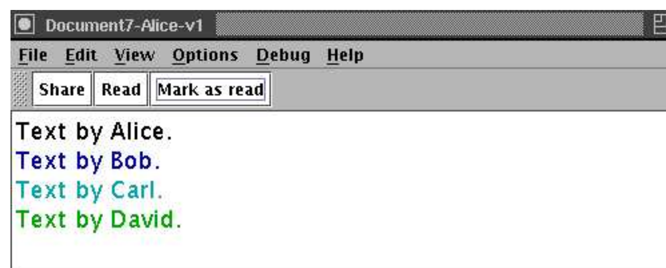

Figure 3.7: Coloring by Author

3.9.3 Text Segmentation

Whenever multiple views create two or more different versions of the same text fragment, one or more conflict sections will be presented to the users showing the different versions. A predefined segmentation of the text – done by an algorithmic rule – is used to determine how much text will be displayed in each conflict section. If the conflicting text region is smaller than one text segment, then each version of that conflicting text will be presented within the context of the (identical) remainder of its segment. Similarly, a disagreement about text that extends over multiple text segments will be displayed as multiple conflict sections. The choice of this text segmentation rule is crucial to whether conflict displays seem natural and informative or unnatural and confusing. This section describes the many design iterations undergone by the segmentation algorithms of the *CASTER* collaborative writing system.

The first subsection will describe the specific rules that were tried for segmenting text, and the experiences of users with each. The next subsection will describe the need for multiple segmentation rules. The final section will describe test results concerning the related problem of how to best display conflict sections for effective understanding and manipulation.

Segmentation Algorithms

The first design iteration defined a document segment as a single line, consisting of zero or more non-carriage return characters followed by a single carriage return:

line-segment := <non-CR> ∗ <CR>

This text segmentation is identical to that used in change sections of *RCS* files. This segmentation performed poorly for text entry: modern text widgets only use explicit carriage return characters to indicate paragraph breaks, and thus each segment is an entire paragraph.

The idea of segmenting by the line breaks (i.e., implicit carriage returns) in the text widget display was not even tried because of its instability: each change near the beginning of a paragraph would change the segmentation for the entire paragraph. In this case, even resizing the window would radically change the segmentation.

The next text segmentation rule tried was to segment by phrase. Specifically, a phrase text segment consists of text followed by punctuation:

phrase-segment := text-char∗ punct
text-char := <regular-character> | <space-character>
punct := <punctuation-character> end-punct-char∗
end-punct-char := <punctuation-character> | <space-character> | <CR>

Note that there may be a fractional segment at the end of the document. This rule worked well, dividing flowing text into manageable segments for conflict sections.

One refinement that was discovered in testing is that the hyphen character (‘-’) must be treated as a non-punctuation character. This was true for two reasons. First, importing text with hyphenated word breaks made a terrible mess when one removed all the hyphens. Second, it was seen by users as unnatural to have naturally hyphenated words split between segments.

In the shakeout testing, however, the test users created many short paragraphs separated by carriage returns, and in these short paragraphs it was discovered that segments

containing both text and carriage returns were extremely confusing for beginners to understand, both in the in-line conflict display and in the conflict resolution dialog box. A new segmentation rule was therefore designed in which each sequence of carriage returns is treated as its own segment and the remaining text is divided into phrases as before:

phrase-or-cr-segment	:=	phrase | <CR><CR> *
phrase	:=	text-char* punct
text-char	:=	<regular-character> | <space-character>
punct	:=	<punctuation-character> end-punct-char*
end-punct-char	:=	<punctuation-character> | <space-character>

This final segmentation rule was used in the controlled tests and worked well enough that it was not noticed by the participants.

Implicit in the preceding segmentation discussions is the assumption that any conflict spanning multiple segments will be presented as a sequence of single-segment conflict sections rather than one multi-segment section. This is indeed the case in the *CASTER* prototype. This approach was taken for two reasons. First, it was thought that a large conflict section, potentially many times the size of the screen window, would be difficult for a user to read, understand, and manipulate. Second, a small change made within a large conflict section would require the conflict section to be split, which would cause a radical and confusing change to the display. Third, as a practical matter it was thought that keeping the conflict sections small would reduce the chance of the text within them losing synchronization. The author has frequently found using *RCS* for source code that when a major change is made, such as moving an entire procedure to another place in the file, that the version comparison functions become unsynchronized, recognizing hardly any of the

matching text in the old and new file versions. Since this sort of de-synchronization would permanently destroy effective collaboration in the document, the size of text segments was limited to minimize the risk of this happening.

Keeping conflict sections short led to another problem: a long change might lead to a long series of conflict sections. In practice this did not happen with text changes: the way the interface works, rewriting a paragraph almost always results in a deleted paragraph followed by a newly written replacement. In practice the only long sequences of conflict sections result from large deletions. These long deletion sequences can only be conveniently resolved (accepted or rejected) by providing a single action that can be repeated for each segment. This repeated action is provided by the **Apply and Advance** button in the conflict resolution dialog box. This **Apply and Advance** button is currently the only reason that the conflict resolution dialog box is necessary: otherwise a direct-manipulation conflict resolution interface would be possible, where the user simply double-clicked on the desired version in a conflict section to select it. Unfortunately, the equivalent of the **Apply and Advance** function in such a direct manipulation interface would require warping both the text pointer and the cursor pointer after each double-click, which is extremely disorienting for users. If a specialized interface were used for conflict sections containing deleted text, such as the strikeout font used in Microsoft Word, then the need for a conflict resolution dialog box would be removed and the direct manipulation interface could be used. This change would make a suitable subject for a follow-up research project.

The Need for Multiple Segmentation Rules

Throughout these design iterations an understanding evolved about how flexible the process of text segmentation needs to be. The initial goal of testing was to find one optimal

text segmentation rule, but early testing showed that different document styles, such as flowing text or TeX source, required different segmentation rules. The interface was modified to allow a choice of segmentation styles at document creation time. More recent empirical observations have indicated that advanced users might like to change the text segmentation for generating conflict sections interactively, and furthermore that text segmentation conceptually should be a display attribute rather than a database attribute. Unfortunately the prototype system was entirely designed around a database composed of segmented text. Future database designs for collaborative writing systems will be vastly improved if they store documents independently of the segmentation rule used to generate conflict sections. One such database design is outlined at the end of Appendix A on page 115.

Conflict Display

Each conflict section in a document is displayed within the body of the text by showing all currently eligible versions of the segment separated by delimiters. A number of delimiter configurations were tried, but the first and simplest one worked best. With this option, shown in Figure 3.8, the entire conflict section is surrounded by brackets and the conflicting versions are separated by the vertical bar character. Another strategy set off each conflict section by indenting it and stacking up its versions, as shown in Figure 3.9. This configuration proved not to be very useful, because the conflict sections were not significantly easier to interpret and the resulting disruption of the narrative flow of the text was significant. A set of HTML delimiters to build a table around each conflict section for export and external display was also tried, as shown in Figure 3.10. Testing showed that users did not want to see conflict sections outside of the *CASTER* program, and thus would resolve conflicts before exporting.

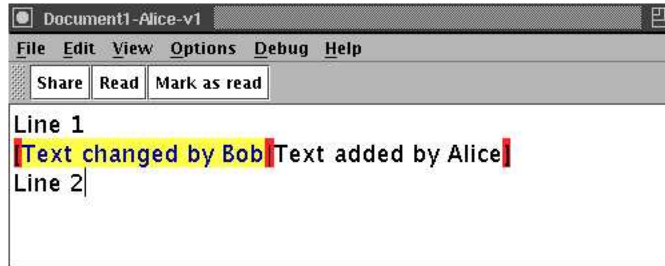

Figure 3.8: Conflict Displayed Side by Side

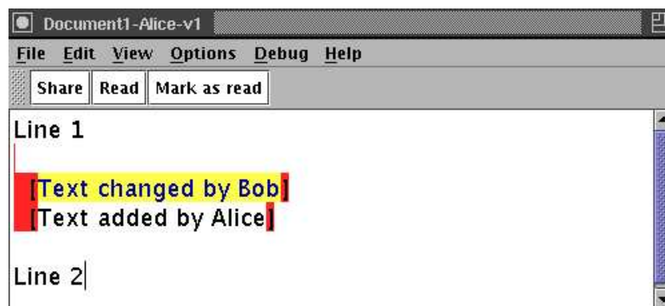

Figure 3.9: Conflict Displayed Stacked Up

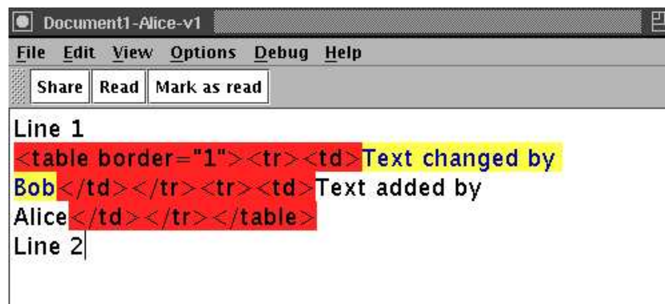

Figure 3.10: Conflict Displayed as HTML Table

Displaying conflict sections for deleted text, where one version is the empty string, proved to be another challenge for the user interface. The design question that was empirically tested was whether to display the empty version as some tag text such as "<deleted>" or to not display it at all. It was found that displaying nothing for the deleted version worked better in the text window, because users quickly got used to it and found the tag text to be distracting. On the other hand, the tag text was necessary in the conflict resolution dialog box: the blank version was not only invisible in the window, it was also not selectable because the Java list selection interface would trim it from the display. Figure 2.12 in chapter 2 on page 42 shows this final design.

3.9.4 Design and Implementation Issues

The previous sections describe user interface issues encountered during iterative testing. This final section describes the problems encountered during iterative testing that were not directly related to the user interface. These areas of concern are the database design and performance optimization.

Database Issues

The original database design was a relational database, accessed using SQL and stored using the MySQL free software database server on Linux. Unfortunately, MySQL is quite difficult to install, configure, and manage. This difficulty not only essentially precluded a downloadable kit for testing by non-programmers, but also made the kit too difficult for software professionals and students to install as well. A relational database was also massive overkill for this application, for the following reasons:

- The *CASTER* database is purely accumulative. No SQL table row, once written, is ever modified or deleted.

- The *CASTER* database is never inconsistent, even temporarily: each *CASTER* transaction can be written entirely with single-row insertion SQL transactions, and any client reading any of these intermediate states will not only get a self-consistent database but also will automatically fill in any gaps during the next incremental read.

- The *CASTER* client uses most of the database to generate the display. Testing experience shows that the majority of text is visible and uncontested, and thus most of the actual text in the database will be displayed in most clients. There is no indication that the number of database records used used for client display would ever be expected to be less than O(N), where N is the number of records in the database. Thus data filtering via a SELECT statement is unnecessary.

- The *CASTER* program determines visibility within each conflict section by scanning the conflict and display directives for that section in the order they were written in the database. Thus database write order is the ideal order for reading data into a client, and thus SQL's indexing features are unnecessary.

To simplify the configuration and leverage the characteristics of the *CASTER* database, a simpler database server was implemented in Java that stored the same information from the original MySQL table rows in a single flat file for each document. The actual contents of the database are detailed in Appendix A.

Performance Optimization

The *CASTER* program was typically tested in the iterative trials with documents about the size of a ten-page technical paper. During these trials only one performance problem caused by program algorithms was found, and this problem was fixed by a change of data structure.

The function that was unacceptably slow determines the *SiteVersion* record (the segment or conflict section) that contains a given text cursor position (number of characters from the start of the document). This function is used during text insertion, text deletion, cut, paste, and choice handling. It was originally implemented as a linear search, iterating through the display list and accumulating *SiteVersion* lengths until it reached the desired position. This algorithm introduced significant delays, especially during operations near the end of the document. The performance problem resulting from this approach was fixed by re-implementing the display list as a balanced binary tree. Each node of this tree, whether an internal node or a leaf, contains a *SiteVersion*. Each tree node also contains a text offset relative to its immediate parent. Thus for a reasonable large document the root node contains the *SiteVersion* halfway through the document and an offset of approximately half the document length. The left child of the root node contains the *SiteVersion* one-quarter of the way through the document, and its offset is approximately negative one-quarter of the document length. This document data structure fixed the performance problems by allowing all formerly O(N) interactive operations to happen in O(lg N) time. Table 3.3 shows the number of data manipulations required to use and maintain this data structure for all significant operations. In this table, N is the total number of *SiteVersion* records in the document and K is the number of *SiteVersion* records affected by the operation, which is usually small relative to N.

Operation	**linear list**	**red-black tree**
Insert character at start	O(1)	O(lg N)
Insert character at end	O(N)	O(lg N)
Insert character, average	O(N)	O(lg N)
Delete character	O(N)	O(lg N)
Resolve conflict	O(N)	O(lg N)
Initial read	O(N)	O(N lg N)
Read new	O(K)	O(K lg N)
Cut	O(N+K)	O(K lg N)
Paste	O(N+K)	O(K lg N)
Share (not affected)	O(K)	O(K)

Table 3.3: Performance Comparison of Display List Datatypes

No other algorithms in the program required O(N) computation for any single text modification operation. Thus a large document may take a significant time to read at application startup, but will not significantly slow down any normal operations.

Chapter 4

Future Directions and Conclusions

This chapter enumerates and describes a number of areas in which this research can be continued and extended, both within and beyond the application of collaborative writing. The final section of this chapter then selects and summarizes both the most significant results and the most promising directions for continuation of this work.

In this project a novel collaborative model was invented and a collaborative writing system was developed using this model. Future directions in which this project can be taken can naturally be divided into four categories: further testing, improvements to the collaborative writing program, applications of the collaborative model to areas beyond collaborative writing, and extensions to the collaborative model itself.

4.1 Further Testing

The controlled tests performed in this study were highly focused on the conflict resolution mechanisms of the program, and were designed to compare these mechanisms to those of shared-cursor systems, the only other approach that has been tried for synchronous collaborative writing.

There are a number of dimensions in which testing could be extended:

- Increase the number of simultaneous document authors to three or more.
- Lengthen the test time and increase the difficulty of the problem. To avoid excess fatigue, it may be best for such tests to have the authors work together for a short time each day over the course of many days.
- Test the system in a partly or fully adversarial environment, such as the drafting of contracts.

4.2 Improvements to the Collaborative Writing Program

This section outlines a number of possible improvements to the *CASTER* collaborative writing program. These features all have the potential to greatly increase the usability of the program. The possible features described here are a content subset browser, a collection of feature refinements suggested by the test results, addition of alternate communication channels, and integration of an outliner into the program.

4.2.1 Content Subset Browser

Although the text data in the *CASTER* collaborative writing system is presented as a single narrative sequence, the text itself is generated by different authors at different times. These differences define a number of interesting subsets that a user may wish to explore and manipulate. These subsets include the following:

- Everything new (not marked as read) in the view. By default, this subset is always highlighted in yellow.

- The contents of a single transaction: everything written by a single **Share** command.
- All contributions created within one view, or all contributions created within all views owned by one author.
- The current view's contrarian opinions. This feature could be used to predict trouble spots in merging this view with others. Part of the problem of developing this feature will be to determine a good definition of what makes a contribution "contrarian".
- A customized subset containing one or more of the above, such as multiple views or multiple transactions, selectable via a query interface.

The user interface for selecting and manipulating such a subset would minimally include the following features:

- A selector for subset type, where the types are as defined above.
- A function to select the subset matching the text at the current cursor position.
- Navigation between non-contiguous parts of the subset: these functions would minimally include first, last, previous, and next.
- Use of the Find command within the selected subset.
- Select-all and reject-all commands that apply to the subset.

4.2.2 Feature Refinements Indicated by Testing

This section summarizes the feature refinements indicated in the earlier sections of test results.

- Make the text segmentation for conflict sections interactively selectable within an existing view, instead of being stored in the database.
- Make every deletion, no matter how large, be a single segment for conflict display purposes. These segments could be displayed using a strike-through font as is done in Microsoft Word.
- Allow the user to choose a direct-manipulation interface for conflict resolution, avoiding the popup dialog box.
- Invent and implement an intuitive user interface for an atomic text-move operation. Such an interface might use the content subset browser described above to allow the user to browse among the potential locations for a block of text.

4.2.3 Alternate Communication Channels

For the purpose of controlled testing the *CASTER* Collaborative writing program allows no interaction between users outside of the written document content. The *QUILT* system[19, 36] and others provide a number of alternate channels to allow co-writers to communicate, including audio, video, and instant messaging. On the other hand, others have suggested that there is no benefit to including these channels in a monolithic application.[1] The effect of such additional channels on the effectiveness of the *CASTER* writing system, whether integrated into the application or provided separately, would make a worthwhile study.

[1] One practitioner, Brooke, states the case as follows:

> [D]esigners of CSCW systems should be striving to construct many small, interrelated applications, rather than building monolithic applications which encompass the whole spectrum of tasks that go to make up cooperative work in any particular sphere.[9, page 29]

In addition, *QUILT* and others have found that annotations greatly increase the effectiveness of collaboration. An annotation is a note which is not part of the artifact being created but which is anchored to some point within it. Threaded discussion within an annotation can be implemented simply by allowing an annotation to be anchored to another annotation. An annotation type would be straightforward to implement in the *CASTER* database, perhaps simply as a specially marked *AtomVersion*. Further study could evaluate how much annotations increase the effectiveness of the *CASTER* collaborative writing system.

4.2.4 Combination With an Outliner

One significant observation from the controlled tests was that some participants spent a lot of their time rearranging lists of items, which is an outlining task. These outlining actions were not universal: groups who generated their task solutions in the form of bulleted lists did outlining tasks, but groups who generated their solutions in narrative form did not. However, there is evidence that outlining is a part of most or all writing of any significant size. Lowry *et. al.*[37], in a nomenclature proposal based on an extensive literature review, suggest that all writing consists of three steps – planning, drafting and reviewing – and that the most common planning activity is outlining. Of these steps, the *CASTER* Collaborative Writing System effectively addresses the last two, drafting and reviewing.

The *CoVer* system described earlier (on page 16) implements multiversioning for hypertext. Their display mechanism consists of either showing a single version or showing two versions for comparison. Although the authors of *CoVer* assert that “Any application domain that can be modeled by interrelated objects can be mapped into hypertext.”[21, page 413], mapping each phrase to a hypertext node would cause problems with both

readability and modification of the text. Left with large single- or multi-paragraph modifiable object nodes – as were shown in the *CoVer* papers – the *CASTER* model should provide a useful complement within single hypertext nodes representing documents or paragraphs. A combination of features from these two systems would make a powerful next-generation collaborative writing system. Such a combined system would generate some significant user interface challenges, as two very different conflict-resolution mechanisms must be sufficiently rationalized that users can easily understand both of them.

4.3 Applying the *CASTER* Paradigm Beyond Collaborative Writing

The *CASTER* collaborative model has been shown to be useful for collaborative writing, but it is also applicable to other applications. This section discusses a sampling of promising applications for this model: hierarchical content, data repositories that never achieve consensus, and data bodies built collaboratively by storing and forwarding.

4.3.1 Hierarchical Document Content

While the *CoVer* project has demonstrated how multiversions can be implemented for a hierarchical outline or body of hypertext, as described on page 16, it should be noted that this approach is also fully compatible with the *CASTER* database structure, and could be used for any hierarchical data.

Although the current *CASTER* writing application is implemented as a flat list of sites, the *CASTER* document model supports a hierarchical organization of document elements. While sites in the writing application are assumed to be all of the same type (each site

currently represents one clause of text), a single site-type parameter could be added to the site record to support a structured hierarchy of data. The hierarchy features supported by the database could be used in many ways, including the following:

- An outliner: store an outline or the section structure of a document by making each subsection a child of its outline topic.
- Hypertext: store a site for each link as a child of the site that points to that link. Each site can be pointed to by multiple other sites.
- Formatted documents: implement text rendition, text fonts, and so on by making each such formatting directive a special type of site and making the formatted text clauses be children of the formatting directive site.
- A parser for a structured language: the *CASTER* database with support for hierarchy and site typing could be used to store a parse tree representing a document or program written in an arbitrary language.

Another research problem is that these different hierarchical organizations may interfere with each other if used together. For example, if a formatting directive begins in the middle of one outline point and ends the middle of another, then the outline points and the formatting directives cannot both be specified by one hierarchy without splitting either an outline point or a formatting directive.

4.3.2 Data Repositories That Never Achieve Consensus

The *CASTER* model is applicable beyond producing a single consensus document. It can also be used to develop a constellation of partially shared documents generated by groups of people with partially shared goals.

An example of such a system might be a position paper on the death penalty. Both pro-death-penalty and anti-death-penalty groups could use the same document. They could reject each others' conclusions and arguments while still sharing common data such as statistics and definitions of terms.

Another example of such a system would be a set of course outlines, curricula, or lecture slides shared by instructors at different schools. Each instructor would create a view for his or her course in which he or she would choose material provided by others and create new content to fill any gaps. There would be no definitive "consensus course"; rather, each instructor would create a course tailored to the needs of his or her school.

4.3.3 Data Sets Accumulated by Store and Forward

The *CASTER* database has no dependency loops in its causality determination: the time ordering of database elements, encoded in the ids of the database elements, can be expressed as a directed acyclic graph (DAG). It can be proved that two such databases representing the same document can be merged simply by iteratively increasing ids of database elements to preserve causality, and that this iterative process terminates. This algorithm is described in Appendix A on page 118.

This merge procedure can be used to replace the central database with a local data file that can either be sent to others or merged with data received from others, both on demand by the local user. Thus an update operation becomes a database merge operation instead of a database read operation.

Such a store-and-forward version of the *CASTER* application could establish point-to-point communication links with other *CASTER* application instances, but it would be more convenient and powerful to use an email or Usenet notes system as the transport for such communications.

This updating strategy enables an entirely new class of applications. The following sections are a sampling of the possibilities.

Sharing Cooperatively Built Configuration Information

There are some types of configuration information that quickly grow sufficiently large that one person cannot easily maintain them. Such configuration information could be built collaboratively by a community of cooperating users. A store-and-forward *CASTER*-style database could be used to allow people to evaluate the suggestions of others, probably incorporating most and rejecting a few.

Applications where this mechanism would be useful include the following:

- anti-spam address lists
- regular expressions to recognize spam
- web browser cookie accept/reject lists

Because these lists are fully under the control of their users, they are defensible against attack. For example, a spammer might wish to attack this system by sending out his or her own suggested anti-spam address list gleaned from the net, but with "allow" directives for the spammer's own addresses. In such a case, merging the spammer's contribution with those of others would highlight the "allow" directives, so that the recipient could not only reject them but also leave a record of the attack in the database as a warning to others.

Sharing Cooperatively Built Collections

The *CASTER* database format could also be used to cooperatively build useful reference collections from semi-structured atomic objects contributed by the participants. This

approach would work for any collection where overall coherence between the entries is unimportant. The following are some examples of objects that could be contributed by individuals and built into useful collections:

- dictionary definitions
- bibTex entries for citations
- playlists for CDs
- diagnostic flowcharts for doctors
- legal precedent flowcharts for lawyers

Again, a store-and-forward approach allows users to build personal collections from contributions by friends without relying on central repositories.

4.4 Extensions to the *CASTER* Collaborative Model

The function of the *CASTER* collaborative model, essentially, is to share work between collaborators while allowing the collaborators to ignore this shared work whenever they feel like doing so. The feature set that makes this possible is a fully visible history of changes by others that persists until the user chooses to process it. The most prominent feature within this set is the display of persistent conflict sections.

These features all work well as far as they go, but the true work of collaboration falls within this "processing" of the change history. For the small test documents used in this study the tools provided for processing changes have been fully adequate. However, this processing task will become increasingly onerous for large documents, contentious documents, or documents with large numbers of authors. In particular, such documents

will contain many conflict sections representing complex webs of interwoven changes by many authors. Choosing versions to resolve such conflicts will be a daunting task.

Fortunately, a large body of research work has been done recently, in many application domains, to address the fundamental problem of intelligently choosing from a large set of interrelated choices. The common thread in this work is the generation and application of decision support data. This data can be exposed to the user through two interfaces: bulk decision making and choice ranking by recommender systems.

4.4.1 Bulk Decision-Making

One decision often predicts another. If one person likes a sentence that another person wrote, there is a significant chance that he or she will choose to accept everything that that other person wrote during that session – or maybe everything that that other person wrote in the entire document. Great efficiency, along with some risk, could be introduced by allowing a user to make one decision to select or reject an entire class of data. The following paragraphs describe four such data classifications for which bulk decisions could be made. The resulting features are transaction aggregation, view inheritance, latest version selection, and view freezing.

Transaction aggregation is the simultaneous acceptance or rejection of all changes made in one other view during one session – i.e., between two successive invocations of the **Share** command. Such a set of changes is extremely likely to be self-coherent, and most likely will focus on a single theme. This feature probably will largely replace single-version selection for many users. This feature would need to be supported by a transaction browser as described in the earlier on page 88.

In view inheritance, a collaborator takes this aggregation approach one step further and specifies that all decisions and change made in another specified view be reflected

in the current view, the only exception being when the current view explicitly contains a conflicting decision or change. Inheritance so defined can be cyclic: this definition allows two or more collaborators who trust each other to inherit each others' views and thus share the task of resolving the conflicts in a rapidly changing document. Each conflict is resolved by whomever encounters it first. There is no practical barrier to such inheritance being recursive, so a recursion option could be offered to users as well. Recursive inheritance would simply involve inserting the inherited view's inherited view list into the current view's inherited view list.

The latest-is-best rule for resolving conflicts, as discussed earlier, is the method used by the class of simpler editing systems that the *CASTER* system was designed to replace. However, there are some cases where it would be useful to have an option to resolve all outstanding conflicts by this simplistic rule. Temporary application of this rule would allow a new participant to get a quick overview of a complex collaborative document. This rule might also be useful for a collaborator who decides that he or she completely trusts the other collaborators, and thus need not question their changes. This rule could also be useful in sections of the document about which the user has no interest.

Finally, version control can be implemented by having an option to freeze a view. In database terms, view freezing is one bulk decision to reject any contributions newer than a given timestamp, usually the timestamp of the freeze-view directive itself. This feature would allow a frozen version of an entire collaborative document to be preserved for publication, while allowing the collaborative writing database to remain active developing the next version.

4.4.2 Decision Support Information

Decision support information can be used to help the user answer the following two questions:

- Which version choice is best for me in this conflict section?
- Which other view is the best candidate for me to inherit?

The decision support information will provide not merely yes-or-no answers to these questions, but rather scores for each candidate (version choice or view, respectively) to help the user compare these candidates to each other. The decision support information can be presented as a list of candidates sorted by rank, optionally annotated with quality scores. The scored view information should be used to present a ranked list of desirable views for possible view inheritance, and the scored version information should be used to determine the display order of the choices in conflict sections.

These rankings can be done by either recommender systems or authority systems. Recommender systems rank others' opinions by similarity to one's own related opinions. Authority systems rank others' opinions by overall popularity.

Recommender Systems

Recommender systems are based on pattern matching mechanisms, which determine a similarity score between two vectors of choices. In general, an overall similarity score is accumulated by increasing it for matching choices and decreasing it for conflicting choices. Many systems modify the weights given to choices based on factors such as the frequency of the choice, the presence of another choice often found with the choice in question, or other more complex factors. The details of the actual matching algorithms

have become extremely complex, but the above description still sufficiently describes their function.

This technique has been applied in many application areas. The *GroupLens*[32, 42] Usenet news recommender asks each user to rate a small number of Usenet news postings and then uses these ratings to present other articles that were of interest to people with similar interests. Avery and Zeckhauser[5] added rewards for users who provide newsgroup ratings. Video recommender systems[28] have also been built on this principle.

These algorithms would be applied to *CASTER*-style views and choices by first computing scores for views and then using the view scores as weights when scoring choices made by those views. To find similar views, recommender system algorithms are applied to the vectors of all choices made by each view in the document, and the output is a vector of similarity scores between the current view and each other view. To find likely choices for the current view, each of the choices made by each other view is weighted by the choice's view's similarity to the current view, and these choices are accumulated for each possible choice in the current view.

Authority Systems

Authority systems rank views and opinions by overall popularity. The simplest authority system is straight voting: the quality of a choice is assumed to be directly proportional to the number of views that have chosen it.

A more comprehensive general-purpose approach is provided by Kleinberg's hubs and authorities model[30], whose basic algorithm can be summarized within this domain by the following pair of interdependent rules:

- A hub is a view that shares many opinions with other authorities.

- An authority is a view whose opinions are shared by many hubs.

These rules define an iterative process by which the quality of each view can be calculated as the degree to which it is a hub or an authority.

More ad hoc rules can be used to determine or modify view quality metrics, including the following:

- How much or how often a view has contributed content and/or choices
- A view's author's name, institutional affiliation, or other external information used by human users to assess the reputation of an author
- Other viewers' opinions of the author, such as the friends, fans, and foes model used by *slashdot*[1].

4.5 Summary and Conclusions

This dissertation has presented the following novel contributions:

- A model for collaboration in which all participants have full control of their linkage to their coworkers, designed to allow concentration when necessary within a collaborative setting.
- A prototype collaborative writing application which implements the above model, thus demonstrating the feasibility of the model.
- A body of test results that demonstrate the usability of this application, primarily in intensive two-person collaboration.

- A database design for this application that straightforwardly supports the handling of multiversions, and incidentally supports concurrent transactions and store-and-forward merging.

This work can provide a springboard for future work in many directions, including but not limited to the following:

- Further testing in different domains.
- Enhancements to the collaborative writing application.
- Integration of the collaborative writing application with a multiversion outliner.
- Application of the collaborative model to other application domains.
- Integration of the collaborative model with decision support technology.

The author believes that this project has opened a new area for research in collaborative systems, and looks forward to participating in further study in this area.

Appendix A

Database Design

This appendix contains a comprehensive description of the database used in the prototype CASTER collaborative writing system.

The novel features of this database design are as follows:

1. Support for an unlimited number of conflicting versions of every user-visible segment of text.

2. Support for an unlimited number of candidate positions within the document (i.e., within its display list) for every user-visible segment of text.

3. A forward-only cross-referencing construction that allows any collection of database changes to be written as a sequence of single-row-insert transactions, arbitrarily interleaved with other concurrent database changes from other users.

4. A novel mechanism for storing random insertions into an ordered list without requiring any changes to existing elements of the list.

This design can be implemented either as a relational database or as a flat file. It was implemented as a relational database in the early iterations of the prototype program, and was reimplemented as a flat file in later versions of the prototype.

The first section in this appendix will describe the database design as it was implemented in relational database format, because the terminology for relational databases is standardized and widely known. The next section will describe the ramifications of changing the database to a flat file format. The remaining section will describe possible database enhancements and extensions.

A.1 Database Design in *SQL*

The design presented here assumes that each CASTER document is stored in its own *SQL* database. The tables in this database can be logically divided into two sets: content tables and containment tables. The content tables hold the records of the text modification actions by which the users created the document. The containment tables provide the organizational framework for this content. The first two following subsections describe these two classes of tables.

The next two following subsections explain two important functions of the database that cannot be inferred from its record structure: specifically, how visibility of content is controlled and how the display list is built.

The final subsection shows the complete database design definition in the form of the sequence of *SQL* statements required to create it.

A.1.1 Containment Tables

The containment tables, *Author*, *View*, and *Transaction*, provide the organizational framework for the application data, telling the application how (or whether) to use it. This database is organized on the principle that an arbitrary number of participants may wish both to contribute new content and to filter the existing content displayed to them. Each such participant is an *Author*, and each display filter created by any *Author* is represented as a *View*. All new content is entered from within a *View*, and thus both contributions and filter choices are associated with a *View*. Each application user is intended to modify the text locally until he or she has a consistent *View* and then to commit all local changes to the database. The set of contributions and filter choices committed together within a *View* are grouped together as a *Transaction*. Figure A.1 shows this structure.

An *Author* record can be uniquely identified by its *username* field. A *View* record can be uniquely identified by its *author* and *name* fields. A *Transaction* record never needs to be looked up for reuse, so it does not need a unique identifier other than its database record *id*.

A.1.2 Content Tables

The content tables together store a journal of every text modification action performed within every view. These tables can be briefly described as follows:

- A *Slot* record provides a hub to connect all information about one segment of text.
- An *AtomVersion* record specifies a string of text that can potentially appear in its specified *Slot*. The *CASTER* collaborative writing program displays a conflict section when more than one of these records are visible for the same *Slot*.

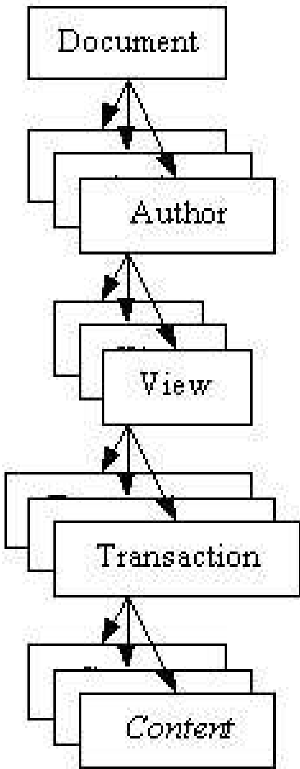

Figure A.1: Containment Structure

- A *SiteVersion* record specifies a document location in which the contents of a *Slot* may appear. *SiteVersion* records support atomic text-move and text-copy operations, which were not implemented in the *CASTER* prototype.

- An *AtomChoice* record specifies whether or not an *AtomVersion* record should be displayed in its creator's view. If an *AtomChoice* record hides an *AtomVersion* created within the same view, then this *AtomVersion* is repudiated and will be hidden in all views. Otherwise, the effect of an *AtomChoice* is local to the view that created it.

- A *SiteChoice* record specifies whether or not a *SiteVersion* is considered visible in its creator's view. As with *AtomChoice* records, a *SiteChoice* record's effect is local unless it repudiates.

A.1.3 Building the Display List

The *SiteVersion* records in a document together form a list that describes the structure of the document. The *anchorID* and *anchorType* fields in the *SiteVersion* record provide the information required to link the *SiteVersion* records together into a display list.

The *CASTER* application starts its display list by retrieving the first *SiteVersion* record in the document. This *SiteVersion* serves as the root node for the display tree. The application then links each remaining SiteVersion (in the order they were read from the database) immediately before or after the *SiteVersion* specified by its *anchorID*. The *anchorType* field specifies whether the *SiteVersion* appears before or after its anchor. If two or more *SiteVersion* records are linked before a common anchor, then the latest one appears closest to the anchor in the final display list. Similarly, if two or more *SiteVersion*

records are linked after a common anchor, then the latest one appears closest to the anchor in the final display list.

A.1.4 Controlling the Visibility of Content

Each *AtomChoice* or *AtomVersion* record affects the visibility of one or more *AtomVersion* records in its *Slot*. Similarly, a *SiteChoice* or *SiteVersion* record affects the visibility of one or more *SiteVersion* records in its *Slot*. This section will describe the form and function of the database fields that determine visibility. The design rationale for these functions is explained in Chapter 2, starting at page 39.

An *AtomChoice* record with a *choiceFlag* of **Accept** or **Reject** affects only the *AtomVersion* record that it specifies, and this effect is always local to the view in which the *AtomChoice* record was created. For these flag values, the *newestID* field in the *AtomChoice* record is ignored.

An *AtomChoice* record with a *choiceFlag* of **Choose**, however, implicitly rejects all other visible *AtomVersion* records in the same *Slot*. The *newestID* field specifies the newest *AtomVersion* record that was visible at the time the *AtomChoice* record was created. Thus any *AtomVersion* record newer than that specified by the *newestID* field is not affected by the *AtomChoice*, and remains visible. The design rationale for the function of the *newestID* field is explained in chapter 2 on page 48.

If an *AtomChoice* implicitly rejects another *AtomVersion* created within the same *View* as the *AtomChoice* record, then that *AtomVersion* is considered repudiated. The repudiated *AtomVersion* is hidden in all views, but all other records rejected by this directive (those created within other views) are only hidden locally.

An *AtomVersion* can also be hidden by replacing it with a newer *AtomVersion*. The semantics of this hiding are straightforward: Each *AtomVersion* record is treated as if

it was accompanied by an implicit *AtomChoice* record, choosing itself. This implicit *AtomChoice* record has a *choiceFlag* of **Choose**, and thus follows the rules described in the previous paragraph.

The visibility logic associated with *SiteChoice* and *SiteVersion* records is identical to that associated with *AtomChoice* and AtomVersion records. This logic is not used, however, because the *CASTER* prototype uses only one *SiteVersion* record per *Slot* and never creates *SiteChoice* records.

A.1.5 SQL table definitions

This section presents the complete database design definition in the form of the sequence of *SQL* statements required to create it.

```
CREATE TABLE Parameters (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  atomType ENUM('Phrase', 'Line', 'Phrase2'),
  databaseVersion INT NOT NULL);
# containment
CREATE TABLE Author (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  username CHAR(50) NOT NULL,
  password CHAR(20),
  UNIQUE (username));
CREATE TABLE View (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  authorID INT NOT NULL,
  name CHAR(10) NOT NULL,
  UNIQUE (authorID,name));
CREATE TABLE Transaction (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  viewID INT NOT NULL,
  timeCommit DATETIME NOT NULL,
  UNIQUE (viewID,timeCommit));
# content
```

```
CREATE TABLE Slot (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL);
CREATE TABLE AtomVersion (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT DEFAULT 0,
  slotID INT NOT NULL,
  content TEXT NOT NULL);
CREATE TABLE SiteVersion (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT DEFAULT 0,
  slotID INT NOT NULL,
  anchorID INT NOT NULL,
  anchorType ENUM('After', 'Before') NOT NULL
  UNIQUE (transactionID,slotID,anchorType,anchorID));
CREATE TABLE AtomChoice (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT NOT NULL,
  choiceID INT NOT NULL,
  flag ENUM('Rejected', 'Chosen', 'Accepted') NOT NULL,
  UNIQUE (transactionID,choiceID));
CREATE TABLE SiteChoice (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT NOT NULL,
  choiceID INT NOT NULL,
  flag ENUM('Rejected', 'Chosen', 'Accepted') NOT NULL,
  UNIQUE (transactionID,choiceID));
```

A.2 Update Handling

This section describes how data can be updated both to the database and from the database. The first subsection describes how commit (or share) operations are processed,

including the handling of race conditions. The second subsection describes how the data displayed locally in an application can be refreshed from the central database.

A.2.1 Commit Processing

The database design allows a self-consistent body of content of arbitrary size to be written to the database (committed) at any time. No matter how big this change list is, each database row on the list can be written by a separate database transaction. The following features work together to make this possible:

1. Auto-increment *id* fields: The primary key of every table is an auto-incremented integer. Thus each row of each table can be identified by one integer. If auto-increment fields are not supported by the database management system being used, they can be simulated by locking the table (which is otherwise unnecessary) and querying for the maximum *id* value.

2. Function to retrieve last written auto-increment *id*. After the commit, this value can be easily retrieved and stored into the client-side copy of the record using the mysql_insert_id() function. Most *SQL* database servers have a similar function.

3. Non-circular dependencies: The inter-table dependencies in this database are shown in figure A.2. There are no loops in this diagram, which means that no forward references need ever be written to satisfy such dependencies if each depended-on record is written before the record that depends on it. The only intra-table dependency is the dependence of a *SiteVersion* record on another *SiteVersion* record (its anchor), and in this case the anchor SiteVersion must exist before a new *SiteVersion* record can refer to it. Again, no forward reference need ever be written to satisfy such dependencies.

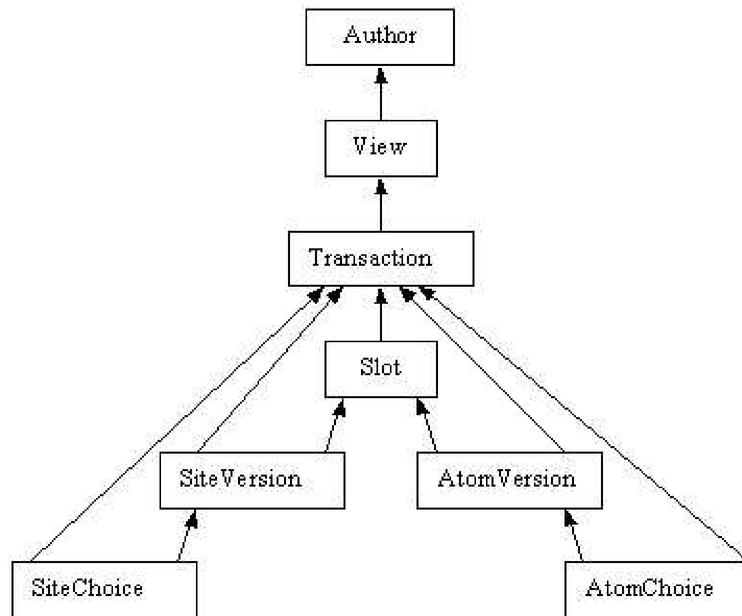

Figure A.2: Inter-Table Dependencies

In the middle of a commit operation (i.e., when some but not all of its component single-write transactions have been completed) the database may contain some non-functional constructs. However, these constructs are always self-consistent. Some examples of such a non-functional constructs are a *Slot* record used by no AtomVersion records, a *Slot* record used by no *SiteVersion* records, an unused Transaction record, an unused *View* record, or an unused *Author* record.

These records can be handled cleanly by the client application: they will be read and stored, but will generate no display changes. Furthermore, the remaining records to complete the partially read transaction will be picked up later when new records are read from the database, and at that time the client can construct the proper display changes just as if it had read the whole transaction at once.

Race Conditions

The following is a complete list of the race conditions that can occur when the single-write transactions of two or more commit operations are arbitrarily interleaved:

1. Document - two users try to create documents with the same name. The second user's CREATE DATABASE command will fail. The application can then prompt the user to select a new document name, and the commit can proceed with the new document name with no loss of data.

2. *Author* - two users try to claim the same author name in an existing document. The second user's INSERT ROW command in the Author table will fail. The application can then prompt the user to select a new author name, and the commit can proceed with the new author name with no loss of data.

3. *View* - one author tries to simultaneously create the same *View* from within two application instances. This is a user error, so the application has some latitude in handling it. The *CASTER* application ignores the failure of the second INSERT ROW command in the View table, thus putting the results of both commit operations into the same new *View*. An alternate approach would be to prompt the user for a new view name, allowing him or her to use the existing view if desired.

A.2.2 Reading the Database Into an Application

When an application using this database design starts running, it can get the data it needs by simply doing bulk reads of the database tables. If the bulk reads are done in an order such that each table is read after the tables that depend on it (see Figure A.2) then the application will have a self-consistent snapshot of the data even if other authors

are committing their results while the application is reading. This reverse-dependency-order reading will mean that the application must create dummy objects for dependencies during the read process, but will guarantee that all of these dependencies will be resolved by the time the read operation is complete.

To support dynamic updating, the application need only know the highest *id* number that it has read from each table. The *SQL* queries to do the dynamic update are then the exact same queries as those used for the initial read, except that they have an additional condition that the *id* number of the row read must be greater than the highest *id* number previously seen. Such an update will always get a self-consistent snapshot of the database, and is guaranteed to not miss any data committed before the latest data shown.

The Possibility of Incremental Reading

Instead of bulk reading each table, the application can choose instead to read the document data incrementally by traversing the *SiteVersion* tree of the document. This process makes many more database queries (one per table row instead of one per table) but might be justified for reading a very small subset of a huge document. Such a traversal should start at the first *SiteVersion* in the desired subset and proceed according to the following recursive algorithm:

```
readSite(SiteVersion t)
    if (display is not full)
        read the Slot for this SiteVersion
        find and read all AtomVersions associated with this Slot
        for each SiteVersion (tChild) anchored to this SiteVersion
            readSite(tChild)
        endfor tChild
    endif display is not full
end readSite()
```

Doing a dynamic update incrementally would involve re-traversing the SiteVersion tree and re-doing each query with the extra minimum-id condition described above. However, doing a bulk read for update would work fine in this case, even if the database were initially read incrementally. Such a bulk update would probably be the best approach unless the document update rate was on the same order as the (huge) document size.

A.3 Possible Enhancements

This section describes two possible database enhancements. The first subsection describes an alternate design for this database that is independent of the text segmentation. The second subsection describes a merge algorithm that enables the use of this database in a store-and-forward application.

A.3.1 A Segmentation-Independent Approach

The database design presented here requires that the segmentation rule be known and fixed before the database is generated. However, many of the concepts used in this database can be reused to create a multi-author, multi-version database independent of segmentation. This section will present one such design for document content.

The containment records for such a design – *Author*, *View*, and *Transaction* – will be identical to what they are in the *CASTER* database. The basic starting point for designing the content records is to assume that each character is its own segment, and then to aggregate *Slot* records (and their attendant *SiteVersion* and *AtomVersion*) together for efficiency.

One possible design resulting from such an approach is to create one database record for each text modification operation, regardless of how large or small the operation is. As

described earlier on page 31, the four basic text modification operations are add, delete, replace, and move. The move operation can be treated as a delete followed by an add, as it is in the existing *CASTER* prototype. The add and delete operations can be treated as special cases of the replace operation, where the *before* text or the *after* text, respectively, are null. Thus all that is needed for content representation is a replace record.

This *Replace* record, which would replace the *Slot*, *SiteVersion*, and *AtomVersion* records in the *CASTER* prototype would specify two things:

1. the *before* text
2. the *after* text

The *after* text can be represented simply as a variable-length array of text characters. The *before* text, however, must be specified as a range of existing text in the document. Such a range is best represented as a *start* position and an *end* position. Each of these positions, in turn, is best represented by another *Replace* record and a byte offset into that record. Thus the content of the *Replace* record so far is as follows:

1. the *after* text
2. the *Replace* record at which the replacement starts
3. an offset into this *start* record
4. the *Replace* record at which the replacement ends
5. an offset into this *end* record

To handle concurrent updates, a *newestID* field should be included. This field can be simply the newest committed *Replace* record in the document.[1]

[1]This field should ideally be the *id* of the newest committed *Replace* record whose scope overlaps the scope of the *Replace* record being created. However, the slight efficiencies realized by using this more

To create private views, a mechanism is required to accept or reject a *Replace* record in the database. This will be a *ReplaceChoice* record, similar to the *AtomChoice* record in the *CASTER* database.

Following the principles used in designing the *CASTER* prototype database, the formal *SQL* definition of these new records would be as follows:

```
CREATE TABLE Replace (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT,
  startID INT,
  startOffset INT NOT NULL,
  endID INT,
  endOffset INT NOT NULL,
  replacementText TEXT
);
CREATE TABLE ReplaceChoice (
  id INT AUTO_INCREMENT PRIMARY KEY NOT NULL,
  transactionID INT NOT NULL,
  newestKnownID INT,
  choiceID INT NOT NULL,
  choiceType ENUM('Reject', 'Choose', 'Accept') NOT NULL
);
```

This database approach creates significantly more work (and thus software complexity) for the client application. This extra work will include the following:

- The application must be able to figure out how to do segmentation itself. In the worst case, one single version presented to the user in a conflict section may contain pieces of multiple Replace operations.

specific *newestID* are probably outweighed by the inefficiency (and additional software complexity) of determining it.

- Conversely, the application must convert a user choice in a conflict section into a set of *Replace* and *ReplaceChoice* records, favoring *ReplaceChoice* records where possible.

- The application must keep track of highlighting for newness independently of segmentation, because a single new *Replace* record may cause parts of many segments to be highlighted as new.

A.3.2 Database Merge

The future work section on store-and-forward databases, on page 94, states that the *CASTER* database has no dependency loops in its causality determination: the required time ordering of database elements, encoded in the *id*s of the database elements, can be expressed as a directed acyclic graph (DAG). It can be proved that two such databases representing the same document can be merged simply by iteratively increasing *id*s of database elements to preserve causality, and that this iterative process terminates. This section provides that proof by outlining such an algorithm.

The input to this merge algorithm is two versions of a database, with a stem of common records (possibly null) after which each database version has different branch containing different records. The output of the merge algorithm is a database containing all records in the stem and both branches in such a way that all causality constraints are obeyed: i.e., there are no forward references.

The first step in the algorithm is to pick one of the database versions and copy it to the output. For efficiency, the larger of the two should be picked. We will call this database DB1. The second database, which we will call DB2, will then be copied after DB1, according to the following algorithm:

```
TL = translation list = empty
FOR each record (R2) in DB2 that is not in DB1
    write R2 to the end of the merged database
    give R2 a new offset greater than any others.
    write R2 and its old and new offsets to TL
ENDFOR R2
FOR each record (RT) in TL
    FOR each record (RR) between RT's old and new offsets
        FOR each cross reference (X) in RR
            IF X points to RT, update its reference
            [if this is a forward reference, then RT is already on TL]
        ENDIF
    ENDFOR X2
ENDFOR R2
```

The only inconsistencies in the database that this procedure can produce arise from the following conditions:

1. IF a record (R1) exists in the branch of DB1 containing a *newestID*

2. AND R1 refers to a *Slot* (S0) in the common stem

3. AND a record (R2) exists in the branch of DB2 containing a *newestID*

4. AND R2 also refers to *Slot* S0

5. AND the *newestID* field (N2) in R2 is also in the branch of DB2

6. THEN R2 will hide R1 (which it should not do)

This problem can be fixed by modifying the *newestID* field in each such record R2 to point to the last record in the main stem referring to that slot, and then adding *AtomChoice* or *SiteChoice* records owned within R2's *Transaction* that individually reject each of the intervening records containing a *newestID* that refer to the slot. This approach will not

hide entries in the other branch because the *newestID* record is not used in an *AtomChoice* or *SiteChoice* record when the *choiceFlag* for that record is **Reject**.

Appendix B

Test materials

During the controlled tests, the test administrator provided each of the participants with seven printed documents. These documents were a consent form, a test overview, descriptions of the two systems, two writing problem statements, and a questionnaire.

During the shakeout test, the test administrator provided each of the participants with two documents. These documents were a consent form and a questionnaire.

These materials are included as follows:

- Figure B.1 describes the procedure for the controlled test, and explains the use of all of the other documents provided to the participants in this test.
- Figure B.2 shows the consent form signed by each participant in the controlled tests. The forms given to the participants were printed on the official letterhead of the department of Computer Science at Wichita State University.
- Figure B.3 shows the consent form signed by each participant in the shakeout test. The forms given to the participants were printed on the official letterhead of the department of Computer Science at Wichita State University.

- Figure B.4 is the description of the control system given to test participants. This system was presented to participants as “system J”.
- Figure B.5 is the description of the CASTER system given to test participants. This system was presented to participants as “system Q”.
- Figure 3.1 on page 54 shows one of the writing assignments given to each test participant. Half of the participants received this assignment for the first writing task and half for the second.
- Figure 3.2 on page 54 shows the other writing assignment given to each test participant. Half of the participants received this assignment for the first writing task and half for the second.
- Section B.1 contains the text of the questionnaire that accompanied the controlled tests.
- Section B.2 tabulates the survey data generated from the controlled tests.
- Section B.3 contains the text of the questionnaire that accompanied the shakeout test.
- Section B.4 tabulates the survey data generated from the shakeout test.

CASTER user interface testing

Thank you for choosing to participate in this test.

PURPOSE

The purpose of this study is to compare two user interfaces for a
collaborative writing program.

PROCEDURE

You and one one collaborator will together complete two writing
tasks, one on each interface. You will also be asked to fill out
a questionnaire in four parts: one about your background, one
about each interface (2 copies), and one about your experiences
during the test.

More specifically, the steps will be as follows, with estimated times:
- (10 min) Initial briefing (this document)
- (2 min) Read and sign the consent form
- (3 min) Background questionnaire
- (5 min) Tutorial on first interface
- (3 min) Review of first problem
- (30 min) Collaborative writing for first problem
- (3 min) Questionnaire about first assignment and interface
- (5 min) Tutorial on second interface
- (3 min) Review of second problem
- (30 min) Collaborative writing for second problem
- (3 min) Questionnaire about second assignment and interface
- (5 min) Final questionnaire

Total time approximately 2 hours

”SCORING”

This is a test of interfaces, not a test of your abilities. There
are no wrong answers. We will compare the two documents that you
write to each other, but not to those written by others. Please
do your best on each writing assignment and don’t worry about the
results.

GROUND RULES

- I will set up the program for you, so you need not select a
filename, etc.

- When you are actually writing, you and your collaborator will
work in separate rooms. Please do not communicate with each
other during this time using anything other than the
collaborative writing program. You may use the program itself
to write messages to each other.

- You will have 30 minutes to complete each writing exercise.
However, if both you and your collaborator agree that you are
happy with the document that you have written before the 30
minutes are up then you can declare yourselves done early.

- Please don’t look at the second problem or the second interface
early, even if you figure out where I keep the files. Looking
at these files early will mess up the test.

- Please make sure that you complete every multiple-choice
question on each questionnaire. Skipping even a few questions
will make the data practically worthless.

Thank you again!

Figure B.1: Overall Test Description

CONSENT FORM

PURPOSE: You are invited to participate in a study of a prototype computer program for collaborative writing. I hope to learn which of two user interface approaches works better.

PARTICIPANT SELECTION: You were selected as a possible participant in this study because you are a college student. The total study will involve between 20 and 40 college students.

EXPLANATION OF PROCEDURES: If you decide to participate, you will receive instruction in how to use the system, watch a short video, and use the system to write summary of the video in collaboration with one other test subject. The entire test will take 1 hour per video, and one or two videos will be used for each test subject.

DISCOMFORT/RISKS: No discomforts or risks are anticipated.

BENEFITS: The subjects will gain some experience in collaborative writing, and the software may give them new and useful perspectives on this activity.

CONFIDENTIALITY: Any information obtained in this study in which you can be identified will remain confidential and will be disclosed only with your permission.

COMPENSATION OR TREATMENT: Not applicable.

REFUSAL/WITHDRAWAL: Participation in this study is entirely voluntary. Your decision whether or not to participate will not affect your future relations with Wichita State University. If you agree to participate in this study, you are free to withdraw from the study at any time without penalty.

CONTACT: If you have any questions about this research, you can contact me at: John Hainsworth, Room 237 Jabara Hall, Box 83, Wichita State University, 1845 Fairmount, Wichita KS 67260-0083, telephone 316-978-5325. If you have questions pertaining to your rights as a research subject, or about research-related injury, you can contact the Office of Research Administration at Wichita State University, Wichita, KS 67260-0007, telephone (316) 978-3285.

You are under no obligation to participate in this study. Your signature indicates that you have read the information provided above and have voluntarily decided to participate.

You will be given a copy of this consent form to keep.

______________________________	______________
Signature of Subject	Date
Signature of Parent or Legal Guardian (omit for subjects consenting for themselves)	Date
Witness Signature	Date
	Form A

Figure B.2: Consent Form for Controlled Tests

CONSENT FORM

PURPOSE: You are invited to participate in a study of a prototype computer program for collaborative writing. I hope to learn whether or not the program being tested is and effective aid to collaborative writing.

PARTICIPANT SELECTION: You were selected as a possible participant in this study because you are a college student. The total study will involve between 20 and 40 college students.

EXPLANATION OF PROCEDURES: If you decide to participate, you will learn about the design features of the system being evaluated and how to use the system to exploit these features effectively. You will then be asked to use the system to do collaborative writing projects of your choice. I will observe your work, ask you questions about it, and invite suggestions about the system. The collaborative writing will be done in multiple sessions, and I may improve the program between sessions based on test results. You will also be asked to fill out a questionnaire.

DISCOMFORT/RISKS: No discomforts or risks are anticipated.

BENEFITS: The subjects will gain some experience in collaborative writing, and the software may give them new and useful perspectives on this activity. The subjects will also gain some insight into how computer user interfaces are designed and tested.

CONFIDENTIALITY: Any information obtained in this study in which you can be identified will remain confidential and will be disclosed only with your permission.

COMPENSATION OR TREATMENT: Not applicable.

REFUSAL/WITHDRAWAL: Participation in this study is entirely voluntary. Your decision whether or not to participate will not affect your future relations with Wichita State University. If you agree to participate in this study, you are free to withdraw from the study at any time without penalty.

CONTACT: If you have any questions about this research, you can contact me at: John Hainsworth, Room 237 Jabara Hall, Box 83, Wichita State University, 1845 Fairmount, Wichita KS 67260-0083, telephone 316-978-5325. If you have questions pertaining to your rights as a research subject, or about research-related injury, you can contact the Office of Research Administration at Wichita State University, Wichita, KS 67260-0007, telephone (316) 978-3285.

You are under no obligation to participate in this study. Your signature indicates that you have read the information provided above and have voluntarily decided to participate.

You will be given a copy of this consent form to keep.

______________________________ _______________
Signature of Subject Date

______________________________ _______________
Signature of Parent or Legal Guardian Date
(omit for subjects consenting for themselves)

______________________________ _______________
Witness Signature Date

Form A

Figure B.3: Consent Form for Shakeout Test

INTERFACE ”J”

OVERVIEW

With this interface, whatever you type on your screen will be reflected on your collaborator’s screen in approximately 2 seconds.

FEATURES IN COMMON TO BOTH INTERFACES

- Marking text as read: New text written by your collaborator will appear highlighted in yellow. You can turn this highlighting off by selecting a region of text and using the ”Mark as read” command. If you prefer, you can ignore this feature and just leave highlighting on for all text changed by your collaborator.

- Text marked by author: Text written by you will be black and text written by your collaborator will be blue.

- Menus::Edit::Select all: This function will select all of the text, for deletion or marking as read.

- Menus::Edit::Find: This function will allow you to find a text string in your document.

- File::Quit: This function will terminate the program.

FEATURES SPECIFIC TO INTERFACE ”J”

No further features: text changes are exchanged automatically between you and your collaborator.

Figure B.4: Description of Control System

INTERFACE "Q"

OVERVIEW

With this interface, whatever you type on your screen will not be sent to your collaborator until you send them, and changes from your collaborator will not be incorporated into your document until you request them. Each section about which you disagree will be shown as a "conflict section", which will display two versions side by side.

FEATURES IN COMMON TO BOTH INTERFACES

- Marking text as read: New text written by your collaborator will appear highlighted in yellow. You can turn this highlighting off by selecting a region of text and using the "Mark as read" command. If you prefer, you can ignore this feature and just leave highlighting on for all text changed by your collaborator.

- Text marked by author: Text written by you will be black and text written by your collaborator will be blue.

- Menus::Edit::Select all: This function will select all of the text, for deletion or marking as read.

- Menus::Edit::Find: This function will allow you to find a text string in your document.

- File::Quit: This function will terminate the program.

FEATURES SPECIFIC TO INTERFACE "Q"

- Menus::File::Share (also a toolbar button): Make your latest editing changes available to your collaborator.

- Menus::File::Read (also a toolbar button): Incorporate the latest editing changes available from your collaborator.

- Menus::File::Discard: Discard any editing changes that you have not yet shared.

- Menus::Edit::Undo: Undo the last editing change that you have not yet shared.

- Menus::Edit::Redo: Undo the last Undo :-)

- Notes about Conflict sections: Each conflict section will contain one version of yours and one version of your collaborator's.

- If you type into a conflict section, then whatever you modified becomes your version. If you had a previous version, it will disappear.

- If you double-click within a conflict section, a dialog box will pop up allowing you to select someone else's version (In these tests, there will be only one choice). Selecting someone else's version repudiates your version, causing it to be removed from all views. The same effect can be achieved by typing and deleting one character in the middle of your collaborator's version.

- The "Apply and Advance" button in the dialog box will accept your collaborator's version and advance to the next conflict section. This feature allows you to quickly accept many changes from your collaborator.

Figure B.5: Description of CASTER System

B.1 Questionnaire for Controlled Test

Notes for the experimenter:

Date	_________
1. Document name	d________
2. Author name	a1 a2
3. First Problem:	A B
4. First Interface Tested:	J Q

This first group of questions asks about your background, your past experiences in writing, and how you find it easiest to express your ideas.
For each question, please circle the answer that best describes your experience.

5. Do you feel better able to express yourself in e-mail, or in conversation with another person?

Much better in e-mail	Somewhat better in e-mail	About the same	Somewhat better in conversation	Much better in conversation

6. Do you feel more able to express yourself in e-mail, or in writing a letter?

Much better in e-mail	Somewhat better In e-mail	About the same	Somewhat better in letter writing	Much better in letter writing

7. Do you feel more able to express yourself in e-mail, or in writing a paper or report?

Much better in e-mail	Somewhat better In e-mail	About the same	Somewhat better in a paper	Much better in a paper

8. When you write a paper or report, do you write and edit a number of versions, or just write it once?

I always write and edit more than 1 version	I usually write more than 1 version	I sometimes write more than version	I always just write it once and submit it

9. When you write a paper or report, do you write an outline first, or just begin writing?

I always write an outline first	I usually write an outline first	I sometimes write an outline first	I always just begin writing

10. When you write a paper or report, do you feel free to write what you really think, or do you write what you think the instructor or reader wants to hear?

I always write what I think	I usually write what I think	I balance the two	I usually write what I think they want	I always write what I think they want

11. When you write a paper or report, do you feel comfortable letting others (besides the grader or instructor) read what you've written?

Very uncomfortable	Somewhat uncomfortable	Neutral	Somewhat comfortable	Very comfortable

12. What is your gender? **Male** **Female**

Thank you for answering these questions. Now it is time to begin trying out the system you will be testing.

These questions ask you to describe your experience with the writing system you've just used.

13. Did you and your collaborator agree on the task you were trying to do with your writing?

We fully agreed	Our views seemed similar	We had some disagreement	We strongly disagreed

14. Do you feel confident that your own contributions and opinions are visible and accurately reflected in the final document?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

15. Did the act of collaboration make you feel you were under time pressure?

Under a great deal of time pressure	Under some time pressure	Neutral	I did not feel any time pressure

16. Did you feel free to write without worrying about what your collaborator was writing?

Felt very free to write without worrying	Felt somewhat free to write without worrying	Felt somewhat worried as I wrote	Felt very worried as I wrote

17. Did you edit yourself or leave things out because you knew your collaborator would see them?

I left out a great deal	I left out some things	I mostly wrote what I wanted	I didn't edit myself at all

18. How easy was it for you to understand the changes your collaborator made to the shared document?

Very easy	Easy	Somewhat difficult	Very difficult

19. Were you confident that your collaborator understood the changes you made?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

20. Did you feel that you and your collaborator had an equal role in deciding on the final product?

Very sure I had a larger role	Somewhat sure I had a larger role	Our roles were equal	Somewhat sure my collaborator had a larger role	Very sure my collaborator had a larger role

21. Were there areas where you "agreed to disagree" and decided not to insist on changes you wanted?

Many areas	Some areas	One or two areas	No, all the changes I wanted were included

22. Were there places where your collaborator wrote something that surprised you?

Many places	Some places	One or two places	None

23. Were there places where your collaborator included details or ideas you hadn't thought of?

Many places	Some places	One or two places	None

24. Do you feel that the document that you just finished writing is complete?

Seriously incomplete	Slightly incomplete	Pretty much complete	Done

25. Overall, how pleasant did you find this computer interface to use?

Irritating	Unpleasant	OK	Pleasant	Fun

26. Are there any other comments or ideas you would like to share about the system you have just used?

Thank you for answering these questions. Now we will continue with the testing process.

These questions ask you to describe your experience with the writing system you've just used.

27. Did you and your collaborator agree on the task you were trying to do with your writing?

We fully agreed	Our views seemed similar	We had some disagreement	We strongly disagreed

28. Do you feel confident that your own contributions and opinions are visible and accurately reflected in the final document?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

29. Did the act of collaboration make you feel you were under time pressure?

Under a great deal of time pressure	Under some time pressure	Neutral	I did not feel any time pressure

30. Did you feel free to write without worrying about what your collaborator was writing?

Felt very free to write without worrying	Felt somewhat free to write without worrying	Felt somewhat worried as I wrote	Felt very worried as I wrote

31. Did you edit yourself or leave things out because you knew your collaborator would see them?

I left out a great deal	I left out some things	I mostly wrote what I wanted	I didn't edit myself at all

32. How easy was it for you to understand the changes your collaborator made to the shared document?

Very easy	Easy	Somewhat difficult	Very difficult

33. Were you confident that your collaborator understood the changes you made?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

34. Did you feel that you and your collaborator had an equal role in deciding on the final product?

Very sure I had a larger role	Somewhat sure I had a larger role	Our roles were equal	Somewhat sure my collaborator had a larger role	Very sure my collaborator had a larger role

35. Were there areas where you "agreed to disagree" and decided not to insist on changes you wanted?

Many areas	Some areas	One or two areas	No, all the changes I wanted were included

36. Were there places where your collaborator wrote something that surprised you?

Many places	Some places	One or two places	None

37. Were there places where your collaborator included details or ideas you hadn't thought of?

Many places	Some places	One or two places	None

38. Do you feel that the document that you just finished writing is complete?

Seriously incomplete	Slightly incomplete	Pretty much complete	Done

39. Overall, how pleasant did you find this computer interface to use?

Irritating	Unpleasant	OK	Pleasant	Fun

40. Are there any other comments or ideas you would like to share about the system you have just used?

Thank you for answering these questions. Now we will continue with the testing process.

Thank you for taking the time to try out these computer programs and to share your experience in collaboration.

This last set of questions asks about your past experiences in working with others on projects.

41. Do you tend to prefer working independently, or working with others, on projects that matter to you?

Much prefer working on my own	Slightly prefer working on my own	Neutral	Slightly prefer working with others	Much prefer working with others

42. Do you tend to like or dislike collaborative or group projects when they are assigned in classes?

I love them	I like them	Neutral	Somewhat dislike them	I hate them

43. If you expressed a preference either way, why do you like or dislike these projects? Please use the space below to respond.

44. About how many group projects – a project involving you and at least one other person, lasting more than one day – do you think you have worked on in your lifetime? (These could be in school, or at work, or as part of a student activity, for example.)

none	one	2-4	5-10	more than 10

45. Approximately how many of these lasted a week or more?

none	one	2-4	5-10	more than 10

46. How many papers or documents have you written in your lifetime that were at least 5 pages in length?

none	one	2-4	5-10	more than 10

47. What is your collaborator's relationship to you?

we just met	acquaintance	friend	have done projects together	family, spouse, or 10+ year friend

48. What is your current student status?

freshman	sophomore	junior	senior	master's student	doctoral student

49. Are there any other comments or ideas you would like to share about the experiment in which you have just participated?

Thank you for your help in testing these computer programs!

B.2 Questionnaire Results for Controlled Test

Table B.1 shows the numerical questionnaire data from the controlled tests. The data was tabulated by assigning a value of 1 to the leftmost response for each question and assigning successive integer values to the other responses.

	Test number																	
Quest	1		2		3		4		5		6		7		8		mean	st.dev
1	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8		
2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1.5	
3	1	1	1	1	2	2	1	1	1	1	2	2	2	2	2	2	1.5	
4	1	1	2	2	2	2	1	1	2	2	2	2	1	1	1	1	1.5	
5	5	5	4	4	2	3	5	3	4	2	5	3	4	5	3	5	3.9	1.10
6	3	3	3	1	2	1	3	2	2	3	1	1	3	4	2	4	2.4	1.00
7	2	4	2	2	2	1	3	2	2	4	2	2	4	1	4	1	2.4	1.10
8	3	4	2	3	4	2	1	3	2	1	3	4	1	3	2	2	2.5	1.00
9	2	3	4	4	2	1	1	3	1	4	1	4	2	4	2	2	2.5	1.20
10	2	2	3	3	3	3	2	3	3	3	3	2	3	3	3	3	2.8	0.45
11	3	2	2	2	4	3	2	4	4	2	5	5	5	4	2	4	3.3	1.20
12	2	1	2	1	2	2	2	2	1	1	2	1	2	2	2	2	1.7	0.48
13	1	1	2	1	3	3	1	1	2	3	1	2	1	1	2	1	1.6	0.81
14	1	2	1	4	2	1	1	2	2	1	1	1	1	1	1	1	1.4	0.81
15	2	2	4	3	3	4	4	2	2	2	4	4	1	4	4	4	3.1	1.10
16	3	2	1	2	2	1	1	2	1	2	1	2	2	1	1	2	1.6	0.62
17	4	3	4	2	3	3	3	3	3	2	3	3	3	4	4	3	3.1	0.62
18	3	3	2	3	2	1	1	3	2	2	2	2	3	1	2	2	2.1	0.72
19	2	2	2	4	2	2	1	1	2	1	4	2	2	1	1	2	1.9	0.93
20	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	4	3.0	0.37
21	4	4	4	2	2	2	4	4	4	3	4	4	4	4	4	4	3.6	0.81
22	3	3	2	4	2	4	4	2	2	4	3	4	3	4	4	4	3.2	0.86
23	2	2	3	2	2	3	3	2	2	2	2	3	3	4	4	2	2.6	0.73
24	4	2	4	3	2	2	3	2	2	3	4	3	2	4	1	1	2.6	1.00
25	5	2	4	3	4	5	5	2	3	4	5	4	3	5	2	3	3.7	1.10
27	2	1	1	2	2	2	1	1	3	2	1	2	1	1	1	2	1.6	0.63
28	1	2	1	3	2	3	1	2	1	1	1	1	4	1	1	2	1.7	0.95
29	3	2	4	2	4	4	4	3	3	3	4	4	2	2	2	4	3.1	0.89
30	1	1	1	2	2	1	1	2	1	2	1	3	1	1	1	1	1.4	0.62
31	4	4	4	4	3	2	3	2	3	2	3	3	4	4	4	3	3.2	0.77
32	2	3	1	3	3	3	2	2	1	2	1	1	2	3	2	2	2.1	0.77
33	4	2	1	4	4	4	2	2	1	1	1	1	4	1	2	4	2.4	1.40
34	3	4	3	3	3	3	3	3	3	3	3	3	4	3	3	3	3.1	0.34
35	3	4	4	4	2	2	4	4	4	3	3	4	4	4	4	3	3.5	0.73
36	4	4	4	4	3	4	4	3	2	2	4	4	3	4	4	4	3.6	0.73
37	2	2	3	2	3	3	3	4	2	2	3	2	3	2	2	2	2.5	0.63
38	2	1	3	2	2	2	4	3	4	4	4	3	2	1	1	3	2.6	1.10
39	3	2	5	3	3	4	4	3	4	4	5	5	3	5	4	3	3.8	0.93
41	4	2	5	3	5	5	5	4	1	4	5	5	1	3	3	4	3.7	1.40
42	2	4	1	2	2	2	1	2	1	2	1	2	4	2	4	2	2.1	1.00
44	5	4	4	3	4	3	5	5	5	5	4	3	5	3	5	5	4.2	0.86
45	4	3	3	2	4	3	5	4	5	4	5	3	5	1	2	4	3.6	1.20
46	5	3	3	5	5	5	5	3	5	4	4	3	5	2	5	5	4.2	1.00
47	3	3	3	3	3	3	4	4	4	4	3	4	1	1	3	3	3.1	0.93
48	1	1	4	4	5	5	5	5	5	5	5	5	6	1	5	4	4.1	1.60

Table B.1: Questionnaire Data for Controlled Test

B.3 Questionnaire for Shakeout Test

This first group of questions asks about your background, your past experiences in writing, and how you find it easiest to express your ideas.
For each question, please circle the answer that best describes your experience.

1. Do you feel better able to express yourself in e-mail, or in conversation with another person?

Much better in e-mail	Somewhat better in e-mail	About the same	Somewhat better in conversation	Much better in conversation

2. Do you feel more able to express yourself in e-mail, or in writing a letter?

Much better in e-mail	Somewhat better In e-mail	About the same	Somewhat better in letter writing	Much better in letter writing

3. Do you feel more able to express yourself in e-mail, or in writing a paper or report?

Much better in e-mail	Somewhat better In e-mail	About the same	Somewhat better in a paper	Much better in a paper

4. When you write a paper or report, do you write and edit a number of versions, or just write it once?

I always write and edit more than 1 version	I usually write more than 1 version	I sometimes write more than version	I always just write it once and submit it

5. When you write a paper or report, do you write an outline first, or just begin writing?

I always write an outline first	I usually write an outline first	I sometimes write an outline first	I always just begin writing

6. When you write a paper or report, do you feel free to write what you really think, or do you write what you think the instructor or reader wants to hear?

I always write what I think	I usually write what I think	I balance the two	I usually write what I think they want	I always write what I think they want

7. When you write a paper or report, do you feel comfortable letting others (besides the grader or instructor) read what you've written?

Very uncomfortable	Somewhat uncomfortable	Neutral	Somewhat comfortable	Very comfortable

Thank you for answering these questions. Now it is time to begin trying out the system you will be testing.

These questions ask you to describe your experience with the writing system you've just used.

8. Did you and your collaborator agree on the task you were trying to do with your writing?

We fully agreed	Our views seemed similar	We had some disagreement	We strongly disagreed

9. Do you feel confident that your own contributions and opinions are visible and accurately reflected in the final document?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

10. Did the act of collaboration make you feel you were under time pressure?

Under a great deal of time pressure	Under some time pressure	Neutral	I did not feel any time pressure

11. Did you feel free to write without worrying about what your collaborator was writing?

Felt very free to write without worrying	Felt somewhat free to write without worrying	Felt somewhat worried as I wrote	Felt very worried as I wrote

12. Did you edit yourself or leave things out because you knew your collaborator would see them?

I left out a great deal	I left out some things	I mostly wrote what I wanted	I didn't edit myself at all

13. How easy was it for you to understand the changes your collaborator made to the shared document?

Very easy	Easy	Somewhat difficult	Very difficult

14. Were you confident that your collaborator understood the changes you made?

Very confident	Somewhat confident	Neutral	Somewhat doubtful	Very doubtful

15. Did you feel that you and your collaborator had an equal role in deciding on the final product?

Very sure I had a larger role	Somewhat sure I had a larger role	Our roles were equal	Somewhat sure my collaborator had a larger role	Very sure my collaborator had a larger role

16. Were there areas where you "agreed to disagree" and decided not to insist on changes you wanted?

Many areas	Some areas	One or two areas	No, all the changes I wanted were included

17. Were there places where your collaborator wrote something that surprised you?

Many places	Some places	One or two places	None

18. Were there places where your collaborator included details or ideas you hadn't thought of?

Many places	Some places	One or two places	None

19. How would you describe the task you were to do? Please use the space below to describe it.

20. Are there any other comments or ideas you would like to share about the system you have just used?

Thank you for answering these questions. Now we will continue with the testing process.

Thank you for taking the time to try out these computer programs and to share your experience in collaboration.

This last set of questions asks about your past experiences in working with others on projects.

21. Do you tend to prefer working independently, or working with others, on projects that matter to you?

Much prefer working on my own	Slightly prefer working on my own	Neutral	Slightly prefer working with others	Much prefer working with others

22. Do you tend to like or dislike collaborative or group projects when they are assigned in classes?

I love them	I like them	Neutral	Somewhat dislike them	I hate them

23. If you expressed a preference either way, why do you like or dislike these projects? Please use the space below to respond.

24. About how many group projects – a project involving you and at least one other person, lasting more than one day – do you think you have worked on in your lifetime? (These could be in school, or at work, or as part of a student activity, for example.)

none	one	2-4	5-10	more than 10

25. Approximately how many of these lasted a week or more?

none	one	2-4	5-10	more than 10

26. How many papers or documents have you written in your lifetime that were at least 5 pages in length?

none	one	2-4	5-10	more than 10

27. What is your current student status?

freshman	sophomore	junior	senior	master's student	doctoral student

Thank you for your help in testing these computer programs!

B.4 Questionnaire Results for Shakeout Test

Table B.2 shows the numerical questionnaire data from the shakeout test. The data was tabulated by assigning a value of 1 to the leftmost response for each question and assigning successive integer values to the other responses.

Quest		mean	std.dev.
1	2 3 5 5 2 4 5 5 1 4 3 1 4 5 5 5 5 2 2 2 3 4 2 3 1 4 5 3 3 3 3 5 3	3.4	1.30
2	1 3 4 3 2 2 3 1 1 3 1 1 1 2 3 1 3 3 2 1 2 2 2 3 3 2 1 1 3 1 3 3 3	2.1	0.93
3	1 4 2 1 2 2 3 5 3 3 1 1 1 3 2 3 5 3 4 2 2 2 3 3 3 1 1 3 1 1 4 1 3	2.4	1.20
4	3 1 2 1 1 3 2 4 4 1 1 4 2 1 1 2 2 4 3 2 2 1 1 3 3 2 1 2 3 3 3 2 2	2.2	1.00
5	4 4 2 1 3 3 3 1 2 1 4 1 3 4 5 3 1 4 4 2 1 1 2 4 2 3 4 1 1 4 2 2 1	2.5	1.30
6	2 3 4 4 2 3 2 3 2 3 4 1 2 3 3 1 2 2 2 3 3 2 4 2 1 1 1 2 1 2 3 3 3	2.4	0.93
7	3 3 4 1 3 4 2 3 5 3 3 3 5 2 4 5 5 2 5 4 4 4 2 4 2 1 1 2 4 5 2 5 5	3.3	1.30
8	2 3 2 2 2 1 2 3 1 1 1 1 2 3 1 2 3 2 3 1 2 3 3 2 2 1 1 3 2 1 2 2 1	1.9	0.77
9	4 2 2 3 1 1 3 4 2 2 1 2 2 3 1 3 2 1 1 2 2 1 4 1 4 2 4 1 2 1 2 2 2	2.1	1.00
10	4 2 4 4 2 4 4 2 3 4 4 2 2 2 3 2 4 3 4 4 3 2 2 4 2 4 4 4 3 2 2 2 3	3.0	0.92
11	1 1 2 1 2 1 1 2 1 1 2 3 3 2 2 1 3 2 1 1 2 1 3 1 3 1 1 3 1 2 2 3 1	1.7	0.80
12	3 3 3 3 2 4 3 2 4 2 2 2 2 3 3 3 3 3 4 2 3 3 3 4 4 4 2 2 4 3 3 1 1	2.8	0.85
13	2 2 2 3 2 1 3 2 3 2 3 2 2 3 1 3 2 2 2 3 2 2 3 3 2 3 2 3 2 2 2 3 3	2.3	0.60
14	4 4 2 2 3 1 4 2 4 1 4 3 3 3 2 3 2 3 3 4 3 1 1 4 1 2 3 3 1 2 3 2 2	2.6	1.00
15	3 3 3 3 3 1 3 3 3 3 3 1 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 1 3 3 3 3	2.8	0.60
16	4 3 2 2 3 2 4 3 2 4 0 1 3 2 2 4 4 4 3 2 3 3 2 4 4 4 3 3 4 2 4 2 4	2.9	1.00
17	2 3 2 3 3 2 4 2 4 4 4 2 3 2 4 2 3 4 3 2 3 4 2 4 4 4 4 2 1 2 2 2 4	2.9	0.95
18	2 3 2 2 2 2 4 2 3 4 2 1 3 2 3 4 1 4 3 2 3 4 2 3 4 2 2 1 1 2 4 2 4	2.6	1.00
21	3 3 4 4 4 3 4 4 5 2 4 5 1 3 3 2 2 2 1 5 3 1 2 2 1 1 4 2 4 2 5 4 5	3.0	1.30
22	3 2 2 2 2 3 2 4 2 2 2 3 4 4 3 4 2 3 4 2 3 2 4 4 5 5 3 5 2 3 2 2 2	2.9	1.00
24	2 4 3 5 4 4 3 3 4 3 3 4 5 5 5 0 4 3 5 3 4 5 5 4 4 4 5 5 5 4 3 4 3	3.8	1.10
25	4 3 2 5 3 4 3 3 3 3 3 3 5 3 5 0 4 3 5 3 4 4 3 3 4 4 4 3 4 3 3 3 5	3.5	1.00
26	4 3 5 5 3 4 3 4 4 3 5 4 5 5 4 0 4 3 5 5 3 5 5 4 5 5 4 4 5 5 2 3 4	4.0	1.10
27	5 4 5 5 5 4 4 5 5 5 5 4 4 3 5 0 4 4 4 5 5 5 4 4 5 4 5 4 4 4 4 5 5	4.3	0.96

Table B.2: Questionnaire Data for Shakeout Test

Bibliography

[1] Slashdot home page. http://www.slashdot.org/.

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

[3] Concurrent version system home page. http://www.cvshome.org/, May 2003.

[4] Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Main_Page, January 2006.

[5] C. Avery and R. Zeckhauser. Recommender systems for evaluating computer messages. *Communications of the ACM*, 40(3):88–89, 1997.

[6] U. Bauer, M. Ott, M. Pittenauer, and D. Wagner. Hydra. http://hydra.globalse.org/, April 2003.

[7] E. A. Bier and S. Freeman. MMM: a user interface architecture for shared editors on a single screen. In *Proceedings of the Fourth Annual ACM Symposium on User Interface Software and Technology*, pages 79–86. ACM Press, 1991.

[8] G. E. Bock and D. A. Marca. *Designing Groupware: A Guidebook for Designers, Implementors, and Users*. McGraw-Hill, New York, 1995.

[9] J. Brooke. Chapter 2: User interfaces for CSCW systems. In D. Diaper and C. Sanger, editors, *CSCW in Practice: An Introduction and Case Studies*, London, 1993. Springer-Verlag.

[10] F. P. Brooks Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, Boston, 1995.

[11] D. Chen and C. Sun. A distributed algorithm for graphic objects replication in real-time group editors. In *GROUP '99: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 121–130, New York, NY, USA, 1999. ACM Press.

[12] J. Conklin and M. L. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. *ACM Transactions on Information Systems (TOIS)*, 6(4):303–331, 1988.

[13] A. Dattolo and A. Gisolfi. Analytical version control management in a hypertext system. In *CIKM '94: Proceedings of the Third International Conference on Information and Knowledge Management*, pages 132–139, New York, NY, USA, 1994. ACM Press.

[14] A. Dillon. How collaborative is collaborative writing? an analysis of the production of two technical reports. In M. Sharples, editor, *Computer Supported Collaborative Writing*, London, 1993. Springer-Verlag.

[15] D. Dobkin. Interview, August 2004.

[16] P. Dourish. Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM Transactions on Computer-Human Interaction*, 5(2):109–155, 1998.

[17] J. E. James Whitehead. Design spaces for link and structure versioning. In *HYPERTEXT '01: Proceedings of the Twelfth ACM Conference on Hypertext and Hypermedia*, pages 195–204, New York, NY, USA, 2001. ACM Press.

[18] R. Eckenrod. Reaching consensus on the Tampa Bay Estuary Program Interlocal Agreement: A perspective. In M. P. Mandell, editor, *Getting Results Through Collaboration*, Connecticut, 2001. Quorum Books.

[19] R. S. Fish, R. E. Kraut, and M. D. P. Leland. Quilt: a collaborative tool for cooperative writing. In *Conference Sponsored by ACM SIGOIS and IEEECS TC-OA on Office Information Systems*, pages 30–37. ACM Press, 1988.

[20] J. Grudin. Why CSCW applications fail: Problems in the design and evaluation of organizational interfaces. In *CSCW '88: Proceedings of the 1988 ACM Conference on Computer-Supported Cooperative Work*, pages 85–93, New York, NY, USA, 1988. ACM Press.

[21] A. Haake and J. M. Haake. Take CoVer: Exploiting version support in cooperative systems. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 406–413. Addison-Wesley Longman Publishing Co., Inc., 1993.

[22] A. Haake and D. Hicks. VerSE: Towards hypertext versioning styles. In *HYPERTEXT '96: Proceedings of the Seventh ACM Conference on Hypertext*, pages 224–234, New York, NY, USA, 1996. ACM Press.

[23] J. M. Haake and B. Wilson. Supporting collaborative writing of hyperdocuments in SEPIA. In *CSCW '92: Proceedings of the 1992 ACM Conference on Computer-Supported Cooperative Work*, pages 138–146, New York, NY, USA, 1992. ACM Press.

[24] U. Hahn, M. Jarke, S. Eherer, and K. Kreplin. CoAUTHOR: a hypermedia group authoring environment. In J. M. Bowers and S. D. Benford, editors, *Studies in Computer Supported Cooperative Work: Theory, Practice and Design*, Amsterdam, 1991. North-Holland.

[25] F. G. Halasz. Reflections on notecards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, 1988.

[26] J. Harris and A. Henderson. A better mythology for system design. In *Proceeding of the CHI 99 Conference on Human Factors in Computing Systems : the CHI is the Limit*, pages 88–95. ACM Press, 1999.

[27] B. Hewitt and G. N. Gilbert. Chapter 3: Groupware interfaces. In D. Diaper and C. Sanger, editors, *CSCW in Practice: An Introduction and Case Studies*, London, 1993. Springer-Verlag.

[28] W. Hill, L. Stead, M. Rosenstein, and G. Furnas. Recommending and evaluating choices in a virtual community of use. In *Conference Proceedings on Human Factors in Computing Systems*, pages 194–201. ACM Press/Addison-Wesley Publishing Co., 1995.

[29] W. C. Hines and D. C. Montgomery. *Probability and Statistics in Engineering and Management Science*. John Wiley & Sons, New York, second edition, 1980.

[30] J. M. Kleinberg. Hubs, authorities, and communities. *ACM Comput. Surv.*, 31(4es):5, 1999.

[31] D. E. Knuth. *The TEXbook*. Addison-Wesley, Reading Massachusetts, 1984.

[32] J. A. Konstan, B. N. Miller, D. Maltz, J. L. Herlocker, L. R. Gordon, and J. Riedl. GroupLens: Applying collaborative filtering to Usenet news. *Communications of the ACM*, 40(3):77–87, 1997.

[33] S. Kristoffersen and F. Ljungberg. An empirical study of how people establish interaction: Implications for CSCW session management models. In *Proceeding of the CHI 99 conference on Human factors in computing systems : the CHI is the limit*, pages 1–8. ACM Press, 1999.

[34] A. Lamott. *Bird by Bird: Some Instructions on Writing and Life*. Anchor Books, New York, 1995.

[35] J. Lee. SIBYL: a tool for managing group design rationale. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 79–92. ACM Press, 1990.

[36] M. D. P. Leland, R. S. Fish, and R. E. Kraut. Collaborative document production using Quilt. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 206–215. ACM Press, 1988.

[37] P. B. Lowry, A. Curtis, and M. R. Lowry. Building a taxonomy and nomenclature of collaborative writing to improve interdisciplinary research and practice. *Journal of Business Communication*, 41(1):66–99, January 2004.

[38] T. Moran, K. McCall, B. van Melle, E. Pederson, and F. Halasz. Design principles for sharing in Tivoli, a whiteboard meeting-support tool. In S. Greenberg, S. Hayne, and R. R., editors, *Designing Groupware for Real Time Drawing*. McGraw-Hill, 1995.

[39] C. M. Neuwirth, D. S. Kaufer, R. Chandhok, and J. H. Morris. Computer support for distributed collaborative writing: Defining parameters of interaction. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 145–152. ACM Press, 1994.

[40] E. R. Pedersen, K. McCall, T. P. Moran, and F. G. Halasz. Tivoli: an electronic whiteboard for informal workgroup meetings. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 391–398. Addison-Wesley Longman Publishing Co., Inc., 1993.

[41] R. Pike. The text editor *sam*. *Software - Practice and Experience*, 17(11):813–845, 1987.

[42] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. GroupLens: an open architecture for collaborative filtering of netnews. In *Proceedings of the Conference on Computer Supported Cooperative Work*, pages 175–186. ACM Press, 1994.

[43] C. Schuckmann, L. Kirchner, J. Schümmer, and J. M. Haake. Designing object-oriented synchronous groupware with COAST. In *CSCW '96: Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, pages 30–38, New York, NY, USA, 1996. ACM Press.

[44] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison Wesley Longman, Massachusetts, third edition, 1998.

[45] M. Stefik, D. G. Bobrow, G. Foster, S. Lanning, and D. Tatar. WYSIWIS revised: Early experiences with multiuser interfaces. *ACM Transactions on Information Systems (TOIS)*, 5(2):147–167, 1987.

[46] J. Stewart, B. B. Bederson, and A. Druin. Single display groupware: a model for co-present collaboration. In *Proceeding of the CHI 99 Conference on Human Factors in Computing Systems : the CHI is the Limit*, pages 286–293. ACM Press, 1999.

[47] C. Sun. Undo any operation at any time in group editors. In *Computer Supported Cooperative Work*, pages 191–200, 2000.

[48] C. Sun and D. Chen. A multi-version approach to conflict resolution in distributed groupware systems. In *International Conference on Distributed Computing Systems*, pages 316–325, 2000.

[49] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(1):1–41, 2002.

[50] S. G. Tammaro, J. N. Mosier, N. C. Goodwin, and G. Spitz. Collaborative writing is hard to support: A field study of collaborative writing. *Computer Supported Cooperative Work*, 6(1):19–51, 1997.

[51] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[52] E. S. Veinott, J. Olson, G. M. Olson, and X. Fu. Video helps remote work: Speakers who need to negotiate common ground benefit from seeing each other. In *Proceeding of the CHI 99 Conference on Human Factors in Computing Systems : the CHI is the Limit*, pages 302–309. ACM Press, 1999.