SHARING SEMI-HETEROGENEOUS SINGLE-USER EDITORS FOR REAL-TIME GROUP EDITING

A Thesis

by

JIAJUN LU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2005

Major Subject: Computer Science

SHARING SEMI-HETEROGENEOUS SINGLE-USER EDITORS FOR REAL-TIME GROUP EDITING

A Thesis

by

JIAJUN LU

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Du Li (Chair of Committee)	Frank Shipman, III (Member)
Susan Pedersen (Member)	Valerie E. Taylor (Head of Department)

May 2005

Major Subject: Computer Science

ABSTRACT

Sharing Semi-heterogeneous Single-user Editors for

Real-time Group Editing. (May 2005)

Jiajun Lu, B.S., Fudan University

Chair of Advisory Committee: Dr. Du Li

A new approach is proposed to transparently share familiar single-user editors without modifying their source code. This approach tweaks a classic diff algorithm to derive edit scripts between document states. Concurrent edit scripts are merged to synchronize states of coauthoring sites. Our concept-proving prototype currently works with familiar, heterogeneous text editors such as GVim and WinEdt that can be adapted to support two basic interfaces, GetState and SetState. The adaption is less expensive and more robust than recent approaches such as ICT and CoWord, which must understand and translate editing operations at the operating system level. Experimental data show that our approach is able to provide sufficient performance for near-realtime group editing.

To my wife Fei Wu, my mother Aifeng Chen and the memory of my father Jiulian Lu

ACKNOWLEDGMENTS

I would like to thank my advisor and committee chair, Dr. Du Li, for continuous guidance. Without his help, this thesis would have never been completed. I am grateful to the members of my committee, Dr. Frank Shipman and Dr. Susan Pedersen, for their valuable suggestions on this thesis. I also wish to thank my collaborators and friends, Rui Li and Yi Yang, for help with miscellaneous technical questions. I would like specially thank my wife Fei for her unconditional love and support through the years.

TABLE OF CONTENTS

Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Many people write together as part of their day-to-day routines[1, 2]. Artifacts of group editing include source code, software documentation, research papers, music scores[3], and online encyclopedia (e.g., http://www.wikipedia.org). Hence there is a huge, potential market if usable group editors are available. This has long been confirmed by the continuing research interests on group editors[4, 5, 6, 7, 8] since the beginning of the Computer Supported Cooperative Work(CSCW) field.

Despite these efforts, however, analyses and studies repetitively show that specialized group editors have been underused[2, 9]. Most of people still use single-user editors for group writing today. The main reasons, as argued in [9, 10], include that group editors generally are not as powerful as familiar single-user text editors or word processors, and that people may not want to learn new user interfaces. Grudin[9] suggests that a more promising way is to build collaboration features into accepted productivity tools. Along this line, two research groups recently explored how to transparently adapt existing single-user editors for group editing without modifying their source code[7, 10].

While these designs are plausible, much progress can still be made in two directions: One is how to transparently adapt single-user editors and the other how to share heterogeneous editors. The rich variety of available text editors[11] and word processors [12] testifies that people have different preferences. Allowing for heterogeneity in group editing increases flexibility and potentially group productivity. Although it is not uncommon today that many people are familiar with several products and can

---

The journal model is *IEEE Transactions on Automatic Control.*

possibly use the same editor for group editing, this does not automatically solve the problem of transparently adapting single-user editors.

In this thesis, a novel approach is proposed to transparently adapting familiar single-user editors for group editing. The editors in question could be homogeneous but are also allowed to be heterogeneous. It uses a slightly modified version of the diff algorithm[13] to derive edit scripts between document states. Concurrent edit scripts are merged at synchronization time. Our approach only assumes two basic interfaces, GetState and SetState, to capture editor state before diffing and reset the editor state after merging respectively. Synchronization can be triggered automatically or initiated manually in group editing.

Text editors and word processors serve different editing purposes and user communities. For example, source code and system configuration files are generally edited with text editors. As another example, in academic coauthoring, while many conferences and journals provide Word templates, many others to our knowledge only provide Latex templates. Supposedly not many people want to use word processors instead of text editors to edit Latex-style documents. Our approach only address how single-user editors (text editors and word processors) are adapted to support group editing. Although it is possible that a group of coauthors use word processors and text editors to edit the same document, translation of different formatting styles (e.g., between Latex and Word) is out of the scope. In this sense, editors in question are only semi-heterogeneous: the content is considered more important than the user interfaces and formattings provided in specific editors.

A. Motivation and Related Works

Recent word processors come with built-in collaboration features. For example, Microsoft Word provides features for synchronous and asynchronous group editing. It allows users to set up online meetings to work in real time. However, synchronous work is largely enabled by NetMeeting, an application sharing technology that enforces a strict what-you-see-is-what-I-see(WYSIWIS) type of collaboration[14], which may result in low system performance and low group productivity in many situations[15]. Microsoft Word also allows users to track and review changes, add comments, compare and merge documents. However, these asynchronous group editing features may only be used with other Microsoft Word users. Differences in word processors or even versions may cause problems in group writing[2].

Placeless Documents project[16] implements mechanisms for document-centered collaboration. It intercepts document-level events, e.g., open and close, and uses these events to drive high-level workflows. Hence specific editors that are invoked to edit documents are not relevant. That is, heterogeneous single-user editors can be used to edit the same document. However, it only supports asynchronous group editing and is complementary to this work.

Our previous work, Intelligent Collaborative Transparency or ICT[10], pioneers the research of transparently adapt familiar, heterogeneous single-user applications for cooperative work. However, it assumes that keyboard and mouse events can be correctly understood and translated into abstract operations. Then the abstract operations obtained at one site are translated at other sites into events that achieve equivalent editing effects. Due to the tremendous difficulties in understanding application behavior at the operating system level, current prototype of ICT only allows for very limited functionality.

The CoWord project[7] adapts Microsoft Word(as well as other Microsoft and StarOffice productivity tools) into realtime group editors. Heterogeneity issues are not addressed. These productivity tools generally provide APIs for third parties to develop add-on features. CoWord uses these APIs to help formalize application behavior and its approach is thus much easier than ICT. Nevertheless, it still needs to understand and translate the keyboard/mouse events into APIs, which is still difficult. As a result, CoWord has to disable many user interface features and has difficulties in making timely responses to version upgrades of Word.

By comparison, our approach resembles Placeless Documents in terms of document-centered collaboration but differs in the capability of supporting realtime group editing. It is more light-weight and robust than the approaches taken by ICT and CoWord in the amount of work to adapt single-user applications. It also differs CoWord in terms of supporting heterogenous editors. This approach only assumes two simple interfaces, GetState and SetState. Most text editors and word processors to our knowledge can be easily adapted to support these interfaces.

Since this new approach still needs to formalize application knowledge in some way, it can be considered the second generation of intelligent collaboration transparency (ICT2). However, differently from the original ICT, ICT2 does not attempt to intercept and understand the operating system level events. Instead, it uses an adapted version of the diff algorithm [13] to derive the edit script between document states.

B. Approach Overview

The basic idea of ICT2 is to use a diff algorithm to analyze the edit operations instead of capturing all of operations. As shown in Figure 1, suppose two users edit a shared

document from the state S_0. One user uses GVim, the other uses WinEdt. The shared document replicated on each site.

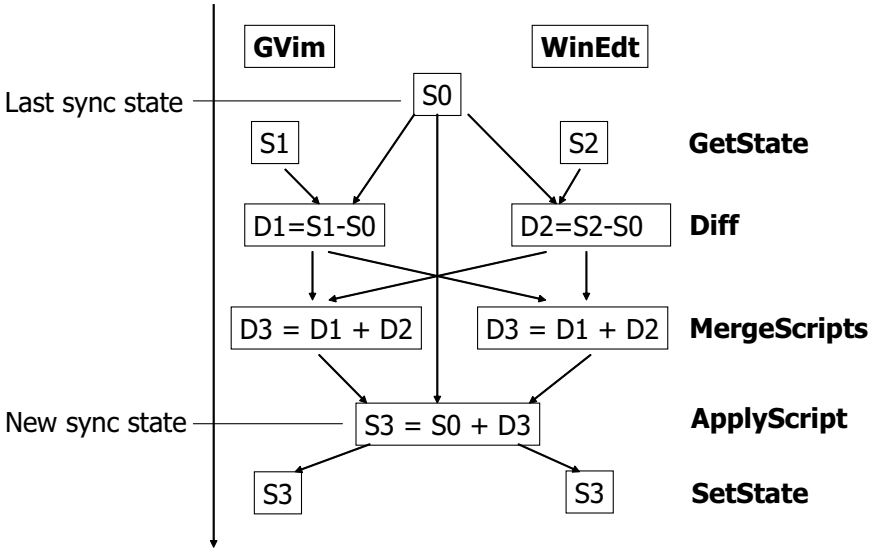

Fig. 1. Basic ideas of ICT2

They work in parallel and reach two different states, S_1 and S_2, respectively. When a sync is initiated, we obtain the new editor state through the GetState interface and compute the edit script by diffing at each site. Suppose the two edit scripts are $D_1 = S_1 - S_0$ and $D_2 = S_2 - S_0$. After exchanging edit scripts between each site, we merge D_1 and D_2 to into D_3, and apply D_3 to the last sync state S_0, which results in S_3. Then we call the SetState interface at both sites to set the editors to S3, which integrates the concurrent changes made at both sites.

Based on this idea, ICT2 system is designed as shown in Figure 2. A group editing session consists of a session manager and a number of clients. Each client runs an agent and an adapter, which collectively provide interfaces for the user to share his/her familiar single-user editor for group editing.

Initially only one client is active. The user edits a document alone using a familiar single-user editor. When other users need to edit the document, they contact

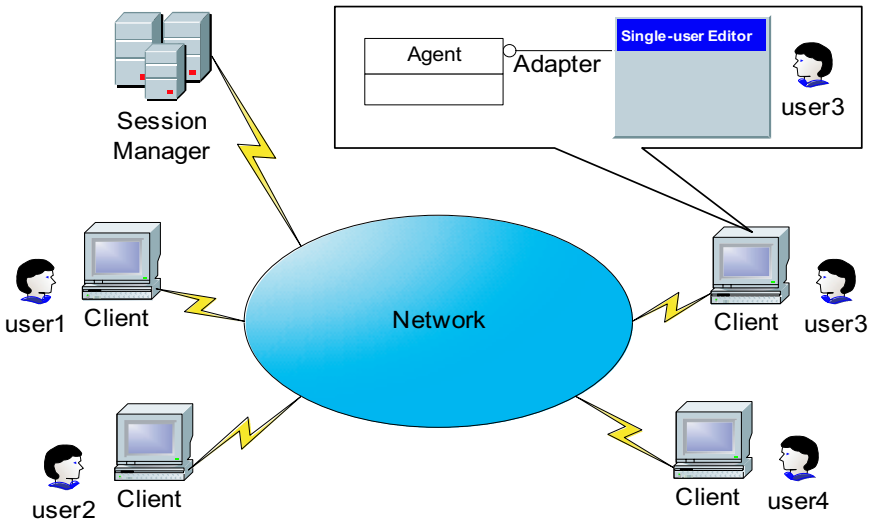

Fig. 2. Sharing familiar single-user editors for group editing

the first client, which launches the session manager. The session manager provides document and session management services such that coauthors can register and log on[17]. Then the shared document is replicated at other sites and loaded into the local single-user editors.

The adapter at each site adapts a single-user editor. It provides two simple interfaces between the agent and the editor, GetState and SetState, for the agent to get and set the editor state, respectively. In our concept-proving prototype, we have adapted GVim and WinEdt, two popular single-user text editors. We choose these two editors mainly for two reasons: First, these two editors are "typical" in that GVim provides APIs while WinEdt does not. We will be able to demonstrate the generality of our approach, whether or not the editors provide APIs. Second, we ourselves are familiar with them and use them for coauthoring papers all the time. We are motivated to use and improve our own system and there is no disparity between work and benefit.

The agents at all sites collaborate to implement synchronization and awareness

control. Each agent interacts with the local single-user editor through the adapter, provides awareness information to the user, accepts commands from the user, and communicates with other agents in the same session.

Since the agent is a separate process external to the single-user editor, it does not change the editor's behavior or user interface. The user's experience with the editor remains the same most of the time: he uses a familiar user interface and focuses on his own part of the work. At the same time he is notified of his coauthors' status and progress through a separate user interface. He is somewhat "disturbed" only when synchronization occurs and coordination becomes necessary.

C. Outline of the Thesis

The remainder of this thesis is organized as follows. In Chapter II, I will first introduce the background related to ICT2. Then I will describe the agent of ICT2 and how it works. After that, I will present how edit scripts are derived and merged in Chapter IV. Particularly, I will show that the classic diff algorithm[13] can be slightly adapted to provide sufficient performance for realtime group editing in most practical situations, and discuss how the merged editing sequences are applied to the document and presented to the users. In Chapter V, I will show two awareness mechanisms, synchronization report and progress report, supported by ICT2 to enhance user cooperation. Next, I will present how to adapt single-user editors to support GetState and SetState interfaces in Chpater VI. Finally, Chapter VII summarizes the main contributions and the limitation, and then points to the future direction of research.

CHAPTER II

BACKGROUND

The booming of computer information and communication technology promotes cooperation between people to the success of most organizations. With the aid of computers, cooperation could be much easier than before. Computer supported cooperative work(CSCW) is a research field on design, introduction, and use of technologies which affect groups, organizations, communities, and societies[18]. CSCW focuses on building groupware technologies as well as their psychological, social, and organizational effects[19]. Typical CSCW applications include E-mail, Web publishing, video conferencing, electronic calendars, workflow system, and knowledge sharing system[19].

ICT2 implements a realtime group editing system which allows multiple users to work on the same document. It is based on application sharing technology, which adapts single-user application into group environment. ICT2 uses an adapted version of the classic diff algorithm[13] to analyze the editing operations, and applies the idea of operational transformation(OT) algorithm[20] for concurrency control.

In the following sections, I will give the background information about group editors, application sharing systems, the diff algorithm, and OT.

A. Group Editing

Group editing is a classic research topic in CSCW. Many researchers use group editing system as models and research vehicles of a wide range of collaborative systems[5, 6]. Challenging issues in group editing range from the technical to the social. Technical issues include system design and concurrency control algorithms. Social issues include how people write together[1, 2]

Generally, group editing systems can be categorized into asynchronous systems where coauthors are separated by relatively long periods, and synchronous systems where coauthors interact simultaneously or are separated by short periods of time[19].

Asynchronous group editing systems provide support for users to control, communicate, and track changes. E-Mail, Wiki, and Concurrent Versioning System(CVS) fall into this category.

Synchronous group editing systems allow users to edit the same document at the same time. They don't enforce users to take turns to edit documents. In the last decades, the research of CSCW has been focusing on synchronous group editing, which is represented by Grove[21] and Reduce[22]. They replicate the shared document at all cooperating sites and allow any editing operations to execute on any part of the document at any time. As a result, group editors can often achieve high local responsive and concurrency.

However, Grove and Reduce, those specific realtime group editors lag behind well-accepted single-user editors in features and compatibility. For example, there's no realtime group editors with competitive editing features as Microsoft Word or GVim. As group features are often used less frequently than features supporting individual activities[23], it discourages many users by forcing them to learn new user interfaces for sporadic tasks. Therefore, a more plausible way is thus to incorporate single-user editors with groupware features[19].

The approach to adding groupware features to single-user applications falls into two categories, collaboration-awareness and collaboration-transparency. The former requires access to proprietary source code, which in practice may be impossible to acquire. Thus, collaboration transparency, also called application sharing[24], appears a more promising alternative in many situations.

B. Application Sharing

Application sharing systems could adapt single-user applications into collaborative systems. Due to the increasing demand of collaboration technologies, sharing mature and popular single-user systems naturally becomes an important method in building collaborative systems. From the late 1960's through the late 1990's, many application sharing prototypes and products have been developed, such as NLS[25], MMConf[23], XTV[26], and Flexible JAMM[15], as well as successful commercial application sharing products such as Microsoft Netmeeting and SunForum.

In general, application sharing systems adopt either a centralized architecture or a replicated architecture, as shown in Figure 3. In a centralized system, there is only one shared single-user application running on a central server site. The display, or graphical output, is broadcasted to collaborative client sites, and a floor control mechanism[23] is provided for users to take turns to interact with the shared single-user applications which cannot handle simultaneous input. However, floor control becomes a sequential bottleneck of collaborative works. In a replicated system, the application and its environment are replicated at each client site, and executed locally. Only input events are propagated to other sites. Broadcasting graphical events requires more network bandwidth than propagating input events[15, 27]. Since local inputs can be executed locally without network transportation, a replicated system could also achieve higher local response, than a centralized system. So a replicated system has more potential in supporting concurrent cooperative work, especially over the Internet.

The methodologies to build application sharing system are divided into three categories, generic application sharing environment, component-replacement approach, and transparent adaptation approach[7].

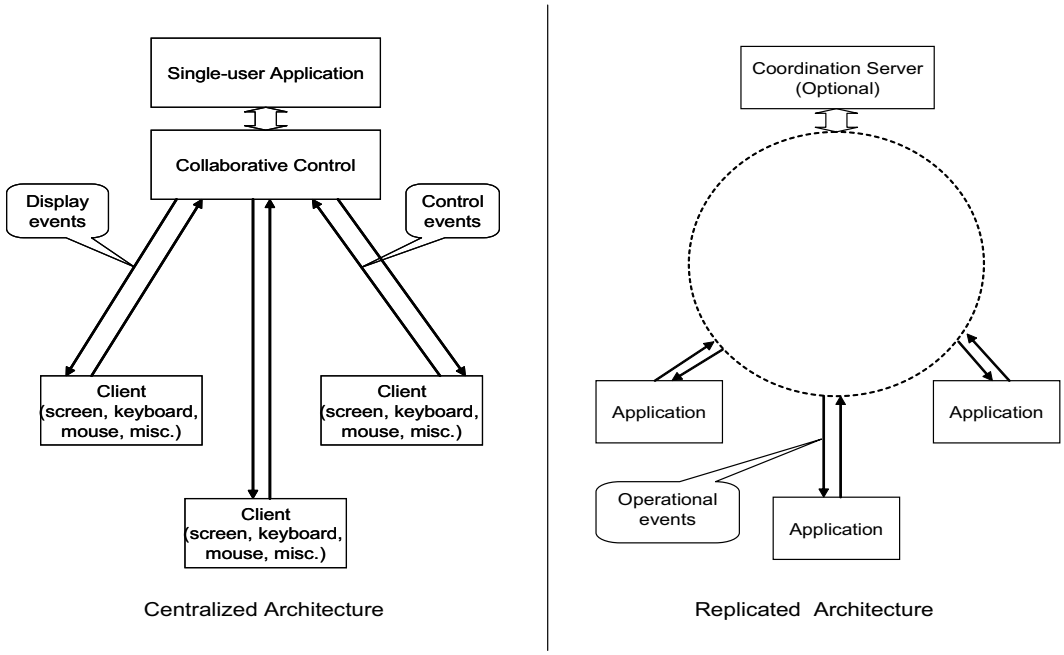

Fig. 3. Centralized & replicated application sharing architecture

1. Generic application sharing environment

Generic application sharing systems use a centralized architecture where only one copy of the application is running. The representatives are XTV[26], Microsoft Netmeeting and SunForum. Users collaborate in a manner of strict WYSIWIS(What You See Is What I See[28]), where users see the exactly same view of the shared application. Floor control mechanism is provided to enforce users to take turns to use the application. Only one user who has the floor can control the application at any instant of time.

The intrinsic attributes, as strict WYSIWIS, sequential problem, and slow local response of centralized systems are too restrictive to some collaborative tasks. Many researchers have criticized that those attributes are the main disadvantage of this kind of system[10, 15].

2. Component-replacement approach

Flexible JAMM[15] replaces single-user versions of Java graphic components with the multi-user versions to share single-user applications. Normally, graphic Java applications are built upon Java SWing and AWT. They are responsible for screen drawing, user input reception, and most of the user interface controls. Flexible JAMM replaces those components in Java Runtime with components integrated with collaborative features. In this approach, selected Java applications to be shared could be automatically adapted into multi-user environment.

Flexible JAMM is based on a replicated architecture, which supports relaxed WYSIWIS, concurrent work, and fast local response. Awareness mechanisms which are only found in traditional specialized groupware applications, such as multi-user scrollbars and radar views, are introduced into application sharing systems for the first time. Late-joining or accommodation of late comers joining to a session which has already started, is supported by direct state transfer, given the whole application state can be serialized and recovered at new sites. Replaceable components including system resources such as files, sockets, and random number generators are provided to support sharing system resources and network connectivity.

However, Flexible JAMM only supports a class of Java based applications. This requirement for single-user applications to be adapted into this system is too constraining. Most off-the-shelf single-user applications cannot meet this requirement.

3. Transparent adaptation

The Transparent Adaption(TA) approach attempts to explore application semantics at some level[7, 10]. It uses applications' and operating systems' API(Application Programming Interface) to intercept user interactions. These interactions will be

translated into abstract operations so that they can be handled by collaboration mechanisms such as OT. It shares single-user applications in a transparent way, i.e., without modifying the source code.

Since the system understands application semantics at some level, it could provide additional session management, awareness control, and other collaborative features in a flexible way. It is able to easily combine with a replicated architecture to achieve relaxed WYSIWIS, concurrent work, and fast local response[10].

CoWord, ICT and ICT2 fall into this category, which use TA approach to share existing applications.

4. Summary

Application sharing systems allow sharing single-user application in a transparent way, without modification of the source code. They support synchronous collaboration. Replicated application sharing systems support relaxed WYSIWIS, concurrent work, and fast local response. Transparent Adaption uses application semantic knowledge and optimistic concurrency control to achieve coordination and consistency, which is very suitable to build synchronous(realtime) group editors.

C. The $O(ND)$ Diff Algorithm

Dynamic programming is one of the earliest algorithms to solve the longest common subsequence problem. It takes $O(N^2)$ time and space to find the optimal solution. The algorithm by Euene W. Myers[13] takes only $O(ND)$ time and $O(N)$ space to get the optimal solution, where N is the sum of the length of two sequences and D is the edit distance between these two sequences. Generally, D is relatively small when the two given sequences are similar. Then the algorithm shows $O(N)$ time complexity.

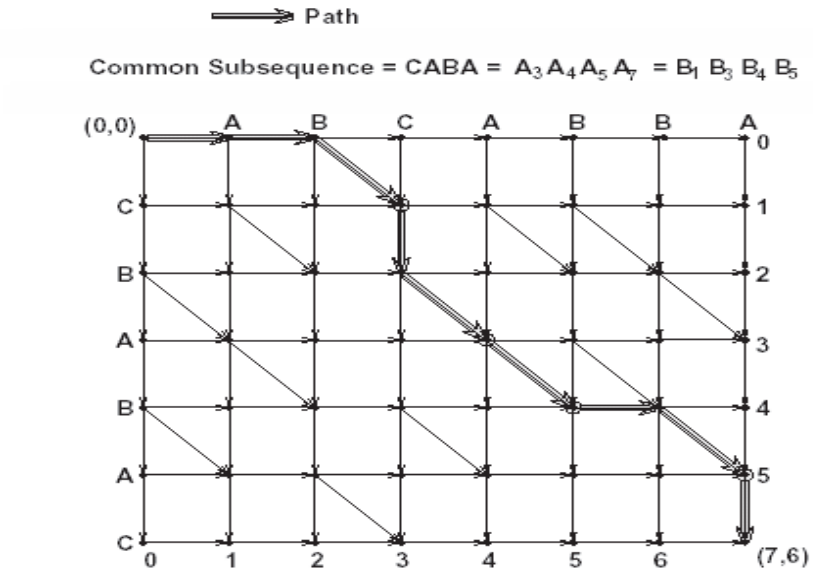

Fig. 4. An edit graph

The idea of this algorithm is to use an edit graph to represent the common subsequence of two sequences. Figure 4 is an example which is adapted from [13]. Each diagonal corresponds to the same characters from those two given sequences at corresponding positions. The horizontal and vertical paths are in correspondence with the deletion or insertion edit scripts. Then the problem of finding a longest common subsequence is equivalent to the problem to find the path from $(0, 0)$ to (M, N) with minimal number of non-diagonal edges.

"Let a D-path be a path starting at $(0, 0)$ that has exactly D non-diagonal edges. Number the diagonals in the grid of edit graph vertices so that diagonal k consists of the points (x, y) for which $x - y = k$."[13] Then the path from $(0, 0)$ to (M, N) is also a D-path and the value of D is between 0 and $M + N$. If two sequences are identical, then D could be 0. Otherwise, we can always find a path with M horizontal paths, N vertical paths and 0 diagonals from $(0, 0)$ to (M, N).

In addition, we could find out that a $D - path$ always ends on the diagonal

within range $-D, -D+2, \ldots, D-2, D$. And we can always get to know the furthest reach of D-path by extending $D-1$-path by greedy approach in constant time[13]. Algorithm 1 shows the basic idea.

Algorithm 1 Basic idea of the algorithm

```
1: for D ← 0 to M + N do
2:     for k ← −D to D in steps of 2 do
3:         Find the endpoint of the furthest reaching D-path in diagonal k.
4:         if (N, M) is the endpoint then
5:             The D-path is an optimal solution.
6:             Stop
7:         end if
8:     end for
9: end for
```

The algorithm takes at most $O((M+N)D)$ time. The two FOR loops are only repeated at most $(D+1)(D+2)/2$ times. $STEP$ 3 takes at most $O((M+N)D)$ time to traverse the diagonals in the graph. Then the total time of the algorithm is only $O((M+N)D)$.

For example, if two given sequences are identical, the FOR loops will be only executed only once. We could extend the 0-path furthest to (M, M) by M steps. The worst case is that the two given sequences are totally different. Then the algorithm has to take $O((M+N)^2)$ time.

In ICT2, the diff algorithm is adapted to derive edit scripts or operations between the old synchronized state and the new concurrent document state. I will explore more issues on this algorithm in Chapter IV.

D. Operational Transformation

Operational transformation(OT) is a technique for concurrency control, it is widely used in group editors[20]. It was originally developed in specialized group editors such as Grove[21] and Reduce[22] for unconstrained cooperative editing of shared documents. To achieve high responsiveness in the Internet environment, group editors use replicated architecture, in which shared document is replicated at the local storage of each participating site. Updates are performed at local sites first, then propagated to remote sites. Local operations are always executed immediately, while remote operations that are concurrent to locally executed operations are transformed before execution. With OT algorithm, users can edit any part of the same document replica at the same time. During editing, users are not forced to take turns, or constrained to a particular part of the document.

As a convention in group editors, the shared (textual) document is modeled as a linear string. Significant editing operations include $insert(S, P)$ and $delete(S, P)$, which insert and delete a string S at position P, respectively. $P(O)$ denotes the position in operation O.

OT is a complex method. Its basic idea can be described as the following simple text editing scenario shown in Figure 5:

Two users are working on a shared document "*abc*" which is replicated at two sites. They generate two concurrent operations $O_1 = insert(\text{“}x\text{”}, 1)$ and $O_2 = delete(\text{“}c\text{”}, 2)$ at each site respectively. On site 1, suppose the operations are executed by the order of O_1 and O_2. After O_1 is executed, the document at site 1 is changed "*axbc*" since O_1 insert character "*x*" at position 1 which is between character "*a*" and "*b*". The following execution of O_2 will incorrectly delete character "*b*" instead of "*c*" since "*b*" is at position 2 now. Hence, to delete the correct character

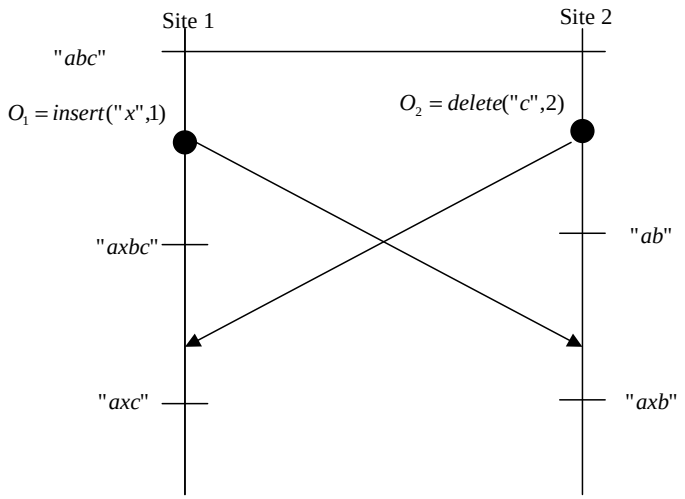

Fig. 5. Operational Transformation(OT)

"c", O_2 must be transformed to $O_2' = delete(\text{“}c\text{”}, 3)$ before execution. $P(O_2')$ equals to $P(O_2) + 1$ because of the insertion of one character "x" by O_1.

The basic idea of OT algorithm is to transform(adjust) the positions of editing operations according to the previous executed concurrent operations in order to achieve the correct effect and a consistent document state[21]. In this thesis, ICT2 uses the idea of OT algorithm to transform concurrent edit scripts to maintain the consistency of the replicated document at each site.

CHAPTER III

ICT2 AGENT

The agent system is the primary component of ICT2. As shown in Figure 6, an agent is composed of five modules: controller, diff & merge, awareness control, user interface, and communication.

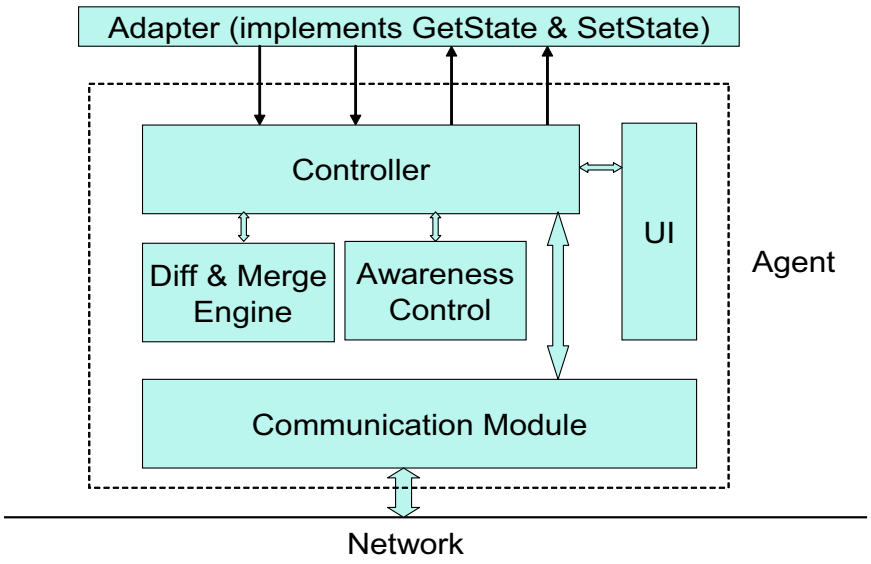

Fig. 6. The adapter and the agent together facilitate collaboration

The controller decides when to get the current state of the editor and when to reset the editor to a new synchronized state. It communicates with the user through the user interface module and communicates with agents at other sites through the communication module. It calls the diff & merge module to derive the local state changes and compute the new synchronized state. It also monitors keyboard and mouse events of the editor to detect local user activities.

The diff & merge engine implements three functions: First, it implements a simple diff algorithm tweaked from Myers's diff algorithm [13] for deriving a shortest

edit script between two documents. Second, it decides how to merge two given edit scripts. Third, it applies a given edit script to a given document state.

The awareness control module collects the local user's information such as where is the users current caret position and which regions of the document were changed. It also collects awareness information about other users such as who are present, who did what and where[29]. In ICT2 system, much awareness information is eventually derived by calling the diff algorithm.

The user interface module implements three types of interfaces: It allows the user to control which document to edit, who are the coauthors, and when to synchronize. It provides interfaces for the user to configure the system, e.g., how often diffing is invoked to compute awareness information, and whether synchronization is automatic or manual. It also presents awareness information to the user.

The communication module provides network communication functions for other modules. The communication between clients is peer to peer connection. Each client connects to other clients directly, but not through session manager server. This kind of design can improve the network communication performance since there is no central server to act as a bottleneck.

CHAPTER IV

DIFFING AND MERGING

Previous approaches such as ICT[10] and CoWord[7] adapt single-user editors by translating their window events into editing operations. The thrust of this approach is to make the adapting of single-user editors more robust with reduced engineering costs. The core idea is to derive the editing operations by diffing between document states instead of translating them from keyboard and mouse events. Then concurrent edit scripts are merged and applied to get a new synchronized state. I will explain how this idea works in the following sections.

A. Deriving Edit Script

The well-known diff algorithm of Myers [13] is tweaked for deriving a shortest edit script between two document states. Suppose S_0 is an earlier state and S_1 the current state. Analogous to established conventions in group editors [20], document states are represented as linear strings and the edit script for transforming S_0 to S_1 is a sequence of insertions and deletions.

The time complexity of the diffing algorithm is $O(ND)$, where N is the sum of the lengths of S_0 and S_1, and D is the size of the minimum edit script for S_0 and S_1. In general the algorithm performs well in typical applications where the edit distance (as characterized by parameter D) are small. Its worst-case complexity is $O(N^2)$, which only happens when S_0 and S_1 are totally different and thus D is equal to $2N$.

We implemented ICT2 and the diffing algorithm in Microsoft .NET. The following experiments were run on an Intel Pentium-4 1.7 GHz PC with 512M RAM.

The original algorithm assumes general sequences and computes the edit script at the character level. For the purposes of supporting human-oriented document writing,

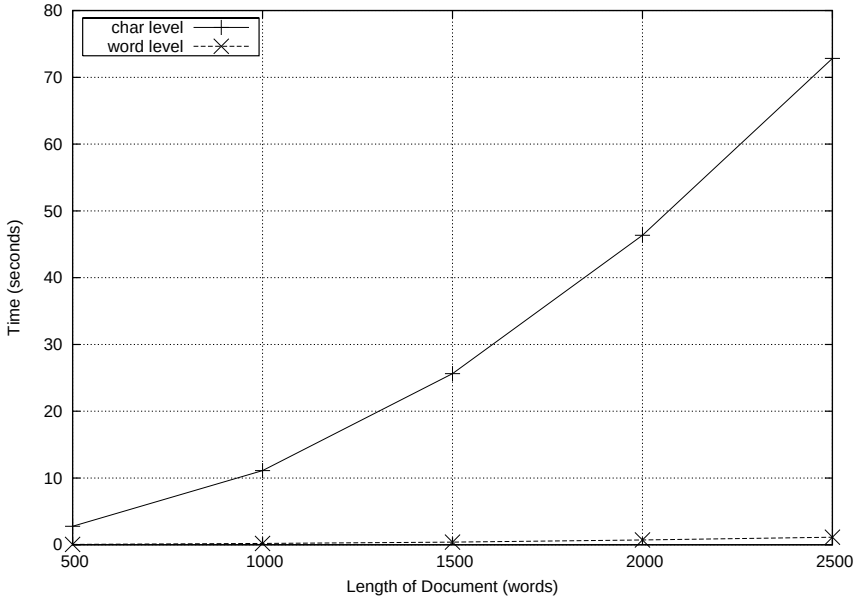

Fig. 7. Performance of character- and word-level diffing in the worst case ($D = 2N$)

many heuristics can be explored for even better performance. For example, if we derive the edit script at the word level, there is a significant performance improvement. Suppose the average length of words is 6 characters. As shown in Figure 7, in the worst case, it takes about 1 second to compute a word-level edit script between two 2500-word documents, while it takes about 73 seconds to compute a character-level edit script between the same documents.

As shown in Figure 8, the word-level diffing time is about 5 seconds when the document length is 5,000 words, and less than 19 seconds when it grows to 10,000 words.

To understand how sensitive the diffing time is to the amount of differences, we ran a third experiment on a 10-page CSCW'04 paper [30] with over 16,000 words. As shown in Figure 9, it takes about 0.5 second to locate 10% random changes, which means over 1,600 string-wise insert/delete operations. Note these operations are at different positions for, otherwise, they would have been combined. Suppose a quick

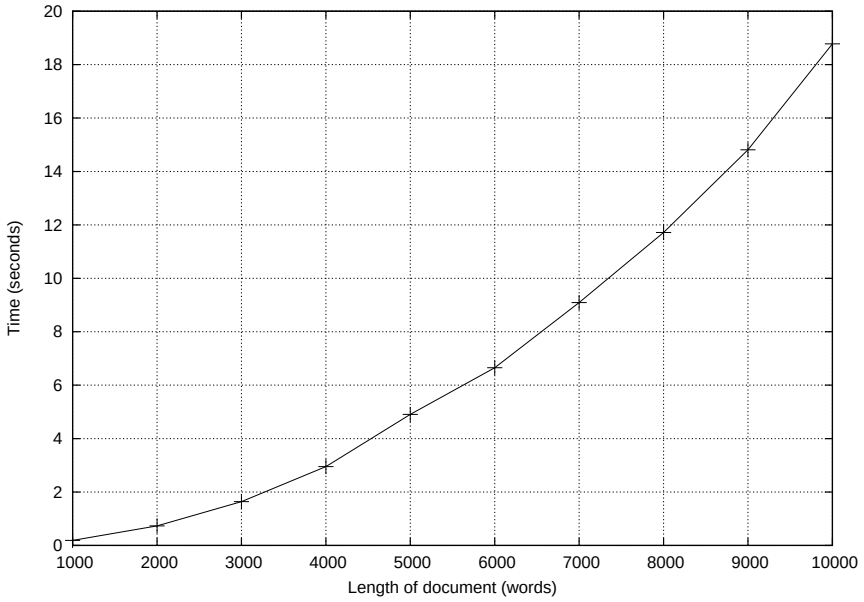

Fig. 8. Worst-case word-level diffing ($D = 2N$)

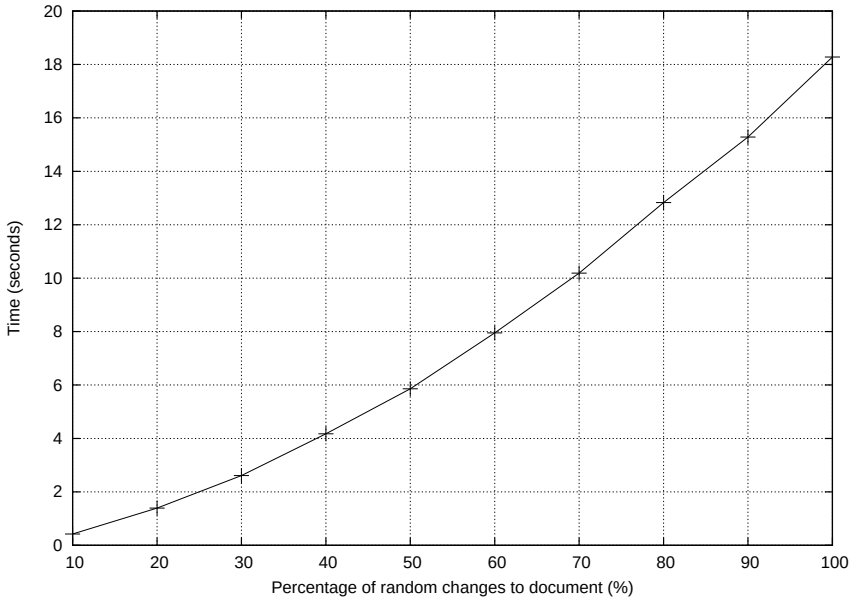

Fig. 9. Diffing on a 10-page paper ($N = 2 * 16752$ words) under various change percentages ($\frac{2D}{N} * 100\%$).

user can make 20 string-wise changes per minute. Then it takes about 1.5-4 hours of work for the user to generate 10-30% changes to a document of 10 pages, which can be located in less than 3 seconds.

In fact, even real time group editing does not mean to synchronize every single operation. As discussed in [31], there are performance problems such as screen flickers when synchronization is too frequent. Hence real time group editors including Reduce and CoWord allow the users to control the timing and granularity of synchronization.

Most main conferences to our knowledge require papers be no more than 8-10 pages. When synchronization happens at a reasonable pace, say every 5 minutes or 100 operations, the diffing time will be bound within 80-100 milliseconds requirement of interactive applications [32].

We hypothesize that in most group writing practices users will neither want to synchronize their document replicas on every editing operation nor want to wait until the documents are *completely* different. Sync at every 10-30% changes would be more reasonable and typical. Therefore diffing can be tweaked to provide near realtime performance for practical group writing tasks.

B. Merge Edit Scripts

Edit scripts output from the diff algorithm are composed by operations in the forms of $INS(S, P)$ and $DEL(P_1, P_2)$, the former to insert string S at position P and the latter to delete characters inclusively from position P_1 to P_2. All the position parameters are relative to the original document. For instance, let the last sync state be a string "*abcde*" and the current state be "*abxc*". Assume the first character has the position value 0. The edit script will be "INS("x",2) and $DEL(3, 4)$.

Let function $P(O)$ denote the effect position of operation O. We define that

$P(INS(S,P)) = P$ and $P(DEL(P_1,P_2)) = P_1$. Operations in an edit script are ordered ascedingly by their effect positions relative to the last sync state. If a INS and a DEL have the same effect position, the INS is sorted before the DEL.

The process to merge two edit scripts is similar to the classic merge-sort algorithm which merge two sorted arrays. It is approximately described by Algorithm 2.

The output sequence Q is initially empty. The algorithm scans the two input sequences Q_1 and Q_2 from left to right and add one operation into Q at a time. Note the position parameters of all operations in Q_1,Q_2, and Q are defined relative to the last sync state and are ordered ascendingly. The merging time is linear to the total length of Q_1 and Q_2. That is, its time complexity is $O(D)$, which is dominated by the diffing time complexity $O(ND)$.

The positions of two operations from two edit scripts may overlap due to concurrency. For instance, two users may delete the same character at the same position. Hence two edit scripts may both contain operation say $DEL(3,3)$. Then the effect positions of O_1 and O_2 tie. In this case, we will only keep one $DEL(3,3)$ in the final result. We use the following rules to deal with the overlapping situation.

1. If two DEL operations overlap, keep the union of these operations in Q. For example, $DEL(5,9)$ and $DEL(6,10)$ are merged as $DEL(5,10)$.

2. If two INS operations overlap,(i.e., with the same effect position), add them into Q in the order of their site IDs.

3. If a INS operation overlap with a DEL operation, split the DEL and add the three resulted operations into Q in order. For instance, $DEL(4,10)$ and $INS(S,5)$ are merged as $DEL(4,5)$, $INS(S,5)$ and $DEL(6,10)$.

C. Applying Edit Script

As shown in Figure 1, after we merge two concurrent edit scripts into a new script Q, we apply Q to the last sync state S_0. This is accomplished as Algorithm 3.

In this algorithm, i indicates the current processing position in S_0. The text before i has been processed. Function $S(O)$ denotes the string parameter of operation O, $S[x, y]$ denotes the substring of S from position x to y, $P(O)$ denotes the position of O, and $P_{end}(O)$ denotes the end position of O which is only applicable to DEL.

The algorithm checks through every operation O in the input edit script Q. If O inserts at current position, we append the inserted text to S_1. If the operation is DEL, we simply skip the text in S_0 by increasing pointer i. If O inserts at a later position, meaning that the text between i and $P(O)$ is unchanged, we simply append it to the new state S_1. Similarly, if Q is empty, we append the rest of S_0 to S_1. Apparently, the time complexity of Algorithm 3 is $O(|Q|)$.

It is possible that several INS oeprations in Q have the same position. They are sorted in Q in the order of their site IDs by Algorithm 2. Hence we just need to apply them by their order in Q, as above.

Algorithm 2 Merge two edit scripts

```
1: procedure MergeScripts(Q1,Q2) :Q
2:     while not end of Q1 and Q2 do
3:         get current operation O1 from Q1
4:         get current operation O2 from Q2
5:         compare O1 and O2
6:         if P(O1) < P(O2) then
7:             append O1 to Q, and move Q1 pointer
8:         else if P(O2) < P(O1) then
9:             append O2 to Q, and move Q2 pointer
10:        else if overlapping then
11:            //omitted
12:        end if
13:    end while
14:    if not end of either Q1 or Q2 then
15:        append all its remaining operations into Q
16:    end if
17:    return Q
18: end procedure
```

Algorithm 3 Apply edit script to last sync state

```
1: procedure ApplyScript(Q,S_0) :S_1
2:     i ← 0
3:     S_1 ← ∅
4:     while Q is not empty do
5:         remove the first operation O from Q
6:         if O is INS then
7:             if P(O) = i then
8:                 append S(O) to S_1
9:             else if P(O) > i then
10:                append S_0[i, P(O_1) − 1] to S_1
11:            end if
12:        end if
13:        if O is DEL then
14:            i ← P_end(O) + 1
15:        end if
16:    end while
17:    append S_0[i, |S_0| − 1] to S_1
18:    return S_1
19: end procedure
```

CHAPTER V

AWARENESS AND COORDINATION

Mutual awareness of each others status and progress is a basis for collaborators to coordinate their activities[33]. ICT2 provides two useful awareness mechanisms. One is the synchronization report which, at synchronization time, shows how the remote site changed the document. The other is the progress report which, before synchronization time, indicates the approximate editing activities (progress) at the remote site. In our system, these awareness mechanisms are deliberately not coupled into the single-user editor interfaces, an "unobtrusive yet accessible" design[19].

A. Synchronization Report

At synchronization time, remote changes are merged with local changes and the resulted edit script is applied to the last sync state. At the system level, it is straightforward to just set the editors to the new sync state at each site. However, we feel it sometimes necessary for the users to make sense what have been changed (inserted and deleted) by the other sites since last sync. This is achieved by a sync report dialog. Additionally we implemented user interfaces for the users to configure whether or not they want the sync report displayed automatically at sync time. They may also click a button to browse the sync report when necessary.

Internally the sync report is represented in the standard Rich Text Format (RTF). RTF is a file format by Microsoft for cross-platform document interchange which most text processing programs are able to process. We use different colors to show operations performed by different users. New inserted texts are underlined and deleted texts are stroke out. As shown in Figure 10, the sync report at each site only shows the changes made by the other user(s).

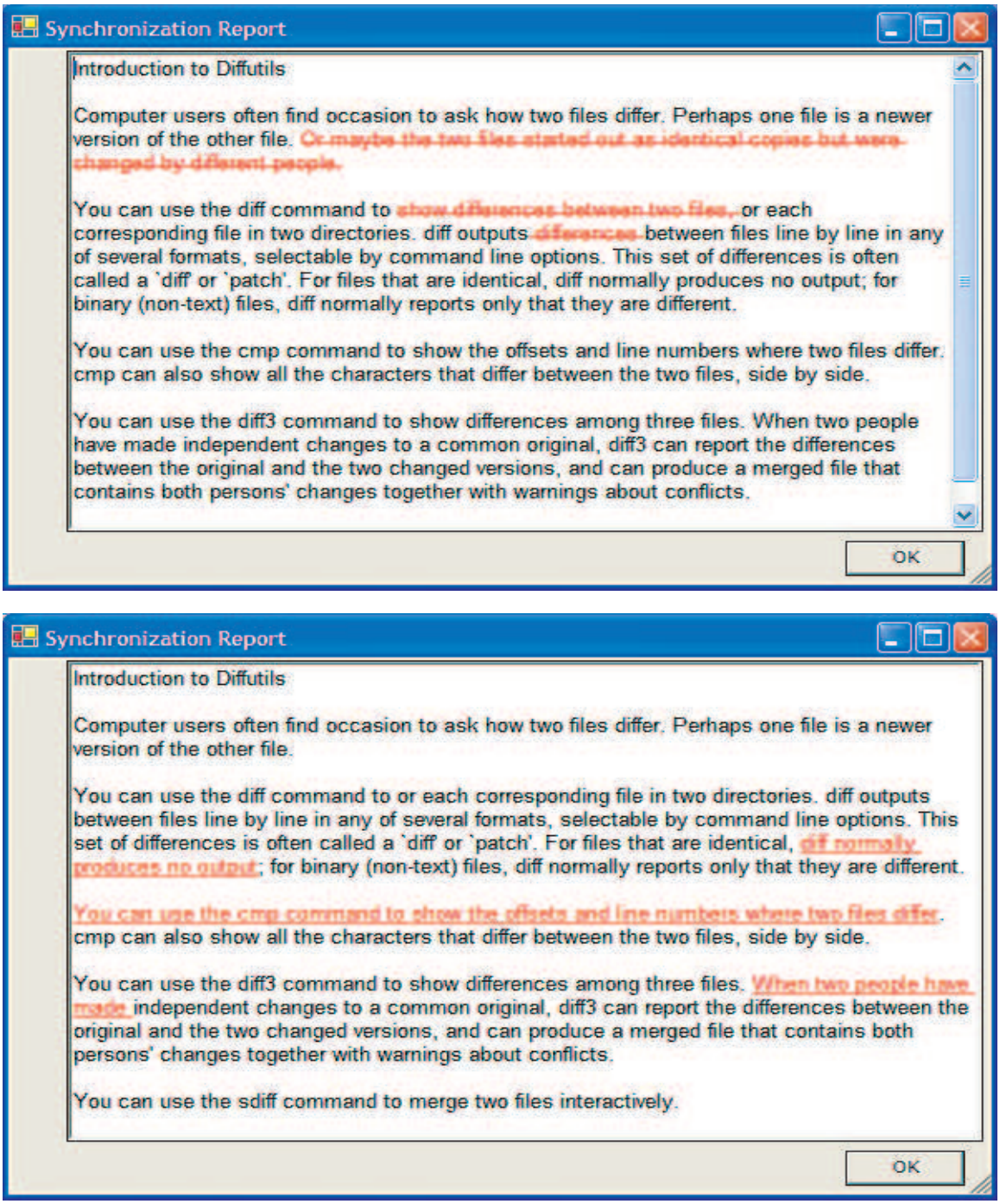

Fig. 10. Different sync reports at different sites

For instance, suppose the last synchronized content is "*abcdef*". User A changes the document to "*abxycd*" by deleting "*ef*" and user B concurrently changes the document to "*abef*" by inserting "*xy*" and deleting "*ef*". While user B concurrently changes the document to "*abef*" by deleting "*cd*". The new synchronized content is "*abxy*". The sync report to user A will be like "*abxy~~cd~~*", while the sync report to user B will be like "*ab<u>xy</u>~~ef~~*".

To implement this reporting feature, Algorithm 2 and 3 are slightly extended to include and process some additional information in the merged edit script such that operations from different sites are distinguished. For example, suppose that we have $DEL(5, 9)$ from site 1 and $DEL(6, 10)$ from site 2. Then in the merged edit script, there are three annotated operations: $DEL_1(5, 5)$, $DEL_{1,2}(6, 9)$ and $DEL_2(10, 10)$ to indicate that site 1 deleted the character at position 5, site 2 deleted the character at position 10, and both sites deleted the text ranging from position 6 to 9. When applying this script, all the three operations are executed. However, in the sync reports, only the effect of $DEL_2(10, 10)$ is displayed at site 1, and only the effect of $DEL_1(5, 5)$ is shown at site 2.

The sync report dialog is implemented in a custom interface. It is possible to integrate the dialog into the editor interfaces if the editors in question provide appropriate APIs. Anyway the existence of the dialog does not impair the principles of collaboration transparency, i.e., adding groupware features to legacy applications without modifying source code[15].

B. Progress Report

As shown in Figure 11, to help the users make more informed sync decisions, we implemented a simple progress report dialog that displays the following awareness

information: how much editing have the local and remote users made since last sync, where relative to the last sync state are those changes made, when did last sync happen, where was the remote user's position, how recent is the local knowledge of the remote site? This information is updated at user-configured intervals. Similar to the sync report, the progress report is external to the single-user editors being used.

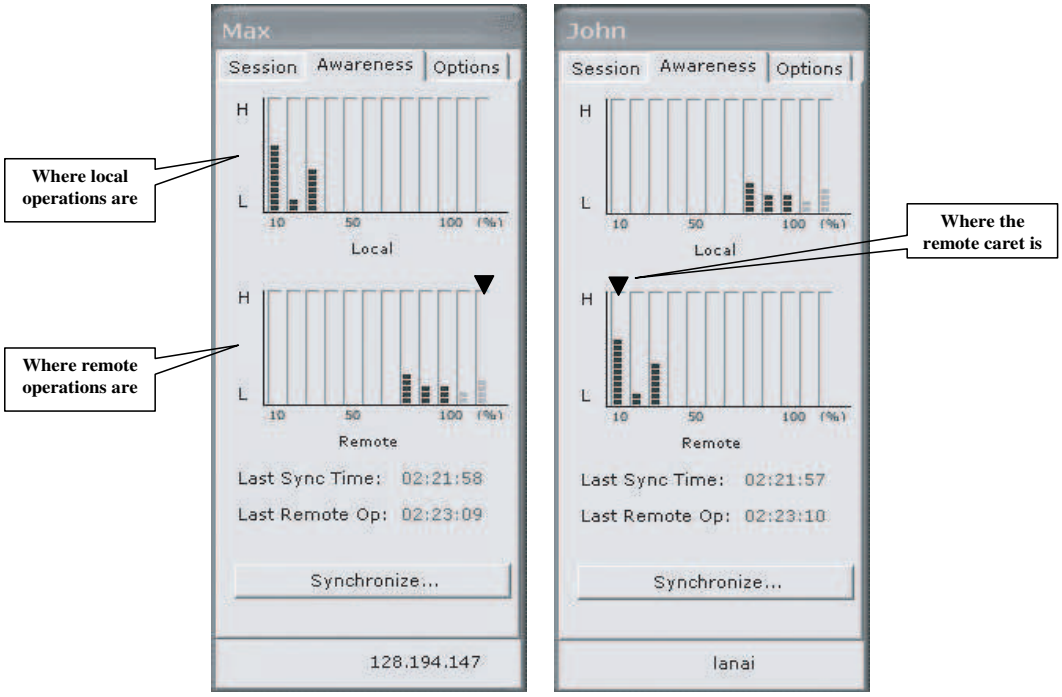

Fig. 11. Providing approximate awareness information of local and remote progress

At each site, we use one chart to visualize the local progress and one for remote progress. In each chart, the first ten columns each represent a ten percent area of the shared document, from 0%-10% to 90%-100 relative to the last sync state. The last two columns, in a different color, represent operations performed in content that was newly appended to the last sync state. In each column, the higher the block stack, the larger volume of operations have been performed in the corresponding area of

the document. The triangle above the remote chart indicates the remote user's caret position to the system's most recent knowledge.

Figure 11 shows the progress reports of two collaborating users. One is logged in as Max and the other John. It is clear from the figure that Max only changed the first 30% of the document, while John changed the last 30% of the document and also added some new content.

The progress information of each user is obtained by diffing between the current state of the editor and the last sync state. On one hand, up-to-the-moment information is necessary for the users to make quality sync decisions and coordinate effectively. On the other hand, since this awareness information is acquired while the user is possibly working, frequent diffing may distract the user. A balance must be sought between awareness, which affects group productivity, and system performance, which affects individual productivity. Similar design considerations are confirmed in [34].

We address this tradeoff at the following two levels. At the first level, invocation of the diff algorithm to derive awareness information can be configured to occur at the paragraph (line, sentence, word) level. The diffing experiments as described in last section showed that diffing at the word level is significantly faster than diffing at the character level. This trend should continue as diffing occurs at coarser granules. Due to labor division that is common in most group writing practices[1, 2], people rarely need to edit the same paragraph at the same time. Awareness at the paragraph level, although somewhat coarse-grained, should be able to serve the need of most practical situations and pay off in terms of much improved system performance.

At the next level, we allow the users to negotiate and configure how often the awareness information is updated. Specifically, we define two system parameters for controlling the triggering of updates. One is the sampling timer. The user can set the

time interval between two consecutive updates, say every 30 minutes. This ensures that awareness information is provided periodically. The other is driven by window events. The agent keeps simple statistics of the amount of keyboard and mouse events the user generated on the editor window since last sync, which indicates how active the user has been changing the document. The user can configure the system such that diffing is called say every 500 events. This ensures a timely report of progress if a user makes a lot of changes in a short time before the current interval ends. The periodic sampling may still serve a good purpose of providing presence awareness even if a user is not active for some time.

CHAPTER VI

ADAPTING SINGLE-USER EDITORS

Now we have discussed how to synchronize the states of single-user editors and how to coordinate synchronization. In this chapter, we shall address the fundamental problem of how to transparently adapt single-user editors to provide the two required interfaces: GetState and SetState.

In a group editing environment with heterogeneous editors, we are not interested in how users press keys and buttons to generate editing operations, as in a homogeneous system like CoWord[7]. Hence an interesting state of an editor in our system only includes its content (string) and the current caret position. Many popular single-user editors provide APIs for third parties to implement add-on features, such as GVim, Emacs and MSWord. For example, the CoWord project builds on the COM-based APIs provided in Word to implement group editing[7]. Obviously, for editors in which APIs are available, it will be trivial to implement GetState/SetState and performance will not be a problem.

In the following I will explore how to adapt editors without APIs, such as WinEdt, to support the GetState/SetState interfaces and how to solve the performance problems that may ensue. The same techniques equally apply to editors with APIs and can be implemented on these editors where generality is the goal to pursue.

A. Adapting Editors without APIs

The adaption is accomplished through simulating select/copy/paste events on the editor and access the clipboard. Mechanisms for achieving these are generally provided in modern window-based platforms such as Microsoft Windows and X Window. This approach has been tested on popular single-user editors including GVim, MSWord,

WinEdt, Notepad, and EditPlus.

For example, on MS Windows platforms, GetClipboard and SetClipboard are APIs for reading and writing the clipboard, respectively. Most single-user editors support keyboard shortcuts: $CTRL{+}A$ for selecting all content of the editor, $CTRL{+}C$ for copying selected text to the clipboard, $CTRL + V$ for pasting the clipboard content to current caret position, $CTRL{+}SHIFT{+}HOME$ for selecting the editor content from the beginning to current caret position, $CTRL + SHIFT + END$ for selecting the editor content from current caret position to the end, $CTRL + HOME$ for setting the caret position to the beginning, and so forth.

Hence the basic idea is to simulate keyboard events on an editor such that the agent can get and set the editor state via the clipboard. For example, suppose we want to set the content of WinEdt to "*abcd*". We can first set the clipboard to "*abcd*", then simulate $CTRL{+}A$ on WinEdt to select all its content, and then simulate $CTRL{+}V$ on WinEdt to replace its content to "*abcd*".

Algorithm 4 Get state by simulating window events

1: **procedure** GETSTATE :S,P
2: simulate $CTRL + SHIFT + HOME$
3: simulate $CTRL + C$
4: $S_1 \leftarrow GetClipboard()$
5: simulate $CTRL + SHIFT + END$
6: simulate $CTRL + C$
7: $S_2 \leftarrow GetClipboard()$
8: $P \leftarrow Length(S_1)$
9: $S \leftarrow S_1 + S_2$
10: **return** (S,P)
11: **end procedure**

Algorithm 5 Set State by simulating window events

```
1: procedure SetState(S,P)
2:     S_1 ← S[0, P − 1]
3:     S_2 ← S[P, |S| − 1]
4:     simulate CTRL + A
5:     SetClipboard(S_2)
6:     simulate CTRL + V
7:     simulate CTRL + HOME
8:     SetClipboard(S_1)
9:     simulate CTRL + V
10: end procedure
```

As shown in Algorithm 4 and 5 , the GetState and SetState interfaces are implemented by simulating window events. To get the state (S, P) of an editor, we first copy its content up to the caret position into string S_1, and then copy its content after the caret position into string S_2. Thus the content of the editor is $S_1 + S_2$, and the caret position is the length of string S_1. To set an editor's content to a given string S and its caret position to a given P, we first replace its content by the substring after the caret position, $S_2 = S[P, |S| - 1]$, and then insert the substring before the caret position, $S_1 = S[0, P - 1]$, to the beginning of S_2. As a result, the editor content is set to $S = S_1 + S_2$ and the caret is set between S_1 and S_2.

B. Restoring Local Caret Position

The caret position returned from GetState is used for awareness. As shown in Figure 11, we can indicate to the local user where the remote caret position was last time GetState was called. However, when synchronization occurs, the local caret position

could be lost after calling MergeScripts and ApplyScript. Hence we must be able to compute the local caret position before SetState is called.

Caret is a mark used by an editor to indicate where something is to be inserted into its text content. After sync, the local caret position may be dislocated due to the merging of remote operations. For instance, as in Figure 1, suppose last sync state S_0 is "*abcd*". The two concurrent states are S_1 = "*abcxyd*" and S_2 = "*azbcd*". Suppose the caret position in state S_1 is $C_1 = 5$, which is between '*y*' and '*d*'. The three edit scripts are $D_1 = \{INS(\text{“}xy\text{”}, 3)\}, D_2 = \{INS(\text{“}z\text{”}, 1)\}$ and $D_3 = \{INS(\text{“}z\text{”}, 1), INS(\text{“}xy\text{”}, 3)\}$. After sync, the new state is S_3 = "*azbcxyd*". The caret position in S_3 should also be between '*y*' and '*d*', which is $C_3 = 6$.

To compute the new caret position, first, we compute the local caret position C_0 relative to state S_0. In the above example, since $C_1 = 5$, we know $C_0 = 3$. However, if C_1 is 3 or 4, we will also get the same result of $C_0 = 3$. This is because of a new string "*xy*" that was not present in S_0. Hence we introduce a parameter δ to distinguish these positions relative to the new string "*xy*". When C_1 is 3, 4 or 5, the value of $C_0 = 3$ and the value of δ should be 0, 1 or 2, respectively.

Second, when calling algorithm ApplyScript, we compute the local caret position C_3 relative to state S_3, from parameters C_0, δ and D_3. This is rather straightforward: Initially we set C_3 to C_0 and use operations in D_3 to adjust C_3. Because all operations in D_3 are sorted ascendingly by their positions, we can easily trace which operations increment or decrement C_3. The final value of C_3 simply adds δ. In the above example, we first get $C_3 = 4$ due to operation $INS(\text{“}z\text{”}, 1)$. Then it is adjusted to 4, 5, or 6 by adding the corresponding value of δ (0, 1, or 2) into C_3.

The unit of operation may be different from the unit of character position. For example, we do diff algorithm on word level, and the position parameter of operation is also based on word. But the caret position is based on character. Others may

Algorithm 6 Compute the old position and δ

```
1: procedure ComputeOldPos(S_0,D_1,C_1) :pos,δ
2:     C_0 ← 0
3:     pos ← 0, δ ← 0
4:     while C_0 < C_1 do
5:         apply D_1 to S_0 like Algorithm 3
6:         adjust C_0 and pos accordingly
7:     end while
8:     δ ← Length(LastOperation) − (C_0 − C_1)
9:     return (pos,δ)
10: end procedure
```

do diff algorithm on other levels, e.g., sentence level, to achieve higher performance. Parameter δ can also distinguish the positions relative to word, sentence or other higher levels. And, we use another *pos* parameter to represent the local caret position relative to state S_0 based on operation level.

Algorithm 6 shows the general method to compute the old position *pos* of S_0 and δ based on the above idea:

pos is the old caret position based on operation level. C_0 and C_1 are both based on character level. For the second step, we use the same idea, but with *pos* instead of C_0, to compute new position pos' which is also based on operation level. Then we can calculate C_3 by pos' and δ.

C. Some Performance Issues

As shown in Figure 1, for simplicity, we only consider two sites in our concept-proving prototype. Due to the way states are synchronized, we can safely consider that all

operations in D_1 are concurrent with all operations in D_2. Hence we do not need consistency control algorithms as sophisticated as those in [20, 30]. This approach slightly constrains synchronization to achieve simplicity and efficiency in consistency control. A similar tradeoff has been confirmed in the previous work[35]. As a result, the above described algorithms for merging edit scripts and computing the new caret position take only linear time.

However, there are performance issues in the adaption approach that must be addressed. The first is related to selection. If the length of selection is 0, Windows Clipboard will keep its current content when we simulate $CTRL + C$ to copy the selection. For example, suppose the content of an editor is "*abcd*" and the caret position is 4 which is after character '*d*'. In procedure GetState of Algorithm 4, S_1 will be "*abcd*" after step 4. However, step 5 will select nothing and hence step 6 will not change Windows Clipboard. Then after step 7, S_2 is also "*abcd*". To solve this problem, before each selection, we reset the Clipboard to a special string. Then we do the copy simulation. If the content of Clipboard turns out different, the new content is the selection. Otherwise, the selection and the Clipboard content are empty.

The second problem is user input interference. While the agent is performing synchronization, the user may not notice this and continue with editing. The user input may change the selection or change the focus, which interferes the routines of Get/SetState and may cause the wrong result. To solve this problem, we use Windows hook techniques to block user input while performing Get/SetState. A hook is a mechanism by which a piece of user-defined code is planted into the target application. It can be used to intercept the system events before they reach the application. With this technique, we catch keyboard and mouse events before they reach the editor while Get/SetState is underway. Only events simulated by Get/SetState are allowed to go through.

The third problem is screen flickers that may be caused by the simulated select/copy/paste events in Get/SetState calls. To reduce the interferences to the user, we use the mask window mechanism.The idea is that before entering GetState or SetState, we create an opaque mask window on top of the editor, whose size is exactly the same as the size of the editor. The mask window displays the current image of the editor so that the user will not notice the simulated operations. The mask window is hidden after the Get/SetState operation is completed. Then the interaction continues as normal.

Techniques to mitigate the second and third problems may temporarily render the user unable to input for a short period of time say a few hundred milliseconds. Given the good performance of the synchronization algorithms, the duration will be too short to disrupt the editing task. Interferences could happen when Get/SetState is called at sync time and when GetState is called for collecting awareness information. On one hand, the users are more often than not prepared psychologically to tolerate some delays during sync time, especially if it is the users who initiate the sync process. Additional information could be displayed so that the users are made aware of the progress of sync. On the other hand, when GetState is called for awareness purposes, information such as the caret position and the edit script does not need to be as accurate as that used for sync. Hence diffing at the paragraph level often suffices. The execution of GetState can be made even less obtrusive, e.g., by scheduling it in periods when the user is not actively editing.

In practice, although awareness is the basis for coordination, human collaboration is rarely confined to any provided CSCW system. People often have alternative channels, such as online chat and email, to get aware of each other's status and progress and to decide when to synchronize. Although group editors can ease the sync process, they are usually used together with other tools, e.g., chat and audio/ video[6].

These tools can often help the users to communicate awareness information, negotiate writing processes and roles, and resolve conflicts and semantic inconsistencies. Our system allows the users to configure the progress report, which can even be disabled in the case that all the above techniques fail to provide desirable performance.

CHAPTER VII

CONCLUSIONS

This thesis presents a novel approach to adapting familiar single-user editors for group editing. It only needs to adapt the editors to support two simple interfaces, GetState and SetState. This is accomplished either by the editor-provided APIs or by simulating window events. Based on these two interfaces, a classic diff algorithm[13] is tweaked to derive edit scripts between editor states. Concurrent edit scripts are merged to synchronize editor states at cooperating sites. Awareness mechanisms are implemented for the users to make sense of each others progress and make informed sync decisions. Experiments show that our approach is able to provide near-realtime performance for most practical group editing tasks.

A. Main Contribution

The main contributions of this approach include low engineering cost and allowing for heterogeneity.

The presented technology contrasts sharply with recent approaches to sharing single-user editors for realtime group editing, such as ICT[10] and CoWord[7]. We do not need to understand and translate editing operations at the operating system level. Hence the development and perfective maintenance costs are significantly less. Moreover, since our approach relies on the differences between document states instead of the actual editing operations that cause the differences, we do not impose any constraint on specific editors and editing commands the users can use. As a result, editors are allowed to be heterogeneous and there is no need to disable features in the familiar user interfaces. The number of single-user editors grows rapidly. Low engineering cost makes it possible for group editors to catch up with single-user editors.

It makes group editing by sharing existing single-user editors a reality.

This approach allows shared single-user editors to be heterogeneous. That means coauthors could use their familiar single-user editors to edit documents together. Due to the variety of editors, it is not desirable to constrain that all the users be familiar with one editor. Allowing for heterogeneity reduces the learning cost, elevates the utilization of group editing system, and then improves the productivity of group.

In addition, ICT2 uses an agent-based system architecture. It does not force the users to use use an unacquainted environment with group features which is less frequently used. Since the agent is a separated and independent software application to single-user editors, it does not change any single-user editor's behavior or user interface. Users' experience on editors is totally same as before. Users can still focus on document editing with their familiar single-user editors.

B. Limitation

The down side of this approach is that it is only able to implement near-real time group editing due to performance problems of diffing. However, we hypothesize that the tradeoffs are reasonable for group writing: the users may prefer a system with certain constraints on synchronization to one that disables many interface features of their familiar single-user editors or word processors. If near-real time synchronization is sufficient for most situations, it justifies to trade certain level of flexibility for significantly lowered engineering costs and near-unconstained use of familiar editors.

Our prototype currently supports text editors such as GVim and WinEdt. The system (or more specifically, the diff algorithm) does not interpret the content of the document. Word processors can be used in the system as long as their content can be accessed as text. This may not necessarily cause a disaster in group writing. Given

that the content often carries more weight and is more difficult to produce than the formattings, the final content can always be formatted, e.g., in Word or Latex, in a later phase of the group writing project. It has been confirmed in previous studies that many group writing projects are carried out in phases[1, 2].

C. Future Work

The system has been prototyped on Windows XP over the past year. The specific techniques discussed have been proved over the past three years to be generally feasible on many popular text editors and word processors as well as Windows and Linux platforms. As a proof of concept, the prototype system is simplified such that only two users can use GVim and WinEdt to edit the same document at the same time. Some constraints are imposed, e.g., user inputs are disabled during sync time, to avoid interferences. However, these constraints are temporary in our current prototype rather than inherent in the technology itself. They will be relaxed in future work after we extend the synchronization algorithms.

Today a plethora of tree- and XML-based diff algorithms are available in the literature, e.g., [36, 37]. The existence of these algorithms suggests that it is possible to support structured documents and more sophisticated editors in our system. Due to the treatment of structures and extra operations, these algorithms as they are may appear less efficient than the text-based diff algorithm of [13]. However, as revealed in our experience, many domain-specific semantics and techniques can often be exploited to tweak these algorithms for sufficient performance in CSCW. For example, in general minimality of the edit script can be traded off for significantly reduced execution time[36]. We plan to explore this direction in future work.

REFERENCES

[1] I. R. Posner and R. M. Baecker, "How people write together," in *Proceedings of IEEE HICSS'92 Conference*, Koloa, Hawaii, 1992, pp. 127–138.

[2] S. Noel and J.-M. Robert, "Empirical study on collaborative writing: What do co-authors do, use, and like," *Journal of Computer Supported Cooperative Work*, vol. 13, pp. 63–89, 2004.

[3] P. Bellini, P. Nesi, and M. B. Spinu, "Cooperative visual manipulation of music notation," *ACM Transactions on Computer-Human Interaction*, vol. 9, no. 3, pp. 194–237, Sept. 2002.

[4] R. M. Baecker, D. Nastos, I. R. Posner, and K. L. Mawby, "The user-centered iterative design of collaborative writing software," in *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, Amsterdam, The Netherlands, Apr. 1993, pp. 399–405.

[5] P. Dewan, R. Choudhary, and H. Shen, "An editing-based characterization of the design space of collaborative applications," *Journal of Organizational Computing*, vol. 4, no. 3, pp. 219–240, 1994.

[6] C. A. Ellis, S. J. Gibbs, and G. L. Rein, "Groupware: Some issues and experiences," *Communications of the ACM*, vol. 34, no. 1, pp. 38–58, Jan. 1991.

[7] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen, "Leveraging single-user applications for multi-user collaboration: The CoWord approach," in *Proceedings of ACM 2004 Conference on Computer Supported Cooperative Work*, Chicago, Nov. 2004, pp. 162–171.

[8] D. Li and R. Li, "Ensuring content and intention consistency in real-time group editors," in *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, Mar. 2004, pp. 748–755.

[9] J. Grudin, "Groupware and social dynamics: Eight challenges for developers," *Communications of the ACM*, vol. 37, no. 1, pp. 92–105, 1994.

[10] D. Li and R. Li, "Transparent sharing and interoperation of heterogeneous single-user applications," in *CSCW '02: Proceedings of the 2002 ACM conference on Computer supported cooperative work*, New Orleans, Nov. 2002, pp. 246–255.

[11] Wikipedia, "Text editor from wikipedia," Web, 2005, http://en.wikipedia.org/wiki/Text_editor.

[12] ——, "Word processor from wikipedia," Web, 2005, http://en.wikipedia.org/wiki/Word_processor.

[13] E. W. Myers, "An $O(ND)$ difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251–266, 1986.

[14] M. Stefik, G. Foster, D. Bobrow, K. Kahn, S. Lanning, and L. Suchman, "Beyond the chalkboard: Computer support for collaboration and problem solving in meetings," *Communications of the ACM*, vol. 1, no. 1, pp. 32–47, 1987.

[15] J. B. Begole, M. B. Rosson, and C. A. Shaffer, "Flexible collaboration transparency: Supporting worker independence in replicated application-sharing systems," *ACM Transactions on Computer-Human Interaction*, vol. 6, no. 2, pp. 95–132, June 1999.

[16] A. LaMarca, W. K. Edwards, P. Dourish, J. Lamping, I. Smith, and J. Thornton, "Taking the work out of workflow: Mechanisms for document-centered collaboration," in *Proceedings of the Sixth European Conference on Computer-Supported Cooperative Work (ECSCW'99)*, Copenhagen, Denmark, Sept. 1999, pp. 1–20.

[17] W. K. Edwards, "Session management for collaborative applications," in *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, Chapel Hill, NC, Oct. 1994, pp. 323–330.

[18] L. J. Bannon and K. Schmidt, "CSCW: Four characters in search of a context," in *Proceedings of the First European Conference on Computer Supported Cooperative Work*, London, 1989, pp. 12–15.

[19] J. Grudin, "Computer-supported cooperative work: History and focus," *IEEE Computer*, vol. 27, no. 5, pp. 19–26, May 1994.

[20] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements," in *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, Seattle, Dec. 1998, pp. 59–68.

[21] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *ACM SIGMOD'89 Proceedings*, Portland Oregon, 1989, pp. 399–407.

[22] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems," *ACM Transactions on Computer-Human Interaction*, vol. 5, no. 1, pp. 63–108, Mar. 1998.

[23] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, and R. Tomlinson, "MMConf:

An infrastructure for building shared multimedia applications," in *Proceedings of ACM CSCW'90 Conference on Computer-Supported Cooperative Work*, Los Angeles, California, 1990, pp. 329–342.

[24] J. C. Lauwers and K. A. Lantz, "Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems," in *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, Seattle, 1990, pp. 303–311.

[25] D. Engelbart and W. English, "A research center for augmenting human intellect," in *Proceedings of Fall Joint Computing Conference*, vol. 33, no. 1, San Francisco, 1968, pp. 395–410.

[26] H. Abdel-Wahab and M. Feit, "XTV: A framework for sharing x window clients in remote synchronous collaboration," in *Proceedings of IEEE Tricomm '91*, Chapel Hill, NC, Apr. 1991, pp. 159–167.

[27] G. Chung and P. Dewan, "Flexible support for application sharing architecture," in *Proceedings of European CSCW Conference*, Bonn, Germany, Sept. 2001, pp. 99–118.

[28] M. Stefik, D. Bobrow, G. Foster, S. Lanning, and D. Tatar, "WYSIWIS revised: Early experiences with multiuser interfaces," *ACM Transactions on Office Information Systems*, vol. 5, no. 2, pp. 147–167, Apr. 1987.

[29] C. Gutwin and S. Greenberg, "A descriptive framework of workspace awareness for realtime groupware," *Computer Supported Cooperative Work, The Journal of Collaborative Computing*, vol. 11, no. 1-2, pp. 411–446, 2002.

[30] D. Li and R. Li, "Preserving operation effects relation in group editors," in

CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, Chicago, Nov. 2004.

[31] D. Li, C. Sun, L. Zhou, and R. R. Muntz, "Operation propagation in real-time group editors," *IEEE Multimedia Special Issue on Multimedia Computer Supported Cooperative Work*, vol. 7, no. 4, pp. 55–61, 2000.

[32] B. Shneiderman, "Response time and display rate in human performance with computers," *ACM Computing Surveys*, vol. 16, no. 3, pp. 265–285, Sept. 1984.

[33] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, Toronto, Canada, Nov. 1992, pp. 107–114.

[34] C. Gutwin and S. Greenberg, "Design for individuals, design for groups: Tradeoff between power and workspace awareness," in *CSCW '98: Proceedings of the 1998 ACM conference on Computer supported cooperative work*, Seattle, 1998, pp. 207–216.

[35] R. Li, D. Li, and C. Sun, "A time interval based consistency control algorithm for interactive groupware applications," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Los Angeles, July 2004, pp. 429–436.

[36] G. Cobéna, S. Abiteboul, and A. Marian, "Detecting changes in XML documents," in *ICDE '02: Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, San Jose, 2002, p. 41.

[37] Y. Wang, D. J. DeWitt, and J.-Y. Cai, "X-Diff: An effective change detection

algorithm for xml documents," in *19th International Conference on Data Engineering*, Bangalore, India, Mar. 2003, pp. 519–530.

VITA

Jiajun Lu was born in Shanghai, China. He received his Bachelor of Science degree in computer science from Fudan University in China, July of 2001. After that he joined Microsoft(Shanghai) as a developer support engineer focusing on libraries and tools of software development. He enrolled in Texas A&M University in College Station in September of 2003. As a research assistant, his main research topic was real-time collaboration. He received his Master of Science degree from Texas A&M University in May, 2005.

His permanent address is: Jiajun Lu, 3751 Zhongshan North Road, APT 14-1605, Shanghai, China, 200062.

The typist for this thesis was Jiajun Lu.