NetEdit: A Collaborative Editor

Ali A. Zafer, Clifford A. Shaffer, Roger W. Ehrich, and Manuel Perez
Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
azafer|shaffer|ehrich|perez@vt.edu

ABSTRACT
We present a collaborative text editor named NetEdit. NetEdit uses a replicated architecture with processing and data distributed across all clients. Due to replication, the response time for local edits is quite close to that of a single-user editor. Clients do not need explicit awareness of other clients since all communication is coordinated by a central server. As a result, NetEdit is quite scalable (linear growth) relative to purely distributed systems (quadratic growth) in terms of number of communication paths required as the number of clients grow. NetEdit uses an n-way synchronization algorithm derived from the synchronization protocol of the Jupiter collaboration system. Along with describing the editor, its architecture and its synchronization algorithm, we present the results of a usability study that evaluated the collaboration awareness tools included in NetEdit.

KEYWORDS: Computer-supported cooperative work, collaboration, text editor, replication.

Introduction

The 1980's and early 90's saw several efforts to produce text editors capable of supporting multiple simultaneous users. Examples include GROVE [6], ShrEdit [4], and Jupiter [11]. Surprisingly, with the recent explosion of Internet use and the availability of new programming environments that would seem to be ideal for implementing collaborative applications, there are few collaborative editors available currently. Notible implementations include the REDUCE editor prototype [15] and Microsoft's NetMeeting [10]. NetMeeting (which is really an environment for allowing coarse-grained sharing of single-user applications such as Microsoft Word) is mostly notable not for its qualities as a collaborative environment, but rather for the fact that its the first commercially viable collaboration system and as such is exposing many new users to the opportunities that collaborative tools can provide.

While we currently do not have many examples of collaborative editors, a great deal of work has taken place in recent years on the underlying infrastructure necessary for such editors to be successful. Recent work has shown that replicated approaches are viable [2, 13], whereas the 1980's typically saw centralized architectures. A great deal of progress has been made on the synchronization algorithms necessary for coordinating the various replicas, in particular a number of variations on the concept of operational transformations [6, 13, 12, 11]. In later sections, we will summarize the issues and the various algorithms that have been proposed.

This paper describes NetEdit, a web-accessible collaborative editor, its design and implementation, and a usability study that evaluated its collaboration awareness tools.

An Overview of NetEdit

NetEdit provides three fundamental functionalities: centralized file and session management, unconstrained group editing of documents, and chat session management for communications between users and groups of users outside of the document or editor session.

NetEdit (see Figure 1) allows two or more users to remotely edit a document simultaneously. The editing is completely unconstrained and users can insert and delete characters at any location. In fact, two or more users could be performing insertions and deletions at exactly the same position.

NetEdit supports collaborative awareness through the use of two interface devices: the radar view and telepointers. Collaborative awareness can be defined as the knowledge of the state or actions of other participants. Collaborative editors generally use one of two styles of collaboration [8, 1]. The first is WYSIWIS (what you see is what I see) which corresponds to tightly coupled group activity. NetMeeting is an example of a system that enforces tight coupling of the view, in that all participants see exactly the same part of the document at all times. Due to this tight coupling, there is not much need to distribute awareness information, as all the participants are working at the same position or on the same artifact. The second collaboration style is called location-relaxed WYSIWIS and corresponds to the ability of participants to each work in separate sections of the document. Thus there is a need to provide appropriate widgets to make them aware of each other's activities.

Telepointers [8, 1] give an indication of the remote partici-

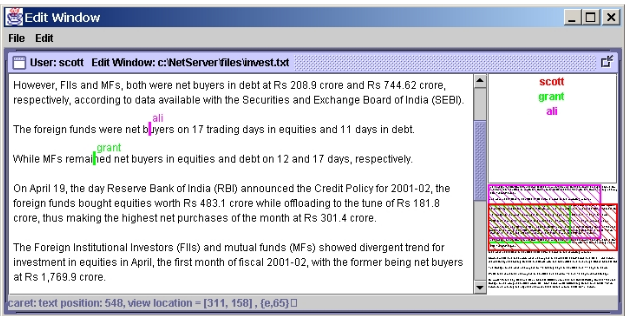

Figure 1: NetEdit's main editor window.

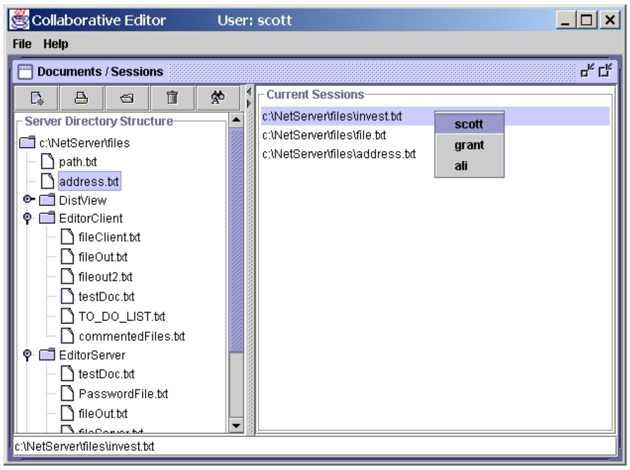

Figure 2: NetEdit's document session window.

pant's pointer location. A variant of this is a **telecaret** (shown in Figure 1) that corresponds to the caret position of the remote participant. It is to be noted that a telepointer or telecaret is useful and needed only when the two remote participants are working in sections of the document that are close enough for these pointers to be seen.

Radar Views [8, 1] provide the extent of view in the document for each remote participant. A radar view compresses the entire document into a miniature view and displays that in a window. Each remote participant's view in the document is indicated by a shaded rectangle of his color on this window, as shown in Figure 1.

Document and session management is provided by a central server that holds the persistent copies of the documents. A "session" is a single group that is editing a document. Users registered to the system may freely join or create a session. A user may be in multiple sessions at one time, and a session is defined to include all users editing a specific document. A separate window (Figure 2) is used to manage the user's participation in sessions, and a separate editor window is created for each session that the user is currently in.

NetEdit provides a chat window (Figure 3) that allows users to broadcast messages to specific individuals currently online in NetEdit, to all users currently online, or to all users currently in the specified session.

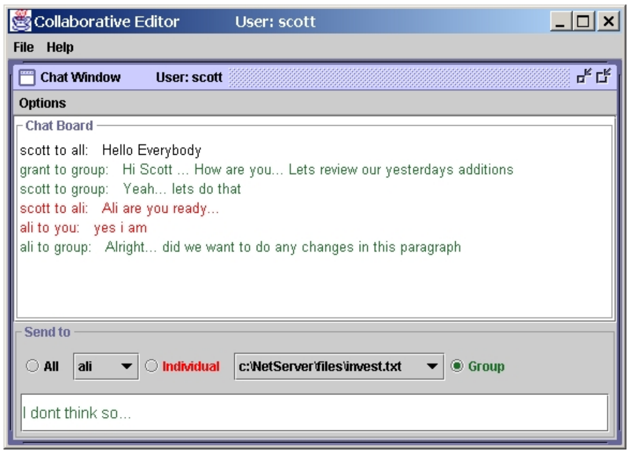

Figure 3: NetEdit's chat window.

NetEdit's actual editing facilities are minimal in the prototype. Support is provided for adding and deleting text, and for block cut, copy, and paste. This functionality is more than sufficient to test most synchronization protocol issues, including aggregation of network messages to reduce total traffic delay. From a user's point of view, it is sufficient for simple memos and student assignments, but clearly needs to be improved. Although NetEdit currently does not support style formatting (e.g. bold, italic, etc.), the system architecture does not prevent implementing these features.

NetEdit is written in Java. The editing window itself is built on Swing's JTextPane widget to hold the document, whose default implementation for its Document interface has been overridden to provide the core conflict management algorithm and transformation engine.

System Architecture

NetEdit aims to provide editing speeds that do not appear signicanlty slower to the user than would a single-user text editor. Groupware systems are inherently distributed, with their components having a protocol for communication between them. Their architecture can range from completely centralized to completely replicated [7, 3, 2]. A middle ground between them corresponds to a hybrid architecture [3] that is replicated but also has certain centralized components.

Centralized architectures use a single application program, residing on one central server machine to control all input and output to the distributed participants. All data and processing reside on this central machine. Client processes at each site are only responsible for passing requests to the central program, and for displaying any output sent to it from the central program. The advantage of a centralized scheme is that synchronization is easy: State information is consistent since it is located in one place, and events are handled at clients in the same order because they are serialized by the server. Its main drawback is latency, as the message corresponding to any action must pass from the client to the server and back again before response to the action is shown.

Replicated architectures execute a copy of the program at every site. Data and processing are distributed to all the remote participants. Thus each replica must coordinate explicitly both local and remote actions, synchronizing all copies of the document.

Replicas need only exchange critical state information to keep their copy of the data current. While remote activities may still be delayed, a person's local activities can be processed immediately. Processing bottlenecks are less likely – each replica is responsible for drawing only the local view, unlike the central model, which must update the graphics of all the client's screens. However, the cost of replication is increased complexity as issues of distributed systems like conflict management, concurrency control, etc., must be handled.

Strictly replicated systems have an enormous growth in terms of the number of communication paths; it grows at the rate of $n(n-1)/2$, where n is the number of clients in the system. As compared to this, centralized systems have linear growth. Thus, centralized systems are more scalable in terms of communication requirements than replicated systems.

Semi-replicated hybrid architectures [3] contain both centralized and replicated components. In such systems, data and processing are replicated at each client but the communication between them goes through a central authority. NetEdit uses such a semi-replicated architecture, with each editor client holding a copy of the document and immediately processing local actions. Actions are then passed as messages to the central server, that then rebroadcasts these messages with appropriate information for coordinating the various clients, as discussed in the next section.

When the server starts, four service threads (ServerLogin, ServerDaemon, Server Awareness Manager, and ChatServer) are created. ServerLogin listens at a port known to the clients, and it knows the ports at which the other services are listening. Each client establishes a socket connection with ServerLogin. After authorization, ServerLogin sends port numbers of the other three services to the client, which can then establish direct socket connections with these services.

When the client establishes a socket connection with ServerDaemon, the server creates a proxy (ClientProxy) that is responsible for maintaining the client's view of the file hierarchy at the server. It executes requests like renaming a file, deleting a file, opening a file etc.

ServerDaemon maintains the sessions that are currently active. An EditServer object is created for each session. A proxy for each client (EditClientProxy) is created at the server, that maintains communication regarding insertion and deletion of characters from the document. These proxies for all participating clients together implement the core conflict and consistency management server-side algorithm for group editing. This algorithm is described in detail later.

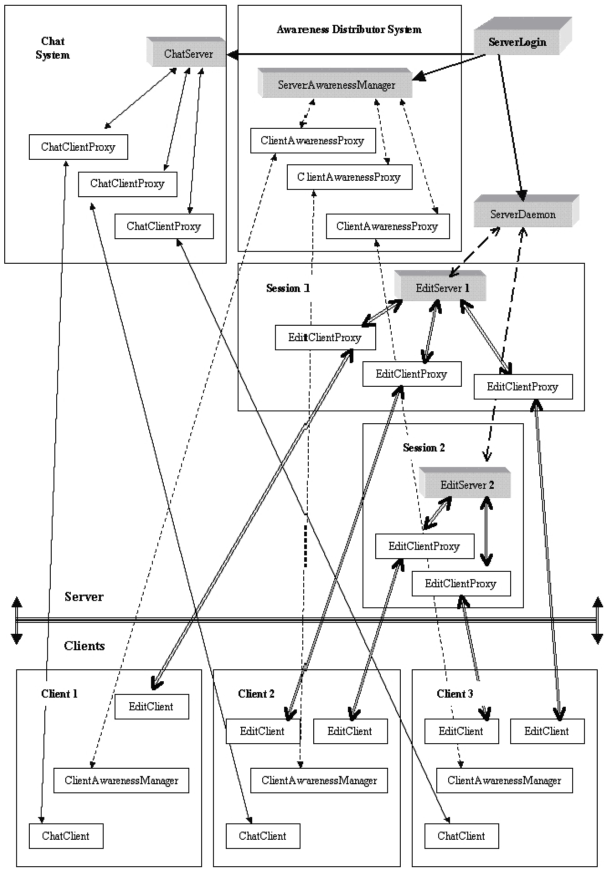

Figure 4: NetEdit System Architecture.

Awareness information is transferred explicitly between server and clients. Each client has a single proxy called ClientAwarenessProxy at the server, regardless of the number of sessions in which it is participating. This proxy is responsible for transferring all the awareness information learned from other clients to this client. The chat server's architecture is similar to that of ServerAwarenessManager.

On the client side, ClientDaemon first establishes a socket connection with ServerLogin and gets the port numbers of other services – ServerDaemon, ServerAwarenessManager and ChatServer. ClientAwarenessManager is responsible for distributing the awareness information updates, obtained from ClientAwarenessProxy, to various graphical user interface widgets – radar views, telecursors, participant's list, etc., at the client. It is also responsible for sending updates back to ClientAwarenessProxy when the state of this client changes – like scrolling to different part of the document, exiting a session etc.

EditClient is a thread object forked for each document that the user is participating in for editing. It communicates all the insertions and deletions through a socket connection with EditClientProxy at the server. The processing at both EditClient and EditClientProxy, and the communication between them, constitutes the core conflict and consistency management algorithm that manages group editing.

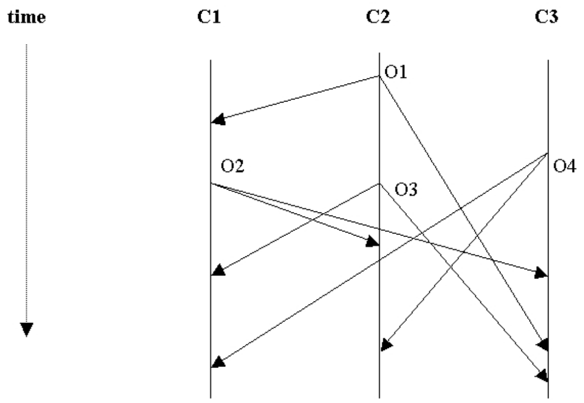

Figure 5: A timeline for propagating four operations across three clients.

ChatClient creates the chat window and exchanges streams of chat messages with with ChatClientProxy at the server.

Issues in the design of a concurrency control algorithm
The 1990's saw major advances in the underlying communications protocols necessary to make such tools successful. Probably the most significant advance, which makes replicated and semi-replicated architectures such as NetEdit viable, is development of the concepts of the operational transformation [5] along with recognition that consistency includes a series of issues beyond simple syntactic consistency between replicas [13].

Suppose there are three clients C1, C2 and C3 that are performing the four edit operations O1, O2, O3 and O4 as shown in Figure 4. Suppose that the edits are applied to the local copy immediately after they are generated and sent to the remote clients where they are applied in their original form, without change of any kind. As a result the following inconsistency problems might arise.

Divergence Due to non-deterministic communication delays, the messages might arrive and be executed in different order at different clients. In Figure 5, the order of arrival for messages is O1, O2, O3, O4 at C1; O1, O3, O2, O4 at C2; and O4, O2, O1, O3 at C3. The operations could be insertions and deletions and hence are not commutative. Thus the state of the document at the three clients might well be different if the operations are performed in different orders. This is unacceptable for a group-editing tool.

Causality Violation As shown in the time-line diagram, O2 is executed after the execution of O1 at C1. Thus O2 is causally dependant on O1. As a result, it is necessary that this order of execution be preserved at all the clients. However, due to non-deterministic communication delay, it cannot be guaranteed that the messages will arrive at each client in this order. We use Lamport's notation [9] for defining causal ordering relationships.

Causal Ordering Relation "$\rightarrow$" Given two operations Oa and Ob, generated at sites i and j, Oa $\rightarrow$ Ob if and only if

1. $i = j$, and the generation of Oa happened before the generation of Ob.
2. $i \neq j$, and the execution of Oa at site j happened before the generation of Ob.
3. there exists an operation Ox, such that Oa $\rightarrow$ Ox and Ox $\rightarrow$ Ob.

Dependent operations Given operations Oa and Ob, Ob is said to be dependant on Oa if and only if Oa $\rightarrow$ Ob.

Independent operations Given operations Oa and Ob, they are said to be independent (occurring simultaneously or concurrently) if and only if neither Oa $\rightarrow$ Ob, nor Ob $\rightarrow$ Oa. This can be expressed as Oa $|$ Ob.

Note that divergence and causality violation can be solved by enforcing a total ordering on the messages. This can be done through the use of a central coordinator who time-stamps each message before sending it to the clients who then execute the instructions based on this total order. This approach does not have good responsiveness (due to round trip delay before a local operation can be applied to the local document), nor can it solve a third inconsistency problem known as intention violation.

Intention Violation In Figure 4, O2 and O3 were generated and applied when the document was in the same state at both C1 and C2. O2 was generated without any knowledge of O3 and vice-versa, thus the two operations are independent. Due to this concurrent generation of operations, the actual effect of an operation at the time of its execution may be different from the intended effect at the time of its generation. Suppose the document at C1 and C2 before the execution of O2 and O3 is "ABCDEFGH." Let O2 be insert[`1234`, 4] (insertion of "1234" at offset 4). Let O3 be remove[2, 4] (delete 2 characters at offset 4 – the intention is removal of "EF"). Executing these two operations and preserving the intention of C1 and C2 should result in "ABCD1234GH." However the state at C1 would be "ABCD34EFGH" if we consider these to be truely independent operations. This violates the intention of operation O2, as string "12" that it inserted is not in the document, and also the intention of operation O3, since "EF" remains present in the document.

Note that these three inconsistency problems are independent of each other and the resolution of one does not guarantee the resolution of others. The problems of divergence and intention violation are of different natures. Applying some serialization technique can solve the former but the latter requires transformation to the operations before they are applied to the document.

Concurrency control algorithms in groupware systems
In this section we summarize some of the algorithms developed to solve the concurrency problems just discussed. These algorithms all distribute processing and data. They use optimistic concurrency control where the basic idea is to take an operation executed in some past state and to transform it

in a way so that it can be applied to the current state.

dOPT Ellis et al. [5] presented dOPT, the first concurrency control algorithm for group editing based on distributed operational transformations. Their model has no central entity. All clients have a copy of the document, and each updates its own copy. Operations generated locally are sent to all other clients. When a remote client receives an operation, the state of its local document might be in conflict with the incoming operation. Hence these incoming operations are transformed, that is, their offset is changed (incremented or decremented) depending on whether they are inserts or deletes and the type of conflict. Since the processing of these remote and local operations is distributed to all the clients, and the remote operations get transformed when they occur concurrent to local operation, this algorithm is referred to as distributed operational transformation (dOPT).

Two research groups working independently discovered a flaw in dOPT [14, 12]. When two users are editing a document, and one issues more than one operation concurrently with an operation by the other user, the document might become inconsistent. For example, O2 and O3 of Figure 4 are generated in the same state and can be transformed against each other. Since O2 and O4 are generated from different states of the document, they cannot be transformed against each other. The dOPT algorithm did not take this into account and did the transformation in both cases. Both groups proposed solutions, as discussed below.

REDUCE REDUCE [14, 13] performs two types of transformations to the incoming messages: inclusion and exclusion transformations. Inclusion transformations transform messages when they are generated from the same state of the document. Exclusion transformations require more processing since here messages are not generated from the same state of the document. REDUCE maintains a history buffer (HB) that keeps track of all the operations that have been executed and is used to perform the transformations.

As with dOPT, all clients maintain a copy of the document. Local operations are distributed to all other clients. When a new operation O at a client's site is causally ready for execution, the following is done: First, all operations in HB that causally follow this operation are undone to restore the document in a state before their execution. Next, operation O is applied to the document. Finally all the undone operations are redone. This series of undo/do/redo operations must be performed as a single transaction.

Jupiter The Jupiter collaboration system [11, 13] uses a central server to coordinate communications, and a two-way synchronization protocol that allows 2 participants to be in sync with each other during editing. Jupiter's implementers suggest that one can use this 2 party synchronization protocol to achieve n-way synchronization. Jupiter uses a 2-dimensional state space graph (Figure 6), instead of a history

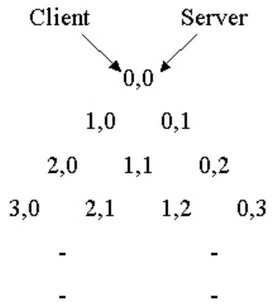

Figure 6: A 2-dimensional state space graph.

buffer as in REDUCE, to keep track of paths to follow during operation transformation. The state graph ensures that the pair of messages undergoing transformation have originated from the same state of the document.

AdOPTed As with REDUCE and the dOPT, adOPTed [12, 13] requires no central coordinating entity. AdOPTed clients maintain an N-dimensional interaction model graph to keep track of all valid paths of operation transformations. This N-dimensional interaction model graph can be viewed as a generalization of the 2 dimensional state space graph in the Jupiter algorithm. By choosing the right paths in this interaction model, the algorithm ensures that any pair of operations involved in the transformation originate from the same state of the document.

REDUCE and adOPTed are completely distributed. Since in a collaborative system no client has ownership of the document, it is not clear in these approaches where the document will be stored and maintained. Both systems are more resilient to failure as compared to the centralized Jupiter system – a failure of one client does not affect the other collaborating participants.

Due to the distributed nature of REDUCE and adOPTed, they are more complex than the Jupiter algorithm. REDUCE and adOPTed require each client to know all the other participants in the system so as to to establish a communication path with them. Thus, the number of communications paths is quadratic on the number of clients. In comparison, Jupiter uses a central server for communicating messages, and for storing documents. However, the clients are not dumb terminals, but need to do their part of processing to the messages received from the server.

The NetEdit Consistency Algorithm

Our goal is a collaborative editing system that is simple, fast, and scalable. The 2-way synchronization protocol developed for Jupiter served as our starting point. We extended this two-way protocol to a multi-way protocol. We used multiple two-way component synchronization to achieve multi-component synchronization protocol.

All the clients maintain a state-space graph [11] as shown in Figure 5, and the server maintains a state-space graph for each client. The state space is used so that each client-server pair can maintain information of where the other is, relative to it, in the editing process. Both the client and the server pass through this state space as they process messages. Each state is labeled with the number of messages from the client and server that have been processed to that point. For example, if the client is in state (2,1), it has generated and processed 2 messages of its own, and has received and processed 1 message from the server. If the server and the client process messages in the same order, then they will follow the same path in the state space graph.

The algorithm labels each message with the state the sender was in just before the message was generated. The recipient uses these labels to detect conflicts. One can transform two concurrent messages only when they are generated from the same state of the document. Otherwise, special handling is required.

The clients also maintain a buffer that contains operations that have been generated and applied locally but have not been acknowledged by the other party. The server maintains one such buffer for each client.

Client Processing When the client receives a message, say s1, from the server with state space value of (a1, a2) then it searchs its buffer from the beginning (i.e. the oldest entry in it) and start discarding those messages with state space (b1, b2) from the buffer such that $b1 < a1$. These are the messages that server has already received and processed.

The client then transform s1 with respect to the next message (the first message after the discarded messages) in the buffer. This is the message that was executed in parallel to s1 and also when the document was in the same state. It might be that there are no messages left in the buffer after discarding; in that case we simply apply the message directly to the document. Otherwise, call this transformed message s1′. Next, transform s1′ with each remaining message in the buffer as follows:

$s1'$ = transforms ($s1'$, next message in the buffer);

Transformation [14] consists of changing the offset of the message at which it is applied in the document, so that the new offset is consistent with the execution of other concurrent local operations. As noted earlier, imposing a global order on the operations can take care of convergence and causality, but its not sufficient to preserve intentions. It is necessary to transform independent operations with respect to each other appropriately, as explained next.

Consider the operations O2 and O3 from Figure 4. Both these operations have originated when the document is in the same state at C1 and C2. Lets say the document contains “ABCDEF” before the execution of either O2 or O3. Also consider that O2 is insert[`123`, 2] and O3 is insert[`abc`, 4]. When C1 receives O3, the document at C1 after applying O2 but before executing O3 is “AB123CDEF”. Since (1) O2 and O3 were independent operations, (2) Both O2 and O3 were generated from the same state of the document and (3) the offset of O3 is greater than the offset of O2, O3 needs to be transformed with respect to O2 at Client C1. The transformation essentially consists of adding 3 (size of O2) to the offset of O3. Thus O3 now gets transformed into insert[`abc`, 7]. This when applied to the document gives “AB123CDabcEF.” Similarly, at client C2, O2 will be transformed with respect to O3 giving the final state of the document at C2 as “AB123CDabcEF.” Thus we see that the intentions of both O2 and O3 were preserved. Similar transformations are required for delete-delete, insert-delete and delete-insert combinations. It is to be noted that when you have delete as one of the operations, it might involve subtraction as well.

The transformation of two operations is warranted only when they originate from the same state of the document. To emphasize this point, consider operations O3 and O4. O3 has seen the effect of execution of O1 but this is not the case with O4. Hence these two operations in their original form cannot be transformed.

The client applies the final transformed message to the document in its current state. While it is transforming s1 with the messages in the buffer, it also transform those messages (the ones in the buffer, say c2, c3, c4 ...) into c2′ , c3′ , c4′ ... and stores them accordingly in the buffer along with updated state space values. That is, they are stored with their original state space values except that the server component – the second component of the 2-tuple – in each will be incremented by one. The state of the client now goes from (x, y) to (x, y+1). This procedure is repeated for all the messages that it receives from the server (i.e., for remote operations).

The operations that are generated locally are applied to the document directly and also stored in the buffer with proper state values (i.e., the state the document was in when the local operation was generated). After applying this local operation, the client moves from state (x, y) to $(x + 1, y)$.

Server Processing Assume that there are 4 clients (C1, C2, C3 and C4) in the system and S1, S2, S3 and S4, respectively, are proxies for them. Let the buffers of S1 through S4 be named as q1, q2, q3 and q4 respectively. Suppose S1, which maintains communication with C1, is in state (x1, y1). Similarly S2, S3 and S4 are in states (x2, y2), (x3, y3), and (x4, y4) respectively.

When message c1 comes from client C1 having state space value (a1, b1), S1 will search its buffer (q1) from the beginning (the oldest entry) and discard those messages that have state space (u1, v1) from q1 such that $v1 < b1$. These are the messages that client C1 has already received and processed.

The server transforms c1 with respect to the next message (the first message after the discarded messages) in the buffer q1 of S1. This is the message that was executed in parallel to c1 and also when the document was in the same state. It might be that there are no messages left in the buffer q1 after discarding; in that case the server simply applies c1 directly to the document. Otherwise, call this transformed message c1′. Next transform c1′ with each remaining message in the buffer (q1) in order until the end is reached as follows:

$c1' =$ transforms ($c1'$, next message in q1);

The final transformed message (say m1) is applied to the document in its current state. While transforming c1 with the messages in the buffer (q1), the server also transform those messages in q1 and restores them in q1 with updated state space values. That is, the messages are stored with their original state space values except that the client component – the second component of the 2-tuple – will each be incremented. The client component of the state for S1 is also incremented.

Add m1 to the buffers of C2, C3 and C4. The state stored in q2 for message m1 would be (x2, y2). Similarly m1 in q3 and q4 will have state (x3, y3) and (x4, y4) respectively associated with it. This corresponds to the states for S2, S3, and S4, respectively, when the message m1 was processed by S1.

m1 will then be sent to all the clients, that is, m1 with state value (x2, y2) will be sent to C2; m1 with state value (x3, y3) will be sent to C3; m1 with state value (x4, y4) will be sent to C4. S2 goes to state (x2, $y2 + 1$). S3 goes to state (x3, y3+1). And S4 goes to state (x4, $y4 + 1$). All the above must be executed as one atomic operation, that is, this processing is completed before considering another editor operation.

At quiescence, S_i must have state same as client C_i for all clients in the system.

The NetEdit Preliminary Usability Study

When users work in groups in the same physical space, they are aware of each other's activities through what they see and hear. In a collaborative system the participants are geographically separated from each other, so workspace awareness is an important functionality [6, 8, 7]. In NetEdit, telepointers and radarview provide workspace awareness. to the users. Apart from external modes of communication like the telephone etc., the participants in NetEdit communicate with each other through a chat window.

To test the usability aspects of NetEdit, we performed a preliminary study. Our study goals were to:

1. Determine the efficacy of the awareness widgets during group editing: were users able to correctly interpret their change, were users distracted by them, etc.
2. Study the efficacy and level of use for the chat window. In particular, was it sufficient for communication during the group activity?
3. Determine the level of usability for other functionality in the system.

Methodology To evaluate the effectiveness of the awareness widgets and the assess the general usability of NetEdit we conducted a formative evaluation with two groups of participants. They were asked to collaborate in the creation of a three-page document using NetEdit. We used observation, self-reporting, questionnaire and discussions with the participants to assess the usability of our tool.

Participants We used students in the Computer Science department, both graduates and undergraduates. A total of 10 participants were used, divided into three groups. The first group was used as a pilot test of our experimental procedure. All three participants in the pilot group were graduate students. The other two groups, one with three and the other with four participants, were used for the formative evaluation. There were 6 undergraduates and 1 graduate student. All undergraduates at the time were registered in the Human-Computer Interaction course being offered by one of the authors (Perez). Eight of ten participants had not used a collaborative tool before.

Experimental setup Participants worked on Microsoft Windows machines located in separate closed rooms. The only way they could communicate with each other was through the chat utility provided by NetEdit. An experimenter was assigned to each participant to observe his or her activities, and take notes from whatever was said during the course of the experiment. The participants were asked to speak aloud their intentions and any remarks they had while performing the assigned task. All communication and editing activities were logged.

Experimental Task The task assigned to the participants was to write as a group a document no longer than three pages. The topic of the document was their evaluation of the usability of the interface of NetEdit. Each user was responsible for doing the usability evaluation of one of the three windows making up the system (Figures 1-3). Even though each evaluated one window, the final document was to be organized based on the following eight characteristics.

1. Visibility, Mapping, Feedback
2. Whether responses from the system make sense
3. User control and freedom
4. Recognition rather than recall
5. Aesthetic and minimalist design
6. Ability to help users recognize, diagnose, and recover from errors
7. Online help and documentation
8. Gulf of evaluation and execution

Hence this exercise made users work in a loosely coupled manner when they were doing their own evaluation and in a tightly coupled manner when they were putting together the final report. The entire process was required to be completed within 45 minutes. All participants were familiar with the above usability principles.

Procedure Participants were introduced to the experimental design and given a demonstration of NetEdit, briefly describing its components and how they worked. The task was then explained and the users were sent to their workstations in different rooms. Each participant was assigned an experimenter, who observed his/her activities and noted any remarks said aloud. The participants were asked to speak aloud their activity/intentions while performing the task.

After about 45 minutes, participants were given a questionnaire that analyzed their experience with the system. Some responses in the questionnaire and any eccentric activity we observed were then discussed briefly with them.

Results The main results obtained from the evaluation are discussed below. They are organized into categories: use of the collaboration awareness, general editing facilities, social structures observed, and use of the chat facility.

Collaboration Awareness The participants started by playing around with the system. They liked radarview and also found it useful during the editing process. Initially when concentrating on their own evaluation, radarview was used often to determine what other participants were doing. Telecursor gained importance when the groups started combining their work together. One of the participants kept watching the telecursors of other users to find out what they were doing and infer their intentions.

We observed that participants spend some effort in determining who had written certain sections of a document before they could address comments to that person. When the participants wanted to communicate with each other about certain text written by someone, they had to first identify who wrote that text, and then talk about that text. The initial round of messages to identify the author added to the volume of messages being passed around without serving any useful purpose. This is an aspect of collaboration awareness that NetEdit did not support, and that users found ways to compensate for via the chat window. However (see below), this increased the dependency on the chat window and thus caused some frustration for the users. A possible solution would be to assign color to the text based on who inserted that text.

General Editing Facilities Three users complained that, whenever there were characters inserted or deleted from the part of the document above the point where a user is working, the user notices a sudden movement of his cursor location. This movement is especially significant when a newline character is inserted. This is because the entire line of text, along with his cursor, suddenly goes to a new line. When this happens, he looses the context of his surrounding text and gets confused.

This could be solved in two ways. One is a gradual change of position so that sudden movement is avoided. For example, in the cloud burst model [6] local operations appear in the window immediately but a cloud appears over remote operations. The text associated with these remote operations is then progressively revealed as the cloud starts fading.

Another possible solution is to modify the JScrollPane and JTextPane widgets that contain the document, so that they grow both ways instead of just in one direction (downwards) when characters are inserted. If characters are inserted above the local user's cursor position, the document would grow upwards, so that the current group of lines being displayed are not altered on the local machine. If the characters are inserted below the local users cursor position, the document would grow downwards, as it does now. This is an example of where a scrollbar, as implemented in Java, is appropriate for a single user, but fails when used in a multiple user application.

Social Structures Observed Two different styles of work were observed. The second-round group with three participants made sections in the edit window and each person worked on his own section. But they took some time to decide on this organization. They were allowed to communicate only through NetEdit's Chat window. The frustration of not being able to get themselves organized was clearly indicated by their messages in the chat window. They had initially started preparing the document as if they were using a single user editor. It took them a little while to realize that for the group activity to be effective, they needed to work differently. It clearly showed that after some learning of not only how to use a collaborative tool but also how to work remotely in groups, the session can be productive. It all came down to establishing a social protocol for editing the same document.

One user remarked that it was difficult during editing to have to continually switch back and forth between the chat and editing window. These observations were substantiated by the responses from the questionnaire and the brief discussion we had with the participants after the experiment. This had an interesting effect, discussed below.

Editing started getting chaotic when they began combining their evaluation into a single coherent document. Some participants tried to move their work closer to the work of other participants without realizing that these other participants were trying to do the same thing. Thus all of them were trying to move their text in between the text of other users. Although there were a lot of chat messages between them so that they could get synchronized, it seemed that communicating using the chat window was too slow. By the time a user typed some chat message and returned to the edit window, the state of the document had changed. Hence the chat comments were no longer valid. This caused a lot of irritation between participants. To avoid switching back and forth between the chat and edit windows, they started communicating using the edit window itself. The document now started to look confusing, as chat messages were inserted into their text.

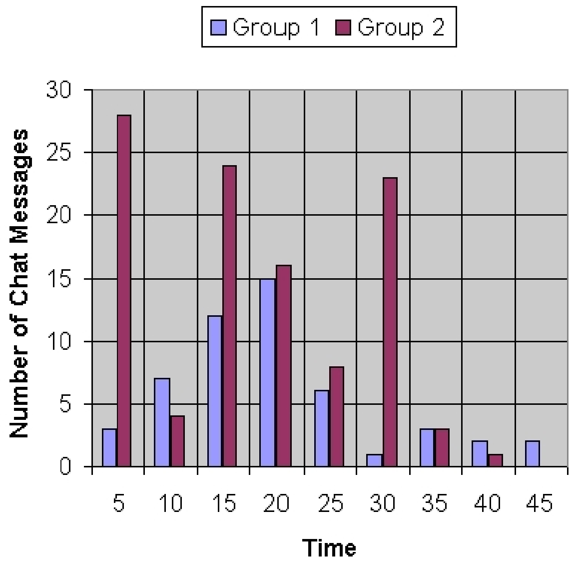

Figure 7: Distribution of chat messages over time.

We explicitly imposed a structure on the group with four users. One of the participants was instructed to act as a director overlooking the activities of all the participants and guide the preparation of the document. However, this group could not get oriented and focused. One of the participants in this group felt a need for the system to provide tools to automatically know the intentions of his group members. It seemed to us that since there were four participants (a bigger group than the others), identifying and synchronizing the activity was more difficult. The slow mechanism for communication (chat messages) exacerbated the problem.

Use of the Chat Facility Figure 7 shows the distribution for the number of chat messages against time during the experiment, for the two groups in the study. The time on the graph is divided in five-minute intervals. Thus Group 1 generated three messages during the first five minutes and seven messages during the next five minutes. Group 1 had three participants and Group 2 had four participants.

Group 1 began by investigating NetEdit, by exploring its widgets, going through different options on various windows etc. There were messages for deciding who would write what and how they would later combine their evaluation. As they proceeded with the activity, the number of chat messages kept increasing for the first 20 minutes. After this, they began to combine their work and, surprisingly, the chat activity started decreasing significantly. This is because the users were getting annoyed by continually having to switch back and forth between the chat window and the edit window. They preferred to communicate with each other through the edit window itself. That explains the low activity in the chat window during the latter 20 minutes of the experiment.

Group 2 (the one with four participants and an imposed structure) started by discussing with each other how they would get organized. This explains heavy activity in the chat window during the first five minutes. Then they went on to exploring NetEdit, its features, different windows etc. Even with this group, we see a dip in the number of messages during the latter part of the experiment since they again used the document itself for communication. This group was less organized. They would chat for extended periods, then shift to doing only editing, and then again come back to only chatting and so on. They strongly felt the irritation of switching between chat and edit windows.

There was a need for faster communication between participants. One solution to this problem is to have a chat window attached to the edit window, specific to each session. This would avoid having the users switch back and forth between typing some text in the edit window and typing messages in the chat window. Another option is to incorporate an audio channel for communication. Furthermore, our collaboration awareness tools need to be extended to include information about the author of different sections of the document, as this will eliminate one use of the chat window.

Future Work

This research brings to light many areas for future work.

Fault Tolerance NetEdit has a centralized architecture with the processing distributed between clients and the server. If the server fails, then the entire system must halt. There is no secondary server that could mirror all the data and operations, and dynamically replace the faulty server. Mechanisms for swapping in a new server must deal with the fact that, while the switch is taking place, there could be operations being performed by the clients.

Algorithm Extension The core algorithm, that manages consistency of the document at all the clients, now processes single characters. However, it is seen that multiple characters inserted or removed in succession, in a short period of time, go through similar transformations. Hence it will be interesting to modify this algorithm so that strings of arbitrary length, instead of single characters, can be transformed. This might potentially increase the speed, as a lesser number of operations now needs to be processed, and also reduce the memory space required, as a lesser number of operations would be needed to be stored in the buffers.

Improving Efficiency While implementing this prototype, we noticed two bottlenecks that affected the performance of the system. One is the bandwidth used for passing messages between clients. There are large numbers of small messages that are being broadcasted to the clients. These messages include operations being performed by the users, awareness information such as change in caret position etc. One might aggregate some of them and thus potentially improve the bandwidth utilization. However, one needs to explore the optimum point beyond which the response time becomes noticeable by the remote users. Another bottleneck is the slow display refresh mechanism of the Java virtual machine. Some of the widgets where this was noticeable were JTextPane (which holds the document being edited), and JScrollPane (which provides scrolling support).

Communication The editor is meant to be where the collaboration product is developed, not where the collaboration takes place. Our preliminary usability study of NetEdit revealed that the existing communication mechanism between participants through a chat window is uncomfortably slow. Various options to address this issue should be explored. For example one could provide an audio channel between participants in the same session and have a chat window for communicating with participants in other sessions, etc. These two modes are suggested to avoid the user from getting overwhelmed by audio messages.

Version Control Keeping track of different versions of the document and reverting to an earlier version can be an important utility. However, it is difficult to determine to which version one must roll back. Is it the version where only the operations done by the local user must be reverted or is it the version where both local and remote operations need to be undone? The correct answer depends on the type of group activity. For tightly coupled collaboration (such as brainstorming a paper where all the participants are working very closely), one might want to revert to a previous version undoing both local and remote operations. However, if the collaboration is loosely coupled, where the participants are working in different parts of the document, one might want to rollback to a previous version undoing only local operations.

Studying Group Mechanics Since the system does not assume or impose any protocol for group activity, the participants can form the editing structure that suits them. These structures could be decided explicitly at the beginning of the session or could be formed implicitly as the group activity proceeds.

REFERENCES

1. J. 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 6*, 2(June, 1999), 95-132.
2. J.M.A. Begole, C.A. Struble, C.A. Shaffer, and R.B. Smith, Transparent Sharing of Java Applets: A Replicated Approach, *Proceedings of ACM UIST'97*, October 1997, 55–64.
3. P. Dewan, Architectures for Collaborative Applications, In *CSCW, Trends in Software Series 7*, M. Beaudouin-Lafon (Ed.), 1999.
4. P. Dourish, and V. Bellotti, Awareness and Coordination in Shared Workspaces, *Proceedings of ACM CSCW'92*, November 1992, 107–114.
5. C.A. Ellis and S.J. Gibbs, Concurrency Control in Groupware Systems, *Proceedings of the ACM SIGMOD Conference on Management of Data*, May 1989, 399–407.
6. C.A. Ellis, S.J. Gibbs, and G.L. Rein, Groupware: Some Issues and Experiences, *Communications of the ACM 34* 1(January 1991), 38–58.
7. S. Greenberg and M. Roseman, Groupware Toolkits for Synchronous Work, In *CSCW, Trends in Software Series 7*, M. Beaudouin-Lafon (Ed.), 1999.
8. C. Gutwin, S. Greenberg, and M. Roseman, A usability study of awareness widgets in a shared workspace groupware system, *Proceedings of ACM CSCW'96*, November 1996, 258–267.
9. L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM 21*, 7(July 1978), 558–565.
10. Microsoft NetMeeting, April 2001. `http://www.microsoft.com/windows/netmeeting/default.asp`
11. D.A. Nichols, P. Curtis, M. Dixon, and J. Lamping, High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System, *Proceedings of ACM UIST'95*, November 1995, 111–120
12. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhauser, An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors, *Proceedings of ACM CSCW'96*, November 1996, 288–297.
13. C. Sun and C.A Ellis, Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements, *Proceedings of CSCW'98*, November 1998, 59–68.
14. 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 5*, 1(March 1998), 63–108,
15. Y. Yang, C. Sun, Y. Zhang, and X. Jia, Real-Time Cooperative Editing on the Internet, *IEEE Internet Computing 4*, 3(May-June 2000), 18–25.