Design considerations in choosing operations for building groupware systems

Atul Prakash
Michael J. Knister

Software Systems Research Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
Phone: (313) 763-1585
Email: aprakash@eecs.umich.edu, mknister@eecs.umich.edu

Abstract

We have implemented a prototype toolkit, called DistEdit, for building interactive group editors in distributed environments. The toolkit allows different editors (e.g., *vi*, *Emacs*) to be used in the same group session. Based on our experience in building the toolkit, we report here some design solutions that are commonly used in single-user editors, but may not necessarily be appropriate in group editors. Furthermore, design of group editors requires a better understanding of the semantics of editing operations than in the corresponding single-user editors. These semantics are useful for providing a variety of services, such as undo, concurrency control, and selective replay. Finally, we point out that implementing high-level actions is much more complex in group editors than in single-user editors because high-level actions may need to be implemented as a transaction to guarantee correct user-level semantics. This position paper is presented in the context of text editors; however, many of the ideas presented here also apply to other types of group editors.

Introduction

One difficulty in building collaboration systems is that they require solutions to problems in distributed concurrency control, fault-tolerance, user-interfaces, psychology, human factors, and software design [1]. The goal of our DistEdit project [3] is to remove most of the concerns of distributed concurrency control and fault tolerance by providing a library of primitives that can be used to build collaboration tools. The library of primitives can be used to add collaboration support to editors without having to deal with distributed systems issues, such as communication protocols and fault-tolerance.

The current version of the DistEdit toolkit provides support for concurrent updates. To keep response time low, any update is performed locally first and then broadcast to other sites. It is well known that concurrent updates can lead to inconsistencies in the buffer state at various sites [2]. We are using an efficient locking-based solution which requires locks to be acquired only at the start of an insert/delete but not during an insert/delete. So, if the user starts to insert a sequence of characters, there is a slight network delay in acquiring a lock prior to the insert of the first character, but after that, inserts proceed at the speed of the local editor. Another reasonable alternative would have been to use a somewhat more complex scheme suggested in [2], which does not require locks.

In the following sections, we discuss some of the implementation issues that arose during our work.

Choice of document-modifying primitives

The issue of choosing a set of document-modifying primitives is not as critical in single-user editors as in group editors. In single-user editors, whether to treat an operation as a primitive or as a sequence of other more primitive operations is dictated primarily by efficiency concerns. For instance, in a single-user editor, an *IndentParagraph* operation may be treated as a primitive operation rather than composed as a sequence of *InsertChar* and *DeleteChar* operations if it is simpler and more efficient to indent a paragraph by direct access to the document buffer than by calling *InsertChar* and *DeleteChar* operations on the buffer. In a group editor, on the other hand, there are many other factors that need to be considered:

- Every editor built using the toolkit has to be prepared to support all the primitives. Thus, if it is expected that the toolkit will be used in a heterogeneous environment, with different users using different editors in the same group session, all the editors need to have routines mapping the primitives to updates on the buffer. Thus, if *IndentParagraph* is made a primitive in an environment where both *vi* and *Emacs* are expected to be used, both editors need to have support for implementing the command. Clearly, the amount of work required in implementing the editors can be large if the number of primitives is large and heterogeneity is to be supported.

- Undos are much more complex in a group editor [4] than in a single-user editor. In particular, the ability to reverse operations and resequence them is needed for all the primitive operations. It is much easier to provide this capability if the set of primitive operations is small and of well-defined semantics. Thus, from the point of view of *undo*, it may be better to map the *IndentParagraph* operation to a sequence of smaller primitives for which such a capability is already provided.
- Even if undo is not supported in a group editor, there may be reasons to keep the primitives restricted to a small set. For instance, if a scheme such as that in [2] is used to ensure consistency, functions similar to *Transpose* are required to reorder a pair of operations. Defining such functions is much easier if the set of primitives is small and has clear semantics.
- Communication requirements and processing requirement will generally go down if a complex operation is made a primitive operation, rather than mapped to a sequence of smaller primitives. For instance, it is probably cheaper to transmit *InsertParagraph* command rather than the sequence of *InsertChar* and *DeleteChar* operations it might map to.

In general, the tradeoffs in choosing the set of primitives is a complex one in group editors, dictated not only by efficiency concerns, but also by ease of implementation of undo and supporting heterogeneity.

Semantics of Primitive Operations

In group editors, knowing the semantics of primitive operations is more crucial than in single-user editors. Knowing the semantics of the operations is helpful in following services:

- *Implementing undo in a group environment* [4].
- *Handling concurrency* [2].
- *Triggers for notification/awareness:* A user may be interested in receiving notifications when certain parts of the document change. In such a case, semantics of the operation need to indicate the part of the document the operation affects.
- *Integration of divergent document paths:* If a document is being edited by two or more users simultaneously from geographically separated sites, it may be desirable to allow editing to go on despite network partitions and then merge the changes later when the sites get connected. Merging the changes requires handling conflicts if some changes were carried out on overlapping regions of the document. For instance, consider two users, who have to work on the same document but run into a network partition. Assume that each user has a copy of the document. If one user, doing grammar correction, has inserted a word in a paragraph in his copy of the document and another user, doing organizational changes, had moved that paragraph to a different place in the document, typical merge tools will fail to merge the changes. However, with if a history list of the operations carried out is kept at each site, a merged version can be created that has the paragraph moved to the new location *and* has the word inserted in the moved paragraph by merging the operations in the history lists. Such operation-based merging can only be carried out if semantics of the operations (in this case, the copy and insert operations) are well-understood.
- *Recording and selective replay of a group session.* Users who miss a real-time conference may wish to replay portions of it selectively.

We are trying to come up with a formal characterization of the semantics of operations in a group environment that would support above types of services. Some of the attributes we believe will be important in the characterization are:

- Inverse of an operation. Also, if an operation is not reversible, it is useful to know that so that the user can be appropriately warned or the document state checkpointed.
- Resequencing of operations. Ability to resequence operations is needed for ensuring consistency in some protocols [2] and also for implementing undo [4].
- A mapping from an operation to the region it affects in the document.

- type of operation: mode-altering, update, navigation, or access control.
- Computational requirements of an operation and its inverse. For instance, if an operation is known to take a long time, user can be warned about it, state of the document can be checkpointed prior to that operation, etc. Also, if an operation is known to be computationally expensive, it may be desirable to execute it at only one site and transmit the results to other sites, rather than executing the operation at all the sites.
- Effect of an operation on display areas.

The above attributes have to be provided by an application to a toolkit because they vary from application to application. The following features can be provided by a toolkit in an application-independent way:

- Storage of operations in a history list; a history list could be useful for implementing undo, replaying a session, etc.
- With each operation, tags could be stored, such as time of the operation, user who carried out the operation, project name, region affected, etc. These tags could be useful for implementing selective history replay and selective undo.
- Support for compound operations. For implementing undo of a compound operation, for instance, additional information may be needed in the history list to identify the set of primitive operations that compose a compound operation.

Notion of Transactions

Editors often provide high-level operations that map to a sequence of lower-level operations. Whether these high-level operations are treated as a single atomic action or not is an issue that is present only in a minor way in single-user editors but becomes a much more important issue in group editors. In single-user editors, treating a group of simple operations as one larger, user-level operation is important primarily for implementing undo — upon an undo, typically user would like to undo all the changes associated with a single user-level operation, rather than undoing changes partially. An example of such a user-level operation is the *search-replace* command that replaces occurrences of one string by another string throughout the document.

In a group editor, the issue is important not only for the purpose of undo, but also when the operation is being carried out. A real example we faced is in handling *search-replace* command in Emacs. In the single-editor version of Emacs, the *search-replace* is implemented as a Lisp function that searches for the specified string, stores its position and replaces the string at that position with a new string and then starts the search from the stored position added to the length of the replacement string. The function assumes that after the replace, the cursor has moved to the end of the replaced string. In the DistEdit-based Emacs, this algorithm can fail in two ways. One, the replace may fail if someone else holds a lock on the string to be replaced. Second, even if the replace succeeds, position of the string may change due to intervening updates from other sites. So, the algorithm would start searching from a wrong place in the document. A way to avoid the problem would be to treat the *search-replace* as a atomic action.

An example of an operation that may be useful to implement as a transaction is the *IndentParagraph* operation. One may get undesirable effects if another user is allowed to carry out updates to a paragraph during the execution of *IndentParagraph*. One strategy for implementing an operation as a transaction is to determine all the locks the operation needs, acquire them, and then carry out the operation. This raises several implementation issues that also occur in databases, such as what to do if all the locks cannot be acquired, how to handle prolonged transactions, etc.

Summary

We have pointed out that the factors used in choosing the primitives in a group editor are very different than in a single-user editor. We believe that knowing the semantics of the primitives is crucial for a variety of reasons, such as implementing undo, concurrency control, supporting disjoint work during network partitions, and providing numerous other services. We are currently working on developing a formalization of the semantics for operations in a group environment. We have also identified the need for treating a user-level operation as a transaction even though it might map to multiple primitive operations. We are exploring the best way to implement transactions in interactive groupware systems.

References

[1] Ellis, C.A., Gibbs, S.J., and Rein, G.L. Group-

ware: Some Issues and Experiences. *Communications of the ACM*(January 1991), 38-58

[2] Ellis, C.A. and Gibbs, S.J. Concurrency Control in Groupware Systems, in *Proceedings of the ACM SIGMOD '89 Conference on the Management of Data* (Seattle, Washington, May 1989), ACM Press, pp. 399-407.

[3] Knister, M. and Prakash, A. DistEdit: A Distributed Toolkit for Supporting Multiple Group Editors, in *Proceedings of the Third Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 343-355.

[4] Prakash, A. and Knister, M.. Undoing Actions in Collaborative Work. *Proceedings of the Fourth Conference on Computer-Supported Cooperative Work*, Oct. 31-November 4, 1992, Toronto.