MACE: A Fine Grained Concurrent Editor

R. E. Newman-Wolfe

Harsha K. Pelimuhandiram

Computer and Information Sciences Department
University Of Florida

Gainesville, FL 32611

(904) 392-1488

Abstract

MACE is a distributed program running on the X Window System and Unix 4.3bsd\(^2\) sockets that permits fine-grained (character-level) concurrent editing of text files. It runs both as a stand-alone program and as an application in the University of Florida's distributed conferencing system (DCS). MACE uses write locks for concurrency control, allowing a locked section of text to be bounded by any pair of characters in the file. Multiple users may read or edit a file concurrently, with all users receiving updates whenever a lock is removed. The level of sharing is controlled by mutual consent, so that users may collaborate to the degree desired, including the option to view updates in real time. MACE is a first step towards a fine-grained, lock-based approach to concurrent text editing.

\(^1\)This work is partially supported by the University of Florida - PurdueUniversity Software Engineering Research Center.

\(^2\)Unix is a trademark of AT&T Bell Labs.

\(^3\)nemo@chameleon.cis.ufl.edu
1 Introduction

The first computers were designed for a single user to run a single job at a time, and much of operating system design is concerned with isolating users from the potentially harmful effects of one another. Traditional single-user programs continue to dominate available software. Direct communication among users is typically through computer mail or ‘talk’ programs. Indirect communication occurs through changes in databases and files. While talk programs provide better real-time interaction between two users, only recently has much work been done in bringing users together as groups.

Group activities dominate much of the modern workplace. The activity may be product development, scientific research, document preparation, administration, or entertainment. In fact, many of the most common multi-user programs available are games. More recently, several programs intended to allow users to collaborate more closely have been developed. Among these, shared text editors rank among the most useful programs for collaboration.

A scenario typical of the joint production of a document is an initial meeting in which the outline of the document is produced and the responsibility for each part of the document is assigned. Next, the participants work in parallel, but in relative isolation, to produce the parts for which they are responsible. These parts are then assembled into a complete draft, which is passed via electronic mail (or by location in a shared directory) from one person to another for serial editing of the latest draft. At this last stage, serialization can produce many problems, as the person having edit rights to the latest draft essentially owns a lock on the entire paper. This forces others to wait, often until a less convenient time for them, and often under deadline pressures. Coordination is often a problem, since the group must repeatedly decide who should edit the latest draft next. A fine-grained concurrent text editor would eliminate these problems.

MACE has its roots in the gossip multi-way talk program produced by Ramirez and Pelimuhandiram in 1989 [24]. An obvious extension to gossip was support for discussion groups, which led to computer conferencing. From this, the Distributed Conferencing System (DCS) evolved, for which MACE is the shared concurrent text editor [19, 20, 23, 25, 32, 33].

This paper presents the issues and design involved in producing MACE. The following section introduces the concepts of concurrent editing and the issues involved. Section 3 analyzes existing shared editors in the light of the issues discussed in the earlier section. We present the requirements and the design of MACE in Section 4 with the implementational details in Section 5. Finally, we conclude with Section 6.

2 Concurrent Editing

Traditional text editors are not collaboration aware: they cater to a single user editing one file at one time. These editors have no built in control over the possibility of voluntary or involuntary multiple invocations attempting to edit the same file. For example, if a file were edited simultaneously through two invocations of the vi editor, only the changes made through one editor would affect the final version of the file. The changes saved first would be rendered ineffective as the last save would overwrite the first.
Systems such as SCCS [2] and RCS [30] support editors externally in preserving the integrity of files. Files need to be checked out through these systems before changes may take place. This method ensures that only one user may edit a file at one time. The CVS system [14] attempts to overcome this serial access optimistically by allowing overlapping file modifications. When the changes are to be saved, CVS determines whether the order in which two sets of changes are applied to the file matters: if it does not, then the changes do not interfere with one another and they are applied. Otherwise, the users who made the conflicting changes are notified of the problem and resolve it between themselves. If a file is not under heavy use, then this method works well. However, if close collaboration in a short time span is necessary, concurrency control is needed to avoid these collisions and their time-consuming recovery.

Concurrency control is based on avoidance or resolution of collisions. A collision happens when two or more users edit the same section of the file. Collisions can be resolved by ordering the changes in a coherent manner. If the basis for the ordering is time, the result may lack semantic, or even syntactic validity. It is possible to preserve the syntax by updating the edit only by words; however, preserving semantic integrity will be impossible. If modification of sections of text are allowed, then some form of mutual exclusion must be used to maintain integrity.

After an editing session is completed, the resulting document should be consistent at all sites. During editing, the stable, committed portions of the document should be identical at all sites, within some reasonable update delay. It is incumbent upon the concurrent editor to ensure that all users have access to the same document, at least in the parts that are not under active modification.

Two factors describing the nature of updates in a concurrent editor are useful. The update frequency is the frequency (in seconds) with which changes are reflected at a remote site. For example, when one user edits a section of a file, if another user sees the changes happen immediately, we shall say that the update frequency is real-time, whereas if the file is actually updated only once every 30 seconds, then the update frequency will be 30. Alternatively, the updates could be user-controlled by explicit export of changes. Update granularity is the amount of text that is updated at a remote site with each local update. This could take values of one character, one word, one section, one page, or the whole file. The update granularity could also be a user-controlled value.

Related to the issue of updates is the viewing paradigm. There are several viewing paradigms present in existing programs. One early and obvious paradigm for view sharing is the WYSIWIS abstraction. WYSIWIS, or What You See Is What I See, supports a chalk-board type of interaction where every one sees the same view of the shared material. This implies a real time update frequency. A major shortcoming of this method is the lack of privacy to the collaborator as strict WYSIWIS outlaws the presence of a private work space [27, 29, 11]. In addition, many problems present themselves in accommodating user control of his view of the shared data [31]. Another abstraction known as the WYSIWIMS, or What You See Is What I May See, allows independent viewing perspectives and private work spaces [19, 20]. Here, the viewer is given the option of seeing only what he wants of the unstable portion of the shared data. This paradigm is more general than the relaxed versions of WYSIWIS provided in Mermaid [31], in that the users need not have the same parts of the shared objects on their screens, and may not even see the same version of the same part of the shared object. The WYSIWIMS paradigm is similar to the filtering provided in Suite, which propagates changes depending on syntactic or semantic integrity in a structured editor environment [6].
The coexistence of shared and private work spaces creates two environments for the user. A seam is an obstacle in switching from one environment to the other [15]. The WYSIWIMS paradigm allows for relatively seamless transitions between shared and private workspaces. Seams may also exist in between asynchronous communication (e.g., email) and real-time communication. The seam between computer-supported and non-automated work is not an issue when video channels are not available, since non-automated work cannot be shared unless the collaborators are co-located. Reducing the impact of these seams is important for reducing confusion in users and improving the productivity of the collaboration.

In addition to the view-sharing paradigm, there are parameters describing the degree of concurrency permitted by a concurrent editor. Let us define the concurrency index to be the maximum number of users who may edit a file concurrently and editable granularity to be the smallest discrete data item that can be edited. These two are inversely related, since simultaneous editing of a data item by two or more users may result in an inconsistent state.

A highly concurrent editor minimizes the editable granularity to achieve a maximum concurrency index. Minimizing editable granularity also reduces the collaboration distance, that is, the software enforced separation of users. For example, a coarse-grained editor may have an editable granularity of a large text block, which distances other users by preventing them from editing text otherwise unaffected by the writer. Generally, finer granularity will be better, provided that this does not prevent acceptable system performance.

It may seem that a good concurrent editor should have real time update frequency. This is not necessarily true at all times. In many cases, the writer wishes to update the file only when he has composed a coherent edition of the text, or when users have reached an agreement that the change should be made. For example, users often wish to prevent printing of a semantically inconsistent, partially edited version of a document. The Suite system in particular addresses the issue of when updates should be propagated [6], based on the notion of update filtering. Suite supports structured editing, so changes and views may be exported and imported based on syntactic and semantic tests, as well as in raw mode (no tests) or user-committed mode (explicit export). Since MACE is not a structured editor, there are no syntactic or semantic tests for the text to pass, other than perhaps a spell checker. The only semantic check in MACE is that the user explicitly commits the change.

A good concurrent editing system should support features that provide the user ease of use, good accessibility to editable objects, hidden seams, and ability to filter remote information. Ease of use is the responsibility of the user interface that the editor provides. Information concerning other collaborators, such as their identification and their roles in participation, should available but unobtrusive. It might be desirable to know the locality of the other user as well: which section is used by whom and in what capacity. Users should be able to control the circumstances under which their changes are made available to others, as well as the circumstances under which they view changes made by others. Eliminating seams is another important feature in a concurrent editor. The user should not be hindered by the fact that the file is being shared unless it is necessary: he may be made aware of the collaboration but should not be taxed by its existence. In sum, sharing should be as transparent as possible to the user, yet details of that sharing should be available if desired.
3 Other Shared Editors

A very flexible approach to shared text editing uses shared terminal emulators and view-sharing facilities for sharing existing serial applications without modification (Augment [8], Dialogo [17], Rapport [1], Timbuktu [9]). A major advantage of these systems is that any serial application may be shared in this limited way without modification, so users may use the software tools to which they are accustomed. As a particular case, these view-sharing systems permit text editors to be run on a file. However, since the original program is not collaboration aware and consequently has no provision for concurrency control, only one user is allowed to edit the document at a time. Thus the editable granularity is the whole file and the concurrency index is unity. Some floor control mechanism is used to pass control between users to allow access to the editing facility. Typically, updates are made in real time in these systems. Because the concurrency index is so low, these systems place a substantial collaboration distance between users even though they allow multiple users to observe editing as it occurs.

CES, Collaborative Editing System [12, 26], is one of the earlier shared editors. The text is partitioned into sections that are owned by each author. The editable granularity of CES is equal to one section, which is coarse. CES does not support real-time updates.

Quilt [10, 18] and the PREP editor [21] are collaborative document production tools that support long-term collaboration. They support annotations and message passing among the collaborators. However, these systems lack tightly coupled real-time collaboration. Neither of them support real-time updates, nor do they have fine editable granularities. Quilt breaks multimedia documents into sections, with possibly different access to each section, and serial access on a per section basis. The concurrency index depends on the number of sections (or really, the set of users who are allowed to edit the sections), and the editable granularity is one section. The use of roles in Quilt is very interesting, and only has a parallel in the larger DCS environment of MACE, in which participants may have read-only access to conference documents.

Of all the extant concurrent editors, GROVE is the most fine grained of which we are aware [7]. It also supports real-time updates. However, GROVE is not a full-fledged text editor, but is rather an outline editor and uses lines of text as the grain size. GROVE avoids locking by use of insert and delete as the only primitives, which allows consistency to maintained while permitting highly concurrent access. GROVE consequently has a high concurrency index and a small editable granularity.

DistEdit [16] is a toolkit that allows programmers to build shared editors for collaboration in real time. Observers may see the at most one ‘master’s’ edit in real-time. DistEdit does not support concurrent editing as it features a strict floor passing mechanism in taking turns at editing the file. An attractive feature of DistEdit is its ability to collaborate with heterogeneous editors.

4 MACE Requirements and Design

Our requirements for MACE are mostly based on the issues discussed in the previous section. In addition, MACE was built to be inserted into the University of Florida’s DCS distributed conferencing system as the text editing application. However, we required that MACE exist as a stand alone system as well, and therefore, its incorporation in DCS should not limit its capabilities.
The editors discussed earlier do not support both real-time and fine-grained concurrent editing. MACE is designed as a tool for tight and synchronous collaboration. This requirement calls for a minimum editable granularity and the support of real-time updates. At the same time, collaboration should be handled in such a way that the users still have some control over their privacy.

Some products even though powerful, are underutilized because of their difficulty of use or unfamiliarity [13]. The requirements for MACE call for a user friendly editor. We also require that MACE be capable of supporting heterogeneous editing command sets and interfaces that allow collaborators to use interfaces familiar to them or customized to their tastes. The fact that the user is editing a shared file should not burden him unless the necessity arises. At the same time, he should be provided with remote data that is necessary but not redundant to him.

The name of the editor MACE, the Mother of All Concurrent Editors, stems from the main architectural requirement. The aim is to provide a modular concurrent editor as a basis for others to use and modify any of the modules in developing more concurrent editors. In the next section, we shall see the capabilities of MACE that meet our requirements.

MACE is invoked through the X Window System [28]. It appears on the user’s screen as another window that can be iconified and resized. The window does not occupy the whole screen and the user is free to move between the MACE window and his private work areas. The file to edit is loaded through the window. The names of the people who are editing the current file is accessible to each user of that file. One may change files while keeping the same window. Multiple MACE windows may be invoked by a user to edit the same or different files. The editing commands are bound to the set of keys and the rest of the functionality is achieved through menu and mouse click operations. We have tried to minimize the number of mouse clicks by using drag menus, and not prompting the user with redundant questions. MACE is a modal editor that has three modes: scroll, edit and updates.

When a file is first loaded into the edit window, the editor is in scroll mode. The user is free to scroll and view any part of the file. Occasionally, he may see certain parts of the file change due to committed updates by other users.

Edit mode is achieved after acquiring a pair of locks (top and bottom locks) that envelopes the section of text to be edited. These locks logically partition the file to grant exclusive rights to the user as a writer while editing that part of the file. Others may view but not interfere with the writer or the section of text while the locks are held. Any number of writers may edit the file within the regions of their own locks. The editor will not switch to edit mode if a requested lock happens to invade a section of text that is already locked. The user may save or undo his changes at any time during the edit. Either action results in his relinquishing the locks and reentering scroll mode. Saving commits the user to update the file letting other viewers see the new version. A remotely committed update will appear on the screens of all other users with the changed section of text within their view. A MACE edit window allows a user to lock only one section of the file. However, if he wishes to lock more than one section, he may invoke multiple MACE windows to the same file and place locks as desired.

MACE’s command set for editing is similar to that of the emacs text editor. Cut, copy and paste operations could be done between MACE and other work spaces using simple mouse clicks. Replace operations apply only to the section that the user edits.
When the editor is in scroll mode and if other users have placed locks in the vicinity of the viewer, he is notified of the remote edit. The viewer may see the locks by requesting to do so. Clicking on a lock pops up a menu with the identification of the remote user and three options: single update, real time updates and extra copy.

When a single update is chosen, the viewer’s screen will be updated to show the remote editor’s screen and the viewer is put back in scroll mode. At most times, the text will appear disjoint since the remote user is caught in the middle of his edit session. If the viewer scrolls out of this area, the update will be lost and he will see the unedited text in its previous state if he returns to it before the writer commits his changes.

The real-time update option will resize and synchronize the viewer’s screen to the screen of the remote editor. The editor enters update mode and every change that the remote user makes is reflected in real-time on the screen of the viewer. The viewer exits back to scroll mode when he wishes or when the remote editor gives up its locks.

The writer has the right to deny updates to a viewer by turning on a private lock. By doing so, the writer obtains the privacy to compose his edit before updating the file or undoing the changes. View locking does not affect the viewers in scroll mode.

The extra copy editing feature is not implemented in the current version of MACE. It will allow a user to edit a section that is already being edited by another. Once an extra copy is chosen, the user may edit the section between the original locks. The editing user will be able to export extra copies to other users. Private locking will prevent others from obtaining extra copies. We plan to use the voting mechanism of DCS [25] in deciding which copy ultimately is saved on to the file.

When extra copies are present, a lock is mapped to more than one user. Therefore, when updates are requested, the viewer will be given the option of choosing any remote editor to view the edit session. By invoking multiple MACE windows, a user may synchronize to any number of users that edit the extra copies. We hope that this will be a useful feature in comparing different versions of documents.

The concurrency index of MACE is infinity. In theory, there is no limit to the number of users that can edit the file in parallel. For practical reasons, the present version puts a limit of sixteen users per file, but this can easily be extended to any number. In MACE, edit locks may be placed in between any number of contiguous characters. One may even place the locks in between two characters, at the beginning or end of the file and start inserting at that point. Thus, MACE has an editable granularity of zero bytes.

MACE could be considered as having three viewing modes: static, snapshot, and synchronous. In static or scroll mode, the editor employs the WYSIWIMS viewing paradigm since each user is free to look at any part of the text that has been committed. The update frequency is controlled by the writer as only the committed writes will update the file, and the update granularity is the size of the section written at update time. The second mode is the one time update or snapshot. Here, the viewing paradigm in the editor is WYSIWIS just at the time the update is obtained, provided the screens of the viewer and the writer are of the same size. From that point on, the viewer sees the text in WYSIWIMS as the writer may or may not continue to update and the viewer, to scroll. The update frequency is viewer controlled and the granularity is one page. Finally, synchronous mode
puts the viewer's MACE window in WYSIWIS. The update frequency in this mode is real-time as all changes are reflected immediately. The update granularity is equal to each edit stroke of the remote user which may be as low as one character.

MACE is the concurrent text editor of DCS. The Distributed Conferencing System (DCS) at the University of Florida is a distributed package providing real-time support for cooperative work. In this system, a set of mechanisms for conference management supports synchronous communication, a notion of user roles, a voting mechanism and a wide range of floor control paradigms. The small, flexible message-passing interface to the conferencing management processes permits shared applications to be installed easily in DCS. Currently, DCS has applications that support concurrent development of text and graphic documents; remote demonstration, testing and debugging of programs; and automatic creation of transcripts of meetings including motions made and voting results.

The features of DCS enhance the power of MACE in close collaboration and control. The DCS discussion window is used for synchronous communication among the users. DCS membership control limits the usage of files and users to those within a conference. MACE inherits DCS's voting mechanism and the user roles which add control to the access of shared files. Members who have writing privileges will be able to edit the files while observers will be restricted to viewing. We have proposed to invoke the voting mechanism in removing locks of users who forget to release or hold on to the locks for prolonged periods of time. As noted previously, the voting mechanism of DCS will be used to decide which copy will be saved when extra copies are present.

5 MACE Implementation

The implementation platform of MACE was chosen on the criteria of availability, portability, extensibility, speed and the ease of implementation. MACE is written in C, a language that can be exploited for speed of execution and portability. The first version is implemented on the Unix operating system which has a wide usage in academic and scientific circles. The user interface is built on the X Window System. The choice of widgets as opposed to the Xlib saved the trouble of rewriting the lower level menuing, window mapping, and event parsing routines. The interprocess communication is made through TCP sockets [3] The choice of sockets for communication was made over the RPC abstraction for reasons of speed and for wide area networking (WAN) capabilities.
An architectural trend in the past in producing concurrent editors has been to enhance an existing single user application to be collaboration aware by building a communication layer below the application. As exemplified in DistEdit, this method suffers from severe limitations in concurrent sharing. Another method is to build a concurrent editor as a whole application by itself. A disadvantage of this method is the inability to collaborate with heterogeneous editors. We have opted to design MACE's software architecture as an open system. The system has been modularized such that it may easily be replaced with a different interface, edit command set or even an editor. The lowest level which is the networking interface also could be customized to comply with the underlying communication system. Another feature of MACE's architecture is its process distribution. It uses a hybrid architecture such that the processes are spread among users' workstations avoiding any overloading. The communication and process synchronization of MACE depend highly on broadcasting and multicasting of messages. We have organized the processes in a star topology to minimize discrepancies in message delays.

MACE consists of three types of processes. Figure 1 shows process hierarchy and the connection topology of the whole system. The topmost process in the hierarchy is the file manager (FM) that handles session control. Each session consists of an editor manager (EM) (generically, an object manager) and a collection of editor windows (EWs) (generically, dialog managers) that make up a partially replicated architecture.

The vertical process modularization roughly conforms to the layering of the ISO OSI [22] system. The main difference is that data units do not travel through all layers at all times. For example, the session layer is used only during session establishment and completion, and the presentation layer is used only when data conversion is necessary. The transport layer is the interface to the underlying network. This layer routes data units to higher layers and also performs packetizing of data units to and from the TCP socket layer. Concurrency control and the editing is done at the application layer. Any of the layers may be modified with little or no effect to others. For example, the transport layer could be changed to support ISDN communication in the future.

The FM is the main daemon of MACE and it exists in a well-known location. It holds file information of sharable files and the EM information associated with the files. The FM's primary responsibility is session management. A session starts when a user loads a file in MACE. The initial EW contacts the FM to request a file list, from which the user chooses the desired file, or requests that a new file be created. If there is no EM associated with the requested file, the FM spawns a new EM,
which then acts as a coordinator for that file. As additional users enter the editing session on that file, the FM connects them to the EM until the session terminates by all users leaving the session. The FM then kills the EM. Figure 2 shows the layers of the FM’s architecture that conform to an open system. The FM consists of a session and a transport layer.

Each file that is edited is associated with an EM. It is the server that controls the edit session. Since the EM is the bottleneck of the topology, we have designed it to carry a minimum process load. The EM is not involved in the actual text edit. It is responsible for the paging mechanism, granting of locks and message multicasting. Figure 3 shows the modular structure of the EM.

The EW is the process that the user invokes to access MACE. It generally runs on the local machine and is replicated at each user location. The EW carries the heavy processing load of editing the file. Figure 4 shows a schematic view of the EW. The application layer is vertically modularized into a control unit, an editor and a user interface. The interface is programmed with the Xt toolkit. In this implementation, a customized version of the Athena text widget is used as the text editor. Therefore, the interface and the editor are tightly coupled. The editor by itself is not collaboration aware. The control unit is the module that carries the responsibility of making the EW collaboration aware. The session layer is used only during session establishment and completion. When the editor is in edit mode, the user interface directly routes the user input (key strokes) to the editor. When a remote user has requested real-time updates, the editor multiplexes the user input to the presentation layer, which converts the key strokes into a canonical form and delivers them to the network through the transport layer. When viewing real-time updates, the switching process blocks the user input and demultiplexes the auxiliary input into the editor via the presentation layer, which converts the input into key strokes that represent the command set of that particular editor. This method of implementation allows users to collaborate with editors having heterogeneous command sets.

The data is partially replicated in MACE. The EM keeps the main copy of the file being edited. Each EW is delivered partial copies of the file in parts that are known as pages. Pages are demanded from the EM, added at one end and discarded at the other as the user scrolls through the file. The edit is performed on the pages at the EWs. More than two pages can be loaded if the editing is
Figure 4. The Edit Window
done over a larger area. The changes are sent to the EM only when the user saves his edit. Thus, the EM is not flooded with each keystroke from all editors.

A partial replication of the file was chosen in order to avoid unnecessary updating of all copies. Only the users who have common pages will be updated by a saved edit. Others will load the pages from the updated file at the EM only if they happen scroll past that particular section. For example, when A in Figure 5 writes the edit, only B might have to be updated. The assumption for this approach is the principle of locality, which suggests that most edit changes will be made in a local area of the file.

Concurrency control of shared files is commonly implemented through three methods: locking, timestamping and optimistic concurrency control [4]. Optimistic concurrency control may not be used in concurrent text editing due to the risk of losing changes. Timestamping requires that the text file be identified by sections since timestamps are associated with each section that is accessed. This method imposes restrictions on the editable granularity as it is not feasible to treat each character as a separate section that could be timestamped. Locking proves to be the most plausible concurrency control mechanism in handling concurrency in text files.

MACE uses locking as the means of concurrency control. The locks logically partition the file enabling MACE to have a fine editable granularity. When a lock is first requested at the EW, it consults the local information on collisions. If no collisions are present, the request is sent to the EM. The EM checks for collisions at a central lock table and grants or refuses the locks. If the locks are granted, they are broadcast to all editors that are affected. The filtering at the EW reduces the number of unnecessary lock requests that reach the EM.

In general, groupware architectures have taken two forms: centralized and replicated. Centralized architectures have a single instance of the application that executes on one machine. Process concentration in a single machine is one of the problems associated with this architecture. Another notable problem is the “stickiness” [5] of play back at remote sites since the input as well as the output is subject to network latency. A fully replicated architecture runs an instance of the application at each user site. Replicated architectures suffer from the difficulty of process synchronization. The stickiness issue is eliminated by the quick response of the local copy of the program. However, replicated architectures suffer from the problem of process synchronization.
In MACE, we use a partially replicated architecture where the EM acts as a central coordinating agent. EM handles two forms of serialization: lock requests and file updates are temporally serialized in a first-come-first-served (FCFS) manner, and spatial serialization of logical locks guarantees collision avoidance in edits. It could be said that the file updates are pre-ordered since the locks are granted before the edit takes place. Thus, we use collision avoidance in ensuring data integrity during normal edit sessions. When there is an extra copy edit, the method used is collision resolving. This will be achieved through a voting mechanism.

MACE processes communicate to each other through a message passing paradigm. The message delivery and reception are asynchronous. There is no acknowledgement system nor do processes ever wait for specific replies. This method of message passing helps us prevent deadlock. Connection oriented TCP sockets guarantee that the messages preserve their order in the communication medium.

6 Conclusions

MACE is a collaboration-aware, concurrent text editor that uses fine-grained locking for concurrency control. It makes several types of sharing possible, implementing a WYSIWIMS (What You See Is What I May See) paradigm. The sender (editor) and the receiver (viewer) have joint control over the degree of view sharing.

The architecture of MACE is federated, a hybrid between replicated and centralized architectures. By using an intermediate editing notation, any number of editing user interfaces may be supported. Because of this modular design, MACE may be considered an open system for editing, allowing third party components to be added later.

In the future, we would like to allow a viewer or even an editor to see real-time updates from more than one other editor. There are some screen update semantics that are not yet clear to us, such as how multiple insertions in the same viewing window should affect the portion of the text viewed, that is, should there be an ‘anchor’ and if so, where? Means for permitting and controlling commitment of multiple versions of sections of text are desirable. Additional editing interfaces that do not require X, as well as some supporting a command interface more familiar to some users (such as vi) would permit greater flexibility and more use of MACE.

As part of the Distributed Conferencing System (DCS), MACE inherits facilities for real time communication with collaborators, mechanisms for controlling the disposition of jointly created objects, and other application windows that may also be useful. The stand alone version of MACE is primarily used as a tool for maintaining shared files consistently, but it lacks sufficient support for collaboration over time. MACE has no facility for indicating changes made during one’s absence, nor does it have a mechanism for annotating text. We plan to incorporate a history versions option to alleviate this problem. By performing a comparison operation on the previous version, a user will be able to identify the changes that were taken place. Providing MACE with annotations will raise issues in the close collaboration aspect of the system. The sharing semantics of the annotations themselves will have to be analyzed before such an enhancement is made.
References


