MULTIMEDIA JOINT EDITING BASED ON RESERVATIONS

Erik Wilde

Swiss Federal Institute of Technology (ETH Zürich)
Computer Engineering and Networks Laboratory
CH – 8092 Zürich
wilde@tik.ethz.ch

appeared in: Proceedings of the 3rd Australian Multi-Media Communications, Applications and Technology Workshop, Wollongong, Australia, 14th-15th July 1993

Abstract

Joint editing as opposed to "normal" editing is an activity carried out by several people simultaneously. It raises the problem of coordinating write access to a document. The approach described in this paper uses an editing model of reserved regions and a client/server architecture. Any region of a document may be selected and reserved (provided that it is not reserved already) and may then be changed by the owner. Other users can only read it. The software basis of the editor is the Andrew Toolkit. This allows the use of arbitrary media types within the document.

1.0 Introduction

Joint editing (also referred to as collaborative editing) is an area of research which was established nearly 10 years ago. It describes the general development of computer applications from tools for the individual user to software for the support of whole working groups (groupware[1]). This has influenced most areas of computer science research.

In research and industry environments, most documents such as technical reports, papers, studies, or similar material are written by a number of people, not just one single author. Most of the time, documents are worked on concurrently, i.e. a number of authors want to change the document at one time. Conventional (i.e. single user) editors do not support this type of work, since they provide write access to all parts of a document and thus are not appropriate for collaborative work, even if they are used by more than one person (e.g. by using window system multiplexors to allow input from several users to one program).

Collaborative editing therefore needs to be provided by specialized programs which "know" that several users are concurrently editing one document and therefore provide mechanisms to avoid access conflicts. Problems may arise when several people try to change one paragraph concur-

[1] The rapidly growing field of *Computer Supported Cooperative Work (CSCW)* is the discipline which concentrates on this issue. CSCW research focuses on using computers for workgroups instead of individuals (for an overview on CSCW see [1]).

rently or when one person edits a part of the document which is at the same time moved or even deleted by another person.

The editor described in this paper is one part of the *MultimETH* conferencing system. However, the editor may be used totally independent of the conferencing system and thus this system is only mentioned when absolutely necessary (e.g. when it comes to communication issues, which are implemented based on the conferencing system's communication mechanisms). The interested reader is referred to [2], which contains an extensive discussion of the conferencing system and its properties.

2.0 The Editing Model

Two major issues have to be taken into account. The first is how the distribution of users is modeled within the system, i.e. which processes reside on which systems and which tasks do they have. The second is how these distributed processes interwork, i.e. how the distributed entities are designed to make joint editing possible by coordinating their activities.

The model of the distribution of processes within the editor described here is fairly simple. We chose a client/server approach to map the human users and a central entity (which is responsible for permitting changes to the document) onto processes. There are two main reasons for this decision (see [3] for an in-depth discussion).

- Joint editing can be seen an example of several people doing work that has to be synchronized. A server can be used to achieve this goal; it is responsible for allowing access to selected parts and making sure that no inconsistencies can occur.
  Since it is connected to all clients (users), the server is ideally suited to control all write accesses to a document. It can also easily determine who has the right to modify something if two users issue a corresponding request simultaneously.
- The conferencing system MultimETH, which is used as a base for the editor, is also structured in a client/server manner. That is, when using the editor with the conferencing system, every user is already represented by a client process (implementing the front end of the conferencing system) and there is one central server entity which is used as the conferencing center.
  If the editor's model of processes is identical with the one of the conferencing system, the transport mechanisms of the conferencing system can be used. Therefore, it is not necessary to deal with communication between computers in this case. Only local communication facilities need to be used to handle data between the editor's distributed components and the conferencing system's processes. This approach also reduces the number of connections necessary.

The second main issue mentioned above is the model of joint editing used by the editor, i.e. the method that is used to make collaborative editing possible. Our approach is based on the observation that typically, every user wants to see and to be able to browse through the whole document, but changes are only made to small (and most of the time few) regions of it. Immediate updates of changed regions are not a critical issue[2], although it must be possible to request the current ver-

sion of any reserved part on request or periodically. Consequently, we make the whole document available for reading, but only explicitly selected regions may be modified.

To satisfy the requirements described above, we provide a mechanism to select and reserve regions of the document, which are (after having been reserved) owned by the user who requested the reservation. Any consecutive region of the document may be selected, i.e. it is not necessary to select complete sentences, paragraphs, or sections. The selection can be made using the normal selection mechanism of the editor, which makes the reservation mechanism an integral part of the editing process. After the reserved region has been modified, it can be set free and becomes a normal part of the document which may then be selected by another user. Figure 1 shows the operations which can be used when working with selections and reservations and how they depend on each other.

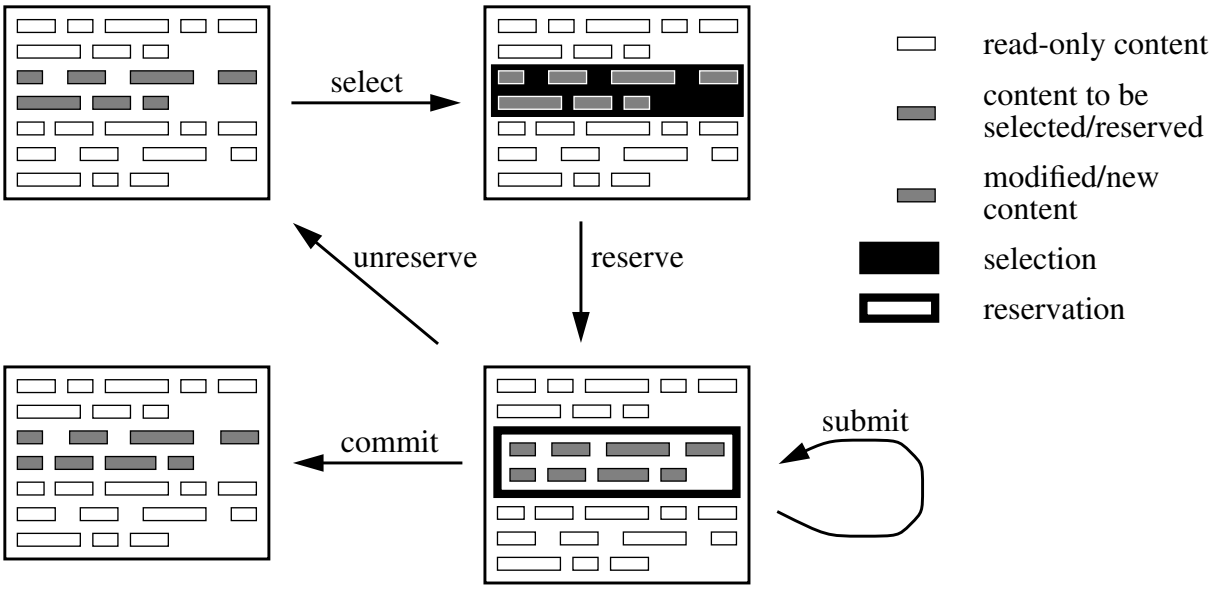

Figure 1: Reservation Model[3]

The figure shows our reservation model and also gives an example of how working with the editor can look like. For changing a certain region of the document, that region must be selected and reserved. After a successful reservation, the region may be modified and finally the changes can be accepted or the original text can be restored. The following list gives a short overview of the operations we have defined.

[2] This question is very controversial. While some people believe that it is necessary to have high speed links connecting the users to provide immediate updates of all media types, we assume that it is sufficient to have a restricted level of synchronicity.

[3] The content within the document could also be a graphic or some other media type, not only text.

- select
  The select operation is the mechanism normally used within editors, i.e. the mouse is used for marking consecutive regions within the document. Thus, there is no explicit select operation or mechanism we have to implement, we simply take advantage of the available operation.
- reserve
  After selecting a region of the document using the select operation, it is possible to use the reserve operation to turn the selection into a reservation. This operation will reserve the selected region for the user who performed it, i.e. no other user of the joint editing session will be allowed to reserve anything within this region of the document. The server is responsible for handling reservations, which are granted to clients on a first-come-first-serve basis.
- submit
  Because there is a certain level of asynchronicity within our editing model (there is no immediate update), there is a special operation which can be used to propagate the changes applied to a reserved region so far to the other participants of an editing session. Submit does not change the status of the reserved region (i.e. it remains reserved), but it causes an update to be sent to all other editor clients[4].
- commit
  If the reservation is no longer required (because all modifications are made or because another participant of the editing session wants to reserve that region), and the changes should be applied to the document, the commit operation is used. After committing the changes, the modified region becomes a normal part of the document (i.e. the reservation's content is integrated into the document) and can be reserved again by any user.
- unreserve
  If, for any reason, the changes are not to be integrated into the document, unreserve may be used to give up the reservation and restore the original content of the reservation. In this case, all changes made within the reservation are discarded.

All operations are context dependent, i.e. it is only possible to perform a certain operation when the context is appropriate. Figure 1 identifies which operations are allowed in which states. The small set of operations makes it easy for users to remember the joint editing facilities of the editor.

3.0 Implementation Aspects

We have implemented the editor using the Andrew Toolkit (ATK) [4], which is part of the Andrew System freely available with the distribution of the X window system. While the Andrew

[4] We decided to allow everyone to use the submit operation, i.e. not only the user who owns a reservation may use this operation. Thus, if any participant is interested in an update of a reserved region, he simply uses the submit operation for requesting the actual content from the owner's client process which is then also transmitted to all other clients.

System itself contains many complete applications (such as a messaging system and a distributed file system), the ATK is a collection of building blocks which can be used for programming multimedia applications. The ATK itself is portable and running under several window systems, the version we have used is based on the Xlib library of the X window system.

The ATK uses a language of its own, which is a superset of the C programming language. It uses a preprocessor to translate the ATK code into plain C code which is then processed with a normal C compiler. The language extensions provide an object-oriented programming environment. The definition of each class is divided into two parts, the class header, which defines the interface of the class, and the class implementation, which contains the code that is executed for each method call. Unfortunately, multiple inheritance is not possible, which would have been advantageous in some cases.

3.1 ATK Insets

The basic building blocks of ATK applications are *insets*. Insets are mainly used to implement new media or information types which should be usable with different ATK applications. Using insets, it is not necessary to change the actual applications. Every inset can be viewed as a pair of classes conforming to a set of conditions (which are necessary to make the interworking of insets and ATK applications possible), one class implementing the *dataobject*, the other class implementing the *view*.

- The *dataobject* is responsible for maintaining the data of an information type and providing the facilities for reading and writing the data. Thus it can be seen as that part of a new information type that is independent from its presentation.
- The *view* of an inset is responsible for displaying the data. A view does not store any data that is relevant to the information type itself, only issues which are of interest to the presentation alone are within the view's responsibility.

The separation of information types into a data and a view part is useful for various reasons. One (more abstract) reason is the conceptual goal of keeping the representation of data clearly separated from its presentation. This enables the user of insets to easily create effects which might be otherwise hard to implement, such as having two consistent views onto one dataobject.

Without any coding efforts, it is possible to create a document which contains a set of numeric data and two different views on this set, e.g. a table and a pie chart. Because both views use the same dataobject for storing the data, every change of the table also changes the pie chart and vice versa, so there are no consistency problems.

Every inset has to satisfy a small set of requirements which are used to make sure that it can be used with nearly every ATK application. While implementing our editor, we faced the problem that every inset handles its own data (i.e. there is no user programmed, central input routine[5]). That is, keyboard (and mouse) inputs are passed down the view hierarchy to the view of the inset

[5] Actually, all input to an ATK program is handled by the interaction manager *im*, which is a special object that receives all keyboard and mouse events and sends it to the "right" view.

which has the input focus and are handled there. Since users should only be able to modify reserved regions of the documents, all other parts must be set read-only. Unfortunately, this is a nonstandard inset property, so that only some insets offer a read-only feature (some as methods, others as variables). However, this property is not mandatory, i.e. not defined in the superclasses dataobject and view.

Consequently, we introduced a new mandatory method which must be implemented by all insets which should be used with our editor. It can be used to set an inset to read-only and should be implemented recursively. With this method, it is possible to set all unreserved regions to read-only, otherwise it would be possible to modify an inset's data even if it is not reserved.

3.2 The Joint Inset

The standard editor provided with the ATK is *ez*. It is an editor program which can be used to handle and manipulate ATK insets. Although it is possible to use ez for editing with multiple users (using multiple views and displaying them on different X servers, a feature that is already built into ez), there is no coordination or access restriction. Therefore we used ez as a base for our editor, extending its functionality with joint editing features. What we needed was a way to distinguish between different users and to restrict write accesses to explicitly reserved regions.

We thus changed the ez application as well as the *text* inset that is normally used with it. The major change was to add a *reserve* command which can be used to reserve selected regions of the document. We used the text inset as the basis for the joint editing inset. Reserved regions are placed in a special type of inset, which is described in detail in the following section. Because the clients are implemented stateless, a reserve operation is simply transmitted to the server and nothing is remembered inside the client. The server then reacts to this request by either granting a reservation which is then transmitted to all clients (including the one that asked for it) or ignoring the request if the reservation could not be made (e.g. because an already reserved region has been selected[6]).

Another modification necessary were the file commands *load* and *save*, which normally work locally, i.e. they read from and write to the local file system. We changed these commands to work over the network, because a save operation should not create a local copy of the document but perform a save operation on the server side. The load operation was modified accordingly.

3.3 The Jnote Inset

The joint application (i.e. our collaborative editor) uses a special inset to represent reservations within the document. We implemented this inset using a given inset as a basis. The *jnote* inset's implementation is based on the *note*[7] inset. We modified the inset to display the name of the

---

[6] If a reservation is part of the selection, the server reserves all content until the beginning of the first reservation. If only existing reservations are selected, the server does not create a new reservation.

[7] The note inset is provided with the standard ATK distribution. It is part of a set of insets created by the MIT. Originally, this inset was designed to make annotations to documents (either open or closed) to allow readers to make comments to documents without "changing" them.

owner of a reservation in its title bar and to provide the operations necessary within a reservation's context, i.e. submit, commit and unreserve. Furthermore, we implemented the submit mechanism to be executed automatically (we called this the autosubmit feature), in case the user wants to send or receive updates periodically instead of initiating submit operations manually. The status of the autosubmit mechanism is also displayed in the jnote's title bar.

These additional operations are available in new items within the menu bar of the jnote inset, such that a user may perform one of these operations whenever he is in the right context (i.e. is working within a jnote). All operations are executed immediately, except the commit operation, for which the user is prompted to type in a short (a few words) description of the changes, which is then added to the document's change log[8].

The autosubmit mechanism works on an adaptive basis. It is configurable with regard to the minimum and maximum delay before a submit operation and can be used to periodically update existing reservations. The adaptation procedure takes the difference of the volume of the two last submit operations to calculate the delay until the next submit is executed. This difference should give an rough estimate of the modifications applied to a reservation. Consequently, the adaptation is always a little bit "too late" and fails to take replacements into account, but it is sufficient for most cases, except when the size of a reservation changes very rapidly (e.g. using large cut or paste operations). Autosubmit may only be enabled by the owner of a reservation.

The jnote inset is implemented in a way that allows it to store all relevant information needed for a reservation. The implementation is based on the ATK's dataobject/view separation which again is very useful in this case. Any jnote view might use at most three different contents stored by the dataobject. One of these is only necessary for the jnote's owner.

- The *original* contains the content originally reserved by the requestor of the reservation. This content is needed if the owner finally chooses the unreserve operation to throw away all changes and restore the original content of a reserved region.
- The *editable* content is necessary for the owner of a reservation only. It is the content that is displayed and directly accessible for making changes in the jnote view's window of the reservation's owner. The editable content is copied to the submitted content (see below) for each submit operation.
- The *submitted* content of the dataobject contains the most recently submitted content of a reservation. In case of a commit operation, this is the content to be inserted into the document[9]. The submitted content is the content displayed by all jnote views which are not owners of the reservation (i.e. all other clients).

---

[8] The change log is part of a document's administrative information and provides a list of triples consisting of a change's date, the name of the user who made the change, and the description entered at the prompt after having selected the commit operation. Thus the change log represents a short history of all changes applied to a document.

[9] The commit operation is implemented in a way that first a submit is executed which copies the editable content of the owner to all submitted dataobjects, which are then integrated into the document.

This use of several content portions within one dataobject makes it easy to manage the different content versions which have to be stored for each reservation. In fact, all jnote views use the same jnote dataobject, but they display different content portions. Performing certain operations (such as submit, unreserve or commit) is as easy as copying one content to another. The update of views whenever the dataobject changes is handled by the ATK automatically, so the coding of the reservations has been relatively straightforward.

3.4 The RMC package

The RMC (Remote Method Call) package has been implemented to allow the execution of instance methods of objects located on remote systems. Whenever a reservation is requested, the client's joint instance executes a special method on the server's joint instance (which is responsible for handling reservations). This remote execution is supported by a special package[10] which allows the client to execute methods on the server's objects and vice versa. RMC is used by registering with the RMC package. The class and the instances must be registered in order to be able to execute a RMC call or to be called by a remote object. The RMC package stores the registration information and pointers to the objects and thus is able to pass the method call to the appropriate instance when it receives such a request over the network.

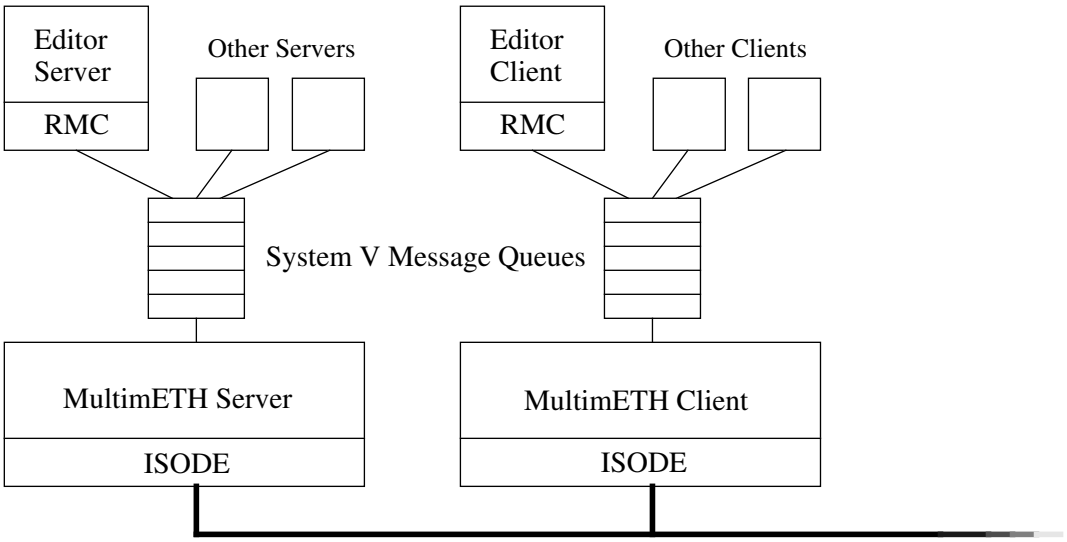

Figure 2: Communications Implementation[11]

Figure 2 shows the basic structure of the MultimETH system which is used for the transport of messages between the editor's components. The RMC part of the editor code is the package that is

---

[10] A package (in ATK terminology) is a special class with no data and no methods; packages are mainly used as subroutine libraries.

[11] ISODE is a communication system development environment which implements several OSI services and protocols. MultimETH uses ISODEs implementation of the *Remote Operations Service Element (ROSE)* and the *Association Control Service Element (ACSE)* of the OSI application layer.

used by various objects to communicate with objects on remote systems. The RMC part is structured again, i.e. it is divided into two parts.

- The first part of the RMC layer (called rmc module) is the communication independent part. It is responsible for providing the interface for the objects using RMC procedures. It is used by all objects which want to call remote methods.
- The second part of the RMC layer (called comm module) is the part dealing with the actual communication. It is the only part that has to be adapted if a new communication subsystem (e.g. sockets) is used. The communication requires an order-preserving, non-duplicating, 8 bit transparent service with no length limitations for the messages being sent. If any of these requirements are not satisfied, they must be implemented within the comm module.

Two comm modules were implemented. The first one uses TCP/IP over stream sockets, which was easy to implement because they satisfy all requirements listed above[12]. This module was used with the first prototype for testing purposes. The second comm module was implemented for the use of the editor within the conferencing system. Because we used the conferencing system's communication mechanisms, the editor was integrated using System V message queues.

The implementation of the message queue comm module was more difficult because we needed a segmentation/reassembly mechanism to get around the length limitations within message queues. Now the method calls are segmented at the caller's side and the comm module on the performer's side reassembles the packages. Messages in the message queue are identified using different mechanisms. The type field in the message structure is used to indicate that the message is for the editor client or server (and not for one of the other processes also using the message queue). It is set by the originator of the message to indicate the source of the message. The message is then transported over the network connection and the conferencing system puts it into the message queue on the other side. This method makes the distribution of the processes transparent to the programs using the message queue interface, because they can use the two message queues on different computer systems as if they were using one local queue.

4.0 Usage of the Editor

The usage of the editor is easy and intuitive if the user is accustomed to the normal ATK editor *ez*. The additional commands available in the MultimETH menu item are used to reserve regions or to work with a reservation, depending on where the cursor is located. This is possible because the ATK provides context-specific menus. Figure 3 shows a screendump of an editing session with the editor. The object located in the upper half of the window is the *access* object which is used to store administrative information for a document (such as the change log and access rights). The reservation is represented by the window with the title bar saying that the reservation belongs to Peter who does not use autosubmit at the moment.

---

12) Messages are indicated by NUL-terminated byte streams. NUL was excluded from the range of valid characters for the message content.

There is one special command in the jnote inset's *reservation* menu item which has not been mentioned until now because it is only of local significance. It is the *open window* menu item which can be used to display a reservation in a separate window. This command is implemented using ATK's separation of views and dataobjects. Open window creates a new view in a separate window (using a new interaction manager and a new frame) and thereby enables the user to either make changes in the separate window or in the normal reservation window within the document.

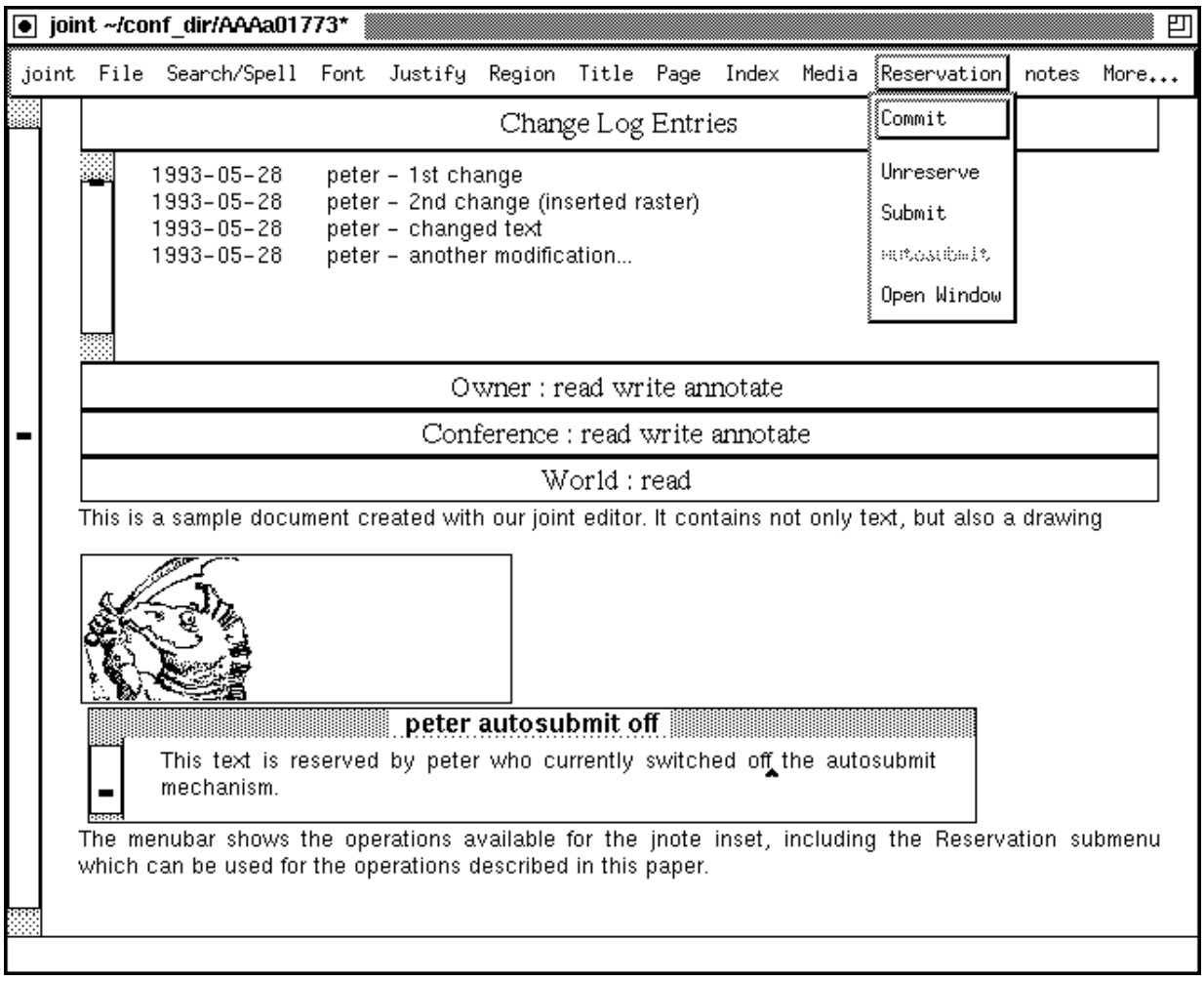

Figure 3: Sample Editing Session

A normal way of working with the editor is therefore given by the following sequence of activities, which is not necessary, but useful for collaborative editing sessions.

- The editing session is started using the conferencing system's mechanisms. The users who want to participate in the joint editing must all be active in one conference. They can use the conferencing system's communication mechanisms such as *broadcast* and *chat* (the system also provides audio-conferencing capabilities) to discuss which document should be edited.

- After editor clients have been started by all participants[13], they see the document without any reservations (provided that it was not saved with active reservations, which is possible) and can start looking for the regions they want to edit. Each participant reserves all the regions he needs. Conflicts are prevented by the server and may be discussed using the conferencing system. After all reservations are made, the document is seen as a sequence of reserved and unreserved regions.
- Now the participants select the *open window* menu item for all reservations they own. They get a separate window for each reservation. After having opened the separate windows, they can *close*[14] their reservations within the document. The participants now can see the document with all reservations made by other participants within the document window, while they can edit their own reservations within the separate windows.
- For each reservation to be committed or unreserved, the appropriate menu item is selected in the menu bar of the separate window. After having entered the description of the changes (in case of *commit*), the separate window disappears and the reservation's icon within the document is replaced by the modified content. If new reservations are made, the procedure described above can be applied again.

To avoid the communication overhead that is caused by the client/server connection when the editor is used by one person only, a stand-alone mode has been implemented. In stand-alone mode, the client is the only component necessary, and runs without a server. All reservations are handled locally, which is sufficient because there is only one user. Reservations may exist because documents can be saved with active reservations. In this case the reservations are still valid when the document is edited and only free regions or regions belonging to the stand-alone user may be modified. Stand-alone mode has been implemented by changing all methods which require communications with the server to be performed locally.

Working with the editor showed that in most cases it is sufficient to have asynchronous but periodic updates of the reserved regions. Only in case of very slow network connections, the delay caused by periodic updates may be too large, but in this case it is possible to reduce the autosubmit frequency or only transmit changes when really necessary, using the submit operation. The editor is not meant for shared whiteboard usage[15], which requires a synchronous connection between the users, so in this case other tools should be used.

There is some other work related to the editor, namely a converter package which can be used to convert documents in ATK format to *Rich Text Format (RTF)* and back to ATK format. Because of the differences of ATK and RTF documents, some information may be lost. Reservations can not be represented in RTF (there is no concept of access restrictions), but they are formatted in a

---

13) The editor server is started automatically if a conference is opened.

14) Close does not affect the reservation. It is a display oriented command which causes the jnote to be displayed as an icon instead of an editable window.

15) Shared whiteboards are used within conferencing applications for sharing a common drawing area between distributed users. For this type of application it is crucial to have synchronous communications for immediately sharing drawings and perhaps cursor locations (see e.g. [5]).

way similar to the joint editor. Another related project is a browser which allows to browse through the document space both on the server or on the local machine. It is used as a part of the conferencing system and can be used to select a document and then launch the editor.

5.0 Conclusions

The design and implementation of an editor for joint multimedia editing has been described. The underlying model follows a client/server paradigm which divides the editor into user interfaces (clients) and a central entity (server) which is responsible for administering changes to the document. The implementation uses an object-oriented toolkit (ATK) which is capable of dynamically adding media types to the system. Every new media type needs at least a basic set of methods which are used to integrate the new objects into the document. Specific methods may be used to edit and present new types, but they are not necessary from the editor's point of view. The model of periodic, asynchronous updates of the reserved regions is suitable for almost all joint editing uses, except when a high level of synchronicity is required.

Acknowledgments

I would like to thank all people who contributed to the whole project, especially Werner Almesberger and Markus Wild, who implemented the first prototype, Daniel Bauer who implemented many improvements, Niklaus Ruess who built the converter package for exchanging documents with existing applications, and Germano Caronni who programmed the browser.

Bibliography

[1] Greif, I. (ed.), "Computer Supported Cooperative Work: A Book of Readings", Morgan Kaufmann Publishers, Inc., 1988

[2] Lubich, H.P., "MultimETH: Ein Beitrag zur Konzeption eines Echtzeit-Multimedia-Konferenzsystems", Informatik-Dissertationen ETH Zürich Nr. 20, 1990

[3] Seliger, R., "Design and Implementation of a Distributed Program for Collaborative Editing", MS-Thesis, MIT, MIT/LCS/TR-350, 1986

[4] Borenstein, N., "Multimedia Applications Development with the Andrew Toolkit", Prentice Hall, 1990

[5] Lubich, H.P., "A Small Conferencing System and Shared Whiteboard as a Testbed for Distributed Multimedia Applications Using OSI Protocols", submitted to IFIP 6.5 ULPAA '94