Network Text Editor (NTE)
A scalable shared text editor for the MBone

Abstract

IP Multicast, Lightweight Sessions and Application Level Framing provide guidelines by which multimedia conferencing tools can be designed, but they do not provide specific solutions. In this paper, we use these philosophies to guide the design of a multicast based shared editor, and examine the consequences of taking a loose consistency approach to achieve good performance in the face of network failures and losses.

1 Introduction

The IP multicast[1] service model and the philosophy of Application Level Framing[2] together hint at new ways in which distributed applications may be developed in many areas of networking. They do not, however, give detailed guidance for application developers, and there is a great deal of relatively unexplored territory, with just a few markers where people have successfully been before. With this in mind, we set out to see where IP multicast and Application Level Framing (ALF) might lead the development of a shared editor for the MBone.

The many-to-many model of IP multicast lends itself readily to distributed data applications, where everyone holds all the data and multicasts changes to this data-set to the other participants. ALF makes this approach powerful and ensures that it can perform well because only the application has sufficient context to cope with the consistency and reliability problems that can occur in a sufficiently flexible manner.

These philosophies can also lead to conflicting design goals. In this section, we discuss these goals, and the influence of multicast and ALF. In section 2 we present the design we devised to overcome these conflicts, and in section 3 we speculate on how some of these mechanisms might be generalized for other applications.

1.1 Conflicting Goals

When designing a distributed data shared application such as a shared editor, the following goals should be satisfied by the dataset distribution mechanism:

- Many users should be able to *manipulate the same data object* over time.
- *Eventual consistency* - the dataset should converge on one dataset after a change (though it may be temporarily inconsistent while changes are propagating due to loss or network failures).
- *Deterministic behaviour* - if a user is allowed to modify a data object, they expect it to stay modified.
- *Fully interactive* - users usually don't want to have to wait for locks to be granted to be able to manipulate a data object, as this leads to indeterministic delays due to loss or failures.

Unfortunately these goals are contradictory in a typical internet environment with unpredictable loss and failures.

Different tools choose not to satisfy one of these goals.

- LBL's shared whiteboard, wb[3], is the only multicast based distributed data application in widescale use. It does not allow different users to be able to manipulate the same object.
- Dissemination applications such as multicasting web pages[5] or Usenet news feeds[6] only have one data source per group, and so avoid this conflict.
- Traditional distributed applications make use of locking, and suffer performance problems as a result.
- Relaxing eventual consistency is not an option.

Thus if we are to achieve performance and allow different users to modify the same object over time, relaxing deterministic behaviour seems to be worth exploring. By this we mean that under some circumstances (that we can aim to occur rarely by careful design), the system asserts a change that wasn't what some subset of the users expected. The hypothesis is that we can make this happen rarely enough and

provide sufficient feedback to the users when it does occur that it will not cause significant problems.

Simply relaxing determinism does not necessarily result in usable applications, and there are additional mechanisms and constraints that are required before the problems caused by this design choice can be said to have been overcome. In particular, achieving global consistency with such a distributed data application is not trivial. We will explore these problems in section 2 and propose solutions that aim to minimize their frequency of occurrence and gracefully deal with them when they do occur.

1.2 Philosophy and Background

IP multicast provides a service model by which a group of senders and receivers can exchange data without the senders needing to know who the receivers are, or the receivers needing to know a-priori who the senders are. In addition to providing efficient data distribution, this service model can lead to scalable and robust applications because the members of the group do not need come to any agreement about who is actually in the group. This has led to the evolution of the scalable *lightweight sessions*[7] model for multimedia conferencing, where there is no explicit group membership mechanism other than joining the multicast groups and sending period session messages to indicate membership.

When it comes to designing shared applications to co-exist with these lightweight sessions, we would like similar scaling and membership properties - members of the group should be able to come and go with minimal impact on the other group members.

IP multicast provides a transport mechanism that is unreliable and non-ordered. We could attempt to provide an end-to-end reliable transport layer over IP multicast for the application to use, but to do so requires abandoning the lightweight sessions model because such a general purpose transport layer requires explicit membership to achieve reliability.

The concept of *Application Level Framing*[2] emerged due to a realization that applications of this sort did not want a single transport layer to perform everything for the application. The ALF philosophy suggests that data should be transmitted on the network in units which are idempotent and thus can be utilized by the application independently of other application data units. This is exactly what is required for multicast based shared applications, so that loose consistency may be maintained, and data can be presented to the user as soon as it is available. The application can then handle reliability and consistency issues as it sees appropriate depending on the application context.

The ALF philosophy applies to time as well as to space. In an ALF application, data need not arrive in a strict order to be meaningful. This allows the application a large amount of leeway in deciding how to provide reliability and consistency. Clearly in a shared application, changes should be sent as soon as possible to provide immediate feedback to the users, but network failures and packet loss will ensure that receivers end up with heterogeneous state. Such inconsistencies can be resolved by retransmission, but can also be resolved using forward error correction techniques, which can be much more flexible and appropriate with ALF.

In the current MBone, significant numbers of points in a distribution tree generate small amounts of loss, and these result in low probabilities that a single packet reaches all the receivers in many situations. In their experiments with 12 sites, Yajnik[8] finds that retransmission would have been necessary for between 38% and 72% of packets, and this would have been the case for around 95% of packets sent to the much larger multicast video sessions observed by Anon[9] if these had required a reliable multicast protocol. Thus the use of some form of redundancy would appear to be very desirable in designing reliable multicast applications.

1.3 Related Work

Most similar work is in the areas of CSCW and replicated databases. In general, the majority of CSCW work has centered around preventing inconsistencies from arising. We believe that temporary inconsistencies are necessary to achieve good performance. Some work does explicitly allow inconsistencies, but most of this work is in the area of so-called "asynchronous" collaboration. Haake and Haake [10] use a versioning system to manage parallel versions, but do not concentrate on subsequent re-synchronisation. Munson and Dewan[11] focus on object merging, but their techniques are applicable only to explicit complete merges in an asynchronous environment rather than the "synchronous" environment we are considering where divergence is usually unforeseen, and merges need to be timely, partial and opportunistic.

Techniques from the area of operational transformation[12] are more applicable to synchronous systems. Such schemes use a model of multiple streams and use a transformation matrix to transform models of remote changes before applying them locally based on the context in which the change occurred. This is effectively a more general approach than we take, but in its generality, it fails to resolve trivial conflicts where one of two changes can be automatically chosen so long as the users are made aware of the the issue, even though this loses information. Thus we believe that (within limits) unexpected changes due to conflict resolution are not substantially different than unexpected changes that cause no conflict.

Distributed databases typically support a transaction model, whereby changes are represented as a sequence of transactions. In replicated databases, the focus is on detection of transaction conflicts, and on finding an execution order that avoids potential conflicts. Such techniques are at odds with the simple use of redundancy that we will propose, and although techniques which allow rollback of conflicting transactions in the face of a conflict might be applicable in a shared editor, we do not believe they add significant useful functionality over rollback techniques that can be performed without such a rigid audit trail.

In general, although there is a significant body of work in the areas of merging inconsistent data sets, no lightweight shared application em protocols have so far emerged. The closest work in the internet community is the SRM[3] work from LBNL, and this takes the more restrictive approach of preventing inconsistencies from arising.

2 Design

To achieve resilience, we adopt a distributed, replicated data model, with every participant holding a copy of the entire document being shared. End-systems or links can then fail, but the remaining communicating sites[1] still have sufficient data to continue if desired. *NTE* uses IP multicast to provide unreliable many-to-many communication at near constant cost to the application, irrespective of the number of receivers. Reliability mechanisms are then the responsibility of the application alone.

2.1 Application Data Units

NTE's data model is determined by interactivity requirements - many users must be able to work on the same document simultaneously - and by the observation of usage modes, particularly the need to be able to keep annotations separate from the primary text being worked on.

The data model is hierarchical, based around blocks of text, each consisting of a number of lines of text. Each block is independent of other blocks - it can overlap them if required although this does not aid readability. An example of blocks used for annotation, is given in figure 1.

Most annotations will not be modified by multiple users simultaneously, and this allows a number of users to be working simultaneously on the document in separate blocks. However, restricting users to simultaneous *annotation* of documents would impose too great a constraint on potential usage modes, and so each line of text is also separate entity, allowing users to be working on separate lines in the same block.

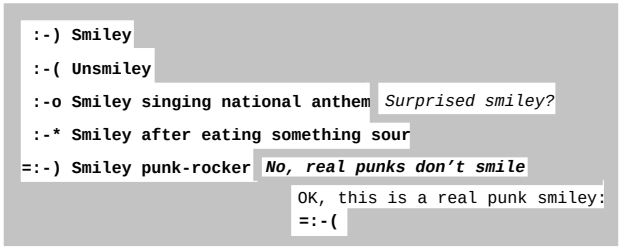


Figure 1: An example of blocks of text used for annotation

Taking this model further, and treating each character of a line as independent is undesirable. Firstly, the amount of state that needs to be kept for each separate entity to ensure eventual consistency is significant. In addition, line ADUs have the advantage that it is unnecessary to receive all the individual changes to the line as a user types - the most recent version of the line is sufficient, giving a large degree of redundancy in the face of packet loss. Lastly, there are transmission failure modes with either line or character ADUs that render no globally consistent ordering for the data. Due to the nature of text editing, these are significantly less likely to occur with lines than with characters. We discuss this and also the implications of simultaneous modification later, in the light of the loose consistency model described below.

When a line is transmitted, it carries the id's of the previous and next lines and the id of the block it forms a part of. Although lines and blocks are not completely independent, blocks can be moved without modifying the lines contained in the block, and lines can be created, deleted and edited independently of other lines or blocks. There are however a number of desirable operations on lines that cannot be carried out independently, and we shall discuss these and their consequences in section 2.6.

2.2 Distributing the data model

The choice of a line as the ADU was made in part due to the simple observation that most consecutive changes are made to a single line - generally because a user continues to type. Thus if the whole line is sent for every character typed the additional overhead (over just sending the changes) is not great, the data transfer is idempotent (assuming a version of the block has already been received), and a great deal of "natural" redundancy is available - so long as a user continues typing on the same line, lost packets are unimportant. However this is only the case if we take a loose consistency approach - changes are displayed as soon as the arrive, irrespective of whether other sites have received them.

[1] In this paper we use the term *site* to indicate a single instance of the application, wherever it is located

In order to be able to use this "natural redundancy" property, it must be possible to identify whether a version of a line that just arrived is more recent than the copy of it we have already. This is necessary to cope with misordered packets from a single source, and to cope with retransmitted information from hosts with out of date versions of the data.

If we assume synchronized clocks at all sites, recognizing out of date information is achieved by simply timestamping every object with the time of its last modification. Copies of objects with out of date timestamps can be ignored at a receiver with a later version of the same data. If we wish to take advantage of redundancy by not requiring retransmission of many lost packets, a receiver must not care if it receives all the changes to a object as they happen; rather it only needs to receive the final version of an object, although receiving changes as they happen is desirable.

In practice, we can't assume synchronized clocks, but we can implement our own clock synchronisation protocol in the application.

There are alternatives to this mechanism, including maintaining a change log with each object, but they do not help greatly. Either they require locking, or they suffer from the same merging of changes problem that timestamping suffers from without significantly helping solve the problem.

2.3 Clock Synchronisation

Given that all changes to a document are multicast, and that all changes are timestamped, we have a simple mechanism for clock synchronisation amongst the members of a group:

- if a site has not sent any data and receives data from another site, it sets its application clock to the timestamp in the received message.
- if a site has not received any data, and needs to send data, it sets its application clock to its own local clock time.
- if a site receives a message with a timestamp greater than its current application clock time, it increases its application clock time to match that of the received message.

These rules ensure that all sites' application clocks are synchronized sufficiently accurately for our purposes.

Figure 2 illustrates this process: source S2 sends the first message, and S1 and S3 synchronize to the timestamp in the message. Neither S1 nor S3 had set their clocks before this point. Two new sources (S4 and S5) then join, and before any of the original sources send a message, S5 does so. As S4 has sent no message (therefore has no data), it now synchronizes

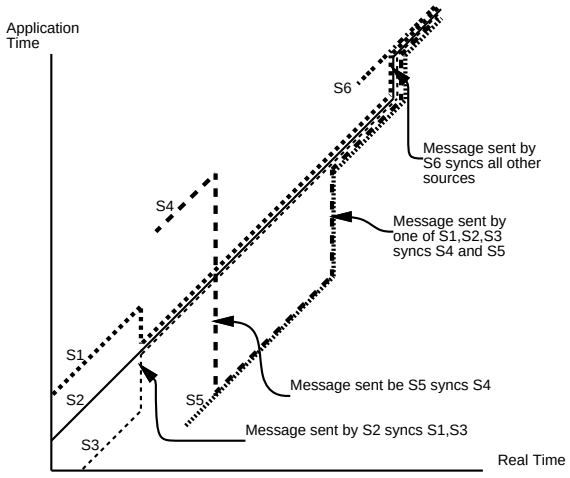


Figure 2: Application based clock synchronisation

to S5. The three original sources have data therefore do not synchronize to S5. One of the three original sources then sends, and both S4 and S5 synchronize to the timestamp in the message.

To show this achieves the desired results, consider three sites A, B, & C, with three application clocks t_A, t_B, and t_C, and positive transmission delays d_{AB}, d_{AC}, etc.

if A sends the first message, we have
$t_B = t_A - d_{AB}$
$t_C = t_A - d_{AC}$

if d_{AB} ever decreases when A sends, then B will increase its clock to match the new delay, and t_A and t_B become closer.

if d_{AB} increases, B continues to use t_B

Now consider a message sent k seconds later by C:

This message arrives at B with timestamp $k + t_A - d_{AC}$ and it arrives at time $k + (t_A - d_{AB}) + d_{CB}$. A comparison is made and only if $d_{AC} < d_{AB} - d_{CB}$ is the clock at B increased to be $k + t_A - d_{AC}$. Thus the clock at B can only get closer to the clock at A when a message is received from C.

The process continues so long as messages are sent.

As all messages are timestamped, clock synchronisation to less than the minimum delay between each pair of active sites is provided for free, and no explicit clock synchronisation protocol is required. This assumes that all local clocks run at the same rate.

This is a reasonable first approximation for almost all of today's workstations, within the bounds that are detectable by human reaction time. Should a clock drift by a few seconds, then it is possible that a change made at one site may be reversed from another site within the bounds of the clock drift.

However in practice this does not happen because the sites need to exchange data to create such an event, and this data exchange causes a clock re-synchronization to the fastest clock.

There are algorithms that synchronize the clocks much more accurately than this, but for the purposes of consistency control, a necessary feature is that clocks are never decreased, and the algorithm given is simple and sufficient.

Implementation of this algorithm reveals that there is a case where clocks do not stabilize. This occurs when two sites with a clock tick of length t are connected by a network with a transit delay of less than t, as illustrated in figure 3. This can happen with some Unix workstations with a 20ms clock resolution connected by a local ethernet. Under these circumstances, the receiver will synchronize to the sender to a resolution of less than t. If the two clocks are not in phase, then the receiver can be ahead of the sender for part of each clock cycle. If their roles as sender and receiver are reversed and the new sender now sends a packet at a point in the clock cycle where its clock is ahead, the old sender then increments its clock to match the new sender. If both sites send alternately, this can result in both clocks being incremented indefinitely. This can simply be avoided if the clock tick interval is known, by simply ignoring clock differences of less than the clock tick interval.

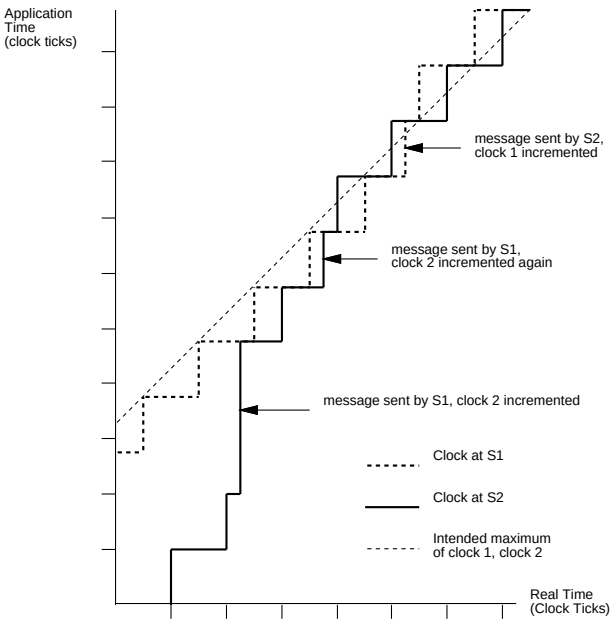


Figure 3: Clock synchronisation failure due to clock granularity being greater than transmission delay

2.4 Reliability Mechanisms

Due to the redundancy inherent in the data distribution model, NTE will sometimes perform reasonably well with no mechanism for ensuring reliability. However, there are also many situations where this is not the case, and so we need a mechanism to detect and repair the resulting inconsistencies.

Inconsistencies may result from:

- Simple packet loss not corrected by subsequent changes (particularly where the last change to a line has been lost, or where data was loaded from a file)
- Temporary (possibly bi-directional) loss of large numbers of modifications due to network partition.
- Late joining of a conference.
- Effectively simultaneous changes to the same object.

2.4.1 Inconsistency Discovery

Unlike the mechanisms used SRM [3] and INRIA's whiteboard [14], inconsistencies due to simple packet loss cannot be discovered simply from the absence of a packet as we wish most such changes to be repaired by redundancy, and therefore do not need to see every packet at a receiver.

Instead we use a mechanism that ensures inconsistencies are resolved, irrespective of the number of packets lost. There are three parts to this inconsistency discovery scheme.

Two mechanisms are based on the session messages each instance of the application sends periodically to indicate conference membership. These session messages are sent by each site with a rate that is dependent on the total number of sites in the conference, so that the total session message rate is constant and low. To detect inconsistencies, each session message carries two extra pieces of information - the timestamp of the most recent modification seen, and a checksum of all the data. If the timestamp given by another site is later than the latest change a receiver has seen, the receiver can request all changes from the missing interval without knowing what that data actually was. This may not fill in sufficient information to ensure consistency, and so the checksum is a last resort to discover that a problem has occurred. This is followed by an exchange of checksums to discover which blocks the differences are in, and then a summary of the line timestamps in the inconsistent block.

The third mechanism is designed to prevent the above mechanisms from needing to be used where possible. We have a concept of the current site - this is the site which has most recently been active. If more than one site is active, any of those sites can be chosen as current site. The current site periodically multicasts out a summary packet giving the timestamps and IDs of all the most recently changed objects. If a receiver has a different version of one of these objects then it is entitled to either request the newer version from the current site, or to send its newer version.

The current site may change at the end of each retransmission round (see section 2.5.1) each time a new user modifies the document; however the rate that these summary packets are sent is a constant whilst any users are modifying the document - a new current site simply takes over from the previous one. If two sites both think they are the current site, the one with the lowest IP address stops sending. Once a document becomes quiet, the rate of sending summary packets is backed off exponentially to a low constant rate.

An alternative to sending explicit summary packets might be for session and data packets to have an additional object ID and its modification timestamp added to them, and for all sites to take turns to report the state of the most recently modified objects. Indeed, this may be preferable for some applications, but because of the retransmission scheme chosen and the dynamics of text editing, we have not used such a mechanism.

2.5 Scalable Retransmissions

When a receiver discovers there is an inconsistency between its data and that of another site, it cannot just send a message to resolve the inconsistency immediately because there is a high probability that its message would be synchronized with identical messages from other receivers and cause a NACK implosion. Instead we require a mechanism to ensure that approximately one site sends the message. If this message is an update, it should be multicast as this may update other sites with out of date information. If this a message is a retransmission request, it should also be multicast, as then the reception of the request can be used at other sites to suppress their retransmission requests.

SRM[3] uses a mechanism by which retransmission requests are delayed by a random period of time partially dependent on the round-trip time between the receiver and the original source. To work most effectively, this requires all participants to calculate a delay (round trip time) matrix to all other sites, and this is done using timestamps in the session messages.

As it has no redundancy mechanism, SRM is more dependent on its retransmission mechanism than NTE is, and thus it requires its retransmission scheme to be extremely timely. NTE does not wish its retransmission scheme to be so timely, as it expects most of its loss to be repaired by the next few characters typed. This results in very significantly fewer packet exchanges because in a large conference on the current MBone, the probability of at least one receiver losing a particular packet can be very high. Thus what we require is a retransmission scheme that ensures that genuine inconsistencies are resolved in a bounded length of time, but that temporary inconsistencies due to loss which will be repaired anyway do not often trigger the retransmission scheme.

SRM can be tailored for redundancy by adding a “dead time” to the retransmission timer to allow a window during which the next change could arrive. If we used SRMs randomized time based scheme, then we would probably opt for not sending summary messages but instead adding ID/timestamp pairs to the session messages as described above. These would then tend to spread the retransmission requests more evenly.

At the time NTE was designed and implemented, SRMs mechanism has not been described in detail, and we used a different sender driven retransmission request scheme. For most purposes we believe SRM is superior, but there are uses for which NTE's sender-controlled scheme is desirable.

2.5.1 Sliding Key Triggered Retransmissions

When a instance of NTE sends a summary packet, it starts upon the process of sending a sequence of keys. When a receiver matches a key sent by the sender, it can immediately send its retransmission request (which can be many objects if necessary) along with the key that was matched. On receiving this request, the sender then starts the retransmission of the missing data.

The sender generates a random integer (key) when it creates its summary message. Upon receipt of the summary message, the receiver also generates a random key. Then the sender sends its key along with a key mask which indicates the bits in the sender's key that must be matched in order for the receiver to send a retransmission request. This key/mask pair is sent several times, and if no retransmission request is forthcoming, the bits indicated by the mask are reduced by one, and the key/new-mask pair is sent again. If no retransmission request is forthcoming by the time the mask indicates no bits need to be matched, then the process is started again with a new random key, a new summary report, and possibly a new current site. If no change has occurred since the previous summary report, the rate of sending sliding keys is reduced to half the rate for the previous round until it reaches a preset lower rate limit. This process is illustrated in figure 5.

This is loosely based on a scheme[13] devised by Wakeman for congestion control in multicast based adaptive video.

As the session messages give a reasonable approximation of the size of the conference at the point when we generate the summary message, the sliding key can be started close to the point where it would be expected to elicit the first response if all receivers need a retransmission. The delay then before receiving a retransmission request scales $O(log(n))$ where n is the number of participants. This is shown in figure 4. For

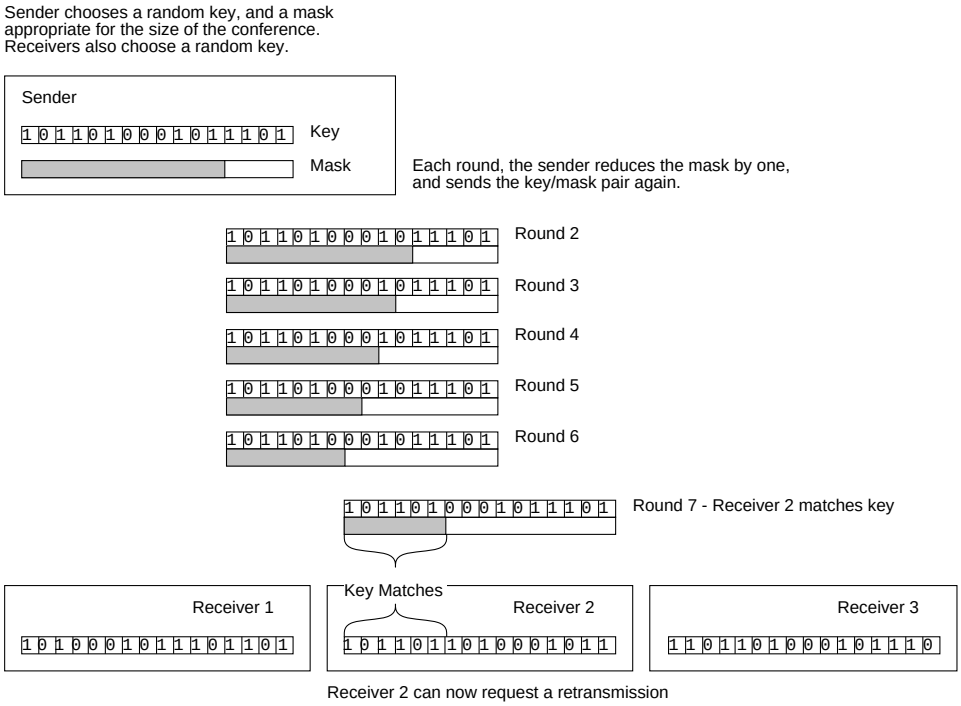


Figure 5: Sliding Key Triggered Retransmission Requests

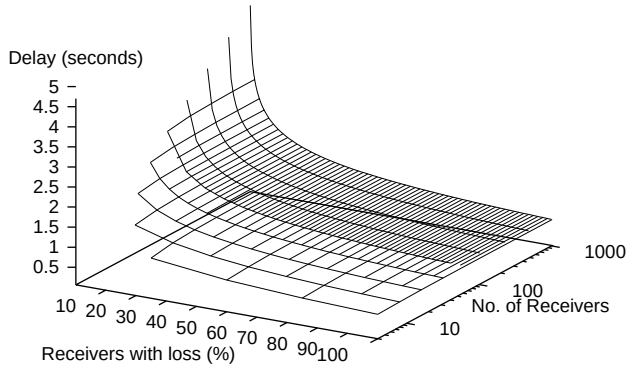


Figure 4: Expected delay before a retransmission request (RTT=250ms)

a typical 1000 way conference, where only one receiver requires a retransmission, with each key/mask pair sent twice per round and an estimated worst case RTT of 250ms, this results in 4 (small) packets per second and a maximum delay of 5 seconds before requesting retransmission. If the conference was smaller, or more sites had suffered loss, this time would be reduced.

2.6 Limitations of the Data Model

We have described a data model and a distribution model which are oriented towards building a scalable distributed shared text editor. However, these models impose a set of limitations on the functionality of the shared editor, or on the way this functionality is implemented.

Whilst a data model based on blocks and lines allows different blocks or lines to be modified simultaneously without any problem, the lock-free distribution model and the choice of a line as ADU mean that it is possible for more two or more users to attempt to modify the same line effectively simultaneously.

In addition, network partitions can result in more complex inconsistencies arising. We show below that careful design choices can result in all such more complex inconsistencies becoming equivalent to a *detectable* case of effectively simultaneous insertion of lines into the same place, and that this can be resolved.

Deletion of Lines of Text

Deleted lines need to be maintained in the data structures, with a marker that they are no longer visible. If they were not stored as actual objects, then they may be re-asserted by sites which have missed seeing the deletion event. Deleted line ID's and deletion times must be transmitted to and stored

at all sites (including sites joining after the deletion event) to prevent unintentional re-assertion after sites that saw the deletion have left the conference.

If an entire block of text has been deleted, sites only need hold the ID and deletion time of the block - the holding of information about lines within the block is unnecessary.

The most significant design decision here concerns the effects of network partitioning - if a user at a site in one partition deletes a line, and subsequently another user in the other partition modifies the line, we have two choices when the partition is resolved:

- Rely only on modification time (i.e. treat deletion as just another modification that can be changed later)
- Make deletion irreversible so that deletion always prevails over modification.

Treating deletion as modification leads to a number of scenarios where it is difficult to achieve eventual consistency.

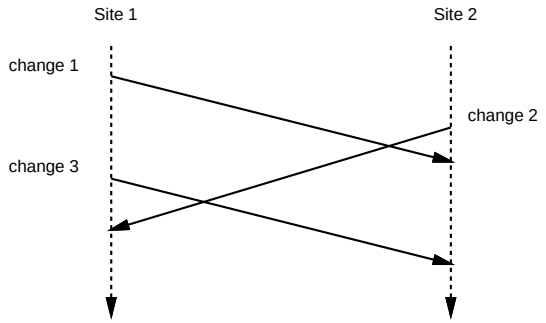
We took the decision that deletion is irreversible, as this provides a mechanism that is irrespective of causal ordering in a partitioned network, and this causal ordering independence provides a mechanism which we can utilize to achieve eventual global consistency.

To achieve inconsistency detection, there is one further requirement; deleted lines must not only be kept at all sites to prevent re-assertion of the line from other sites, but they must also be kept in their original place in the list of lines at each site. For example, a line preceding a deleted line must still be transmitted with its "next line" field indicating the deleted line. This ensures simultaneous insertion is a detectable situation.

Effectively Simultaneous[2] Changes to the Same Line

NTE is designed for use in multimedia conferences where there are additional channels of communication between users. Thus two users attempting to modify the same line will normally see the change made by the later site being substituted for the change made by the earlier site. If such changes do not involve adding a line break to the middle of a line, then no lasting confusion should remain, and the loser will be notified of what is happening. Typically, at this point the users will talk to each other and decide who should actually make the change.

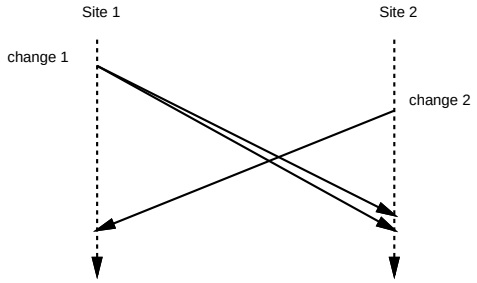
It should be noted that although there are circumstances when one site does not even see the *change* made by the other site (for example, Site 1 does not see Change 2 in figure 6), the user interface should signal that another user is attempting to modify the line, so that both users do realize exactly what is happening.



Changes 1 and 3 prevail. Change 2 is seen briefly at site 2, then disappears.
Site 1 never sees Change 2 - it is already too old when it arrives at site 1.

Figure 6: Two users attempting to simultaneously modify the same line

If one of the users adds a line break to the middle of a line, this could be mapped into a truncation of the existing line, and the creation of a new line after the existing line with the remainder of the text. The overlapping request, if it happens after the line break insertion, but before the receipt of the message (see figure 7) will then re-assert the previous line. This will be confusing for users, as suddenly text has been duplicated.



change 1 results in a line being split in two, creating a modified line and a new line.
change 2 modifies the original line.
In a simple scheme, the new line from change 1 and the modfied original from change 2 remain. This is most likely with a network partition, but could also occur due to simple transmission delays.

Figure 7: Undesirable behaviour due to simultaneously splitting a line and modifying it.

However, if deletion is an irreversible operation (i.e., deletion overrides modification), then deleting the line to be split and inserting two new lines in its place prevents this undesirable behaviour at the expense of additional state. These deletion and insertion operations can be atomic. We shall show later that simultaneous insertion of lines is also detectable.

In the case of simultaneous modification due to transmission delay it is easy to inform users of the problem, but we can-

[2] By "effectively simultaneous" we mean changes to the same line where due to network delay, or partitioning, two users attempt to change the same line without seeing the effects of the others changes first. Thus the timestamps of the changes may not be identical.

not do so in the case of simultaneous modification due to network partitioning.

If only single lines have been modified, there is no real problem, as the later change will be asserted, and although this may not be what the user whose change just got replaced actually wants, at least the document ends up in a consistent state.

It is possible to keep a local copy of the state of any line we have modified at the time we last modified it in order to be able to re-assert the state if the conflict resolution was not what we actually wanted. However, it is easy to envisage ways to defeat such a scheme if it were to be performed automatically, so such a re-assertion should only be performed with explicit consent from the user for each step, and should not be the primary consistency mechanism.

Moving of sections of text

There are two ways to view the moving of a chunk of text:

- Deletion and then insertion of the contents of the deleted text
- Changing the bordering lines neighbours (sending modifications of the top and bottom lines of text specifying new neighbours)

Deletion and then re-insertion is wasteful of bandwidth and causes extra state to have to be held (and sent) for the deleted lines. We are sure however that no-one can have already changed the new lines, and so the original lines cannot be re-asserted.

Modifying the context information of the bordering lines and just re-sending this information has the possible advantage that someone modifying the moved text in its old position due to temporary partitioning sees her changes reflected in the new positioning after the partitioning has been resolved. However, it is possible that one or more of the bordering lines are modified in a partitioned site after the move has occurred, and that the resolution of the partition then undoes or partially undoes the move, resulting in a block where no-one knows the correct line ordering.

Allowing the possible situation of no-one knowing the correct line ordering is extremely undesirable, and so we treat moving as deletion and subsequent re-insertion, despite the additional overheads this entails.

Garbage Collection and Check-pointing

Irreversible deletion allows for check-pointing of blocks and for garbage collection of deleted lines by deleting a block, and then re-asserting the same data as a new block and new lines. For the additional cost of keeping a deleted block, we no longer need to store the deleted lines that used to be in the block. In addition, it means that in any subsequently resolved network partitioning, changes made in the other partition will not be asserted. This has to be implemented carefully, as the users in the other partition may not wish to see their changes simply discarded, but there are times when it is desirable to assert the current state. If both partitions check-point the same block, then the block will be duplicated at the time of partition resolution, which then allows user intervention.

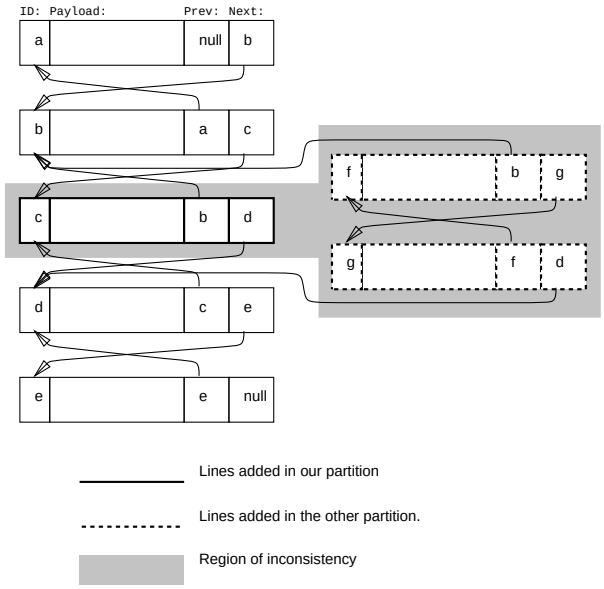


Figure 8: Simultaneous Insertion is a Detectable Situation

Effectively Simultaneous Insertion of Lines

Effectively simultaneous modification of a single line always results in a block that is internally consistent and the document in a globally consistent state. We've shown that irreversible deletion allows combinations of moving, deletion, line splitting and so forth to be performed, with the side-effect that some circumstances can result in effectively simultaneous insertion of lines.

Effectively simultaneous insertion of lines not only results in a document that is not internally consistent (we have two copies of essentially the same lines) but also a document is not globally consistent either (different sites can have different views of the document). This occurs because two or more lines each have the same neighbouring lines. It can be detected at all sites, because a set of lines is received that should fit in the same place in a block as a set of lines we already have.

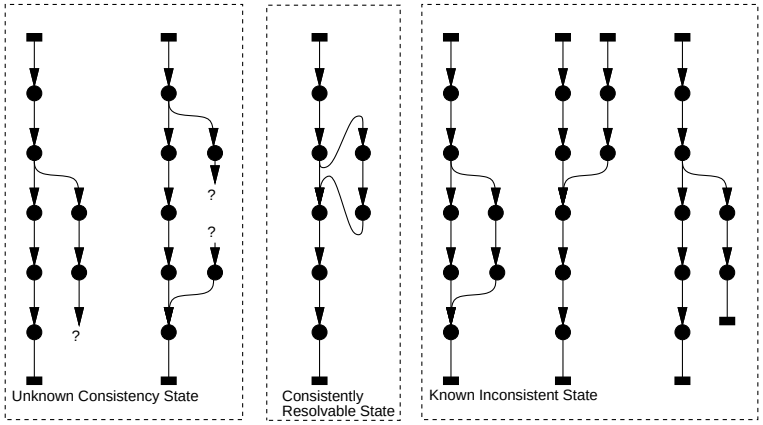


Figure 9: Known consistency, unknown state, and known inconsistency

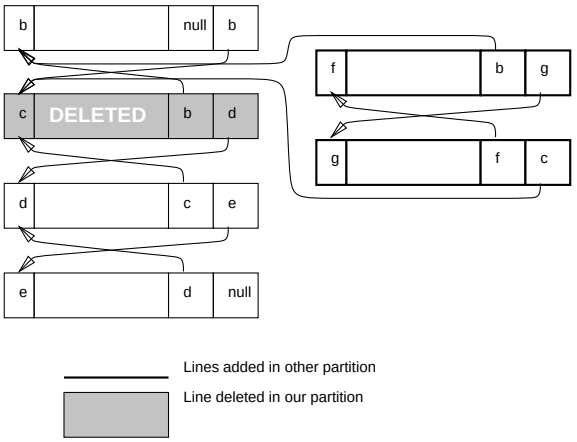


Figure 10: Simultaneous Deletion and Insertion - no inconsistency

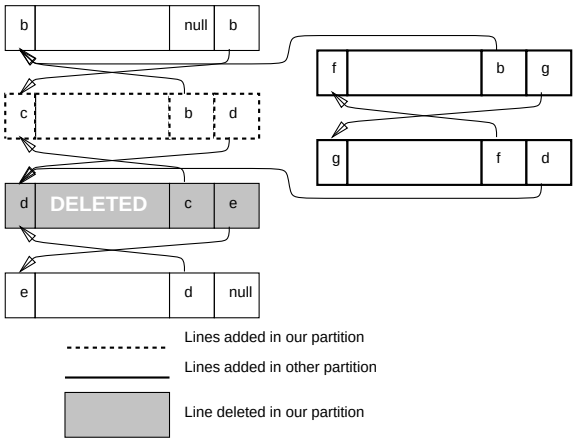


Figure 11: Simultaneous Deletion and Insertion resulting in inconsistency

When a modified line arrives, we may have seen the line it says is its previous line, the line it says is its following line, neither or both. It also may be the start or end of a block.

Thus we may temporarily not know whether we have inconsistencies such as those shown in figure 9. If we have unknown consistency, we should not display the received line until we reach a state of known consistency or known inconsistency. Figure 8 shows in more detail detectable inconsistency caused by simultaneous insertion.

Figures 10 and 11 illustrate why deleted lines must be kept in place in the data structures to make inconsistency detectable.

As a result of these rules, inconsistencies such as shown figure 8 and 11 can only occur when alternative new lines were inserted effectively simultaneously. Thus the question comes down to which of the two sets of lines to keep and which to delete. We must also ensure that the block is globally consistent *including the position of the deleted lines* or we may not be able to detect further inconsistencies at all sites. However, we do not care about the order within a set of consecutive deleted lines, so instead of deleting one of the two alternative sets of lines, we must actually delete both alternative sets of lines and re-insert one of them again before global consistency can be restored to the block.

Although the decision to perform the deletion can be made locally, the decision as to which alternative to choose cannot be easily be made consistently and locally in a manner that is likely to choose the better option. We could use arbitrary criteria based upon the data itself (but not on its modification time), although this seems undesirable.

The choice of which alternative set of lines to retain can be made automatically (by the “current site”) or it can be made clear to the users that there is a problem that requires resolution. Centralizing the decision in this way allows easier selection of policies based on the duration of partition, size of changes, etc. In a short partition with few changes, and au-

tomatic solution is preferable. In a long partition, it is more likely that a use should be involved. Either way, if more than one site attempts to resolve the conflict, the cycle is simply repeated again.

To illustrate this process, assume a consistent dataset comprising doubly linked list
$L_0 = (l_1, \ldots, l_m)$,
and that a partition then occurs. After some changes, two new consistent states L_A and L_B result:

$$L_A = (l_1, \ldots, l_n, a_1, \ldots, a_p, l_{n+1}, \ldots, l_m)$$
$$L_B = (l_1, \ldots, l_n, b_1, \ldots, b_q, l_{n+1}, \ldots, l_m)$$

At partition resolution, this is a detectable inconsistency as there are two paths from l_n to l_{n+1}. The inconsistency is resolved by deletion of the loop with consecutive deleted objects forming a single set-element, and re-insertion of one of the alternatives:

$$L_C = (l_1, \ldots, l_n, \{a_1, \ldots, a_p, b_1, \ldots, b_q\}, c_1, \ldots, c_r, l_{n+1}, \ldots, l_m)$$

where $(c_1, \ldots, c_r) \in \{(a_1, \ldots, a_p), (b_1, \ldots, b_q)\}$

Should this mistakenly be performed by more than one site, the process will be repeated as the new inconsistency is detected.

Any additional partitions that may have separated from L_A or L_B during the original partition can also be re-merged. For example, L_D may have partitioned from L_A during the original partition:

$$L_D = (l_1, \ldots, l_n, a_1, \ldots, a_x, d_1, \ldots, d_s, a_{x+y}, \ldots, a_p, l_{n+1}, \ldots, l_m)$$

where $1 \leq x \leq (x + y) \leq p$

On re-merging, the resultant dataset becomes:

$$L_E = (l_1, \ldots, l_n, \{a_1, \ldots, a_p, b_1, \ldots, b_q, c_1, \ldots, c_r, d_1, \ldots, d_s\}, e_1, \ldots, e_t, l_{n+1}, \ldots, l_m)$$

where

$$(e_1, \ldots, e_t) \in \{(a_1, \ldots, a_p), (b_1, \ldots, b_q), (a_1, \ldots, a_x, d_1, \ldots, d_s, a_{x+y}, \ldots, a_p)\}$$

In practise, it is rare that this mechanism is required.

Inconsistency Avoidance Mechanisms

To summarize the limitations necessary to ensure eventual consistency after the resolution network partitioning:

- Deletion must always override modification, irrespective of the timing of the two operations.
- Deleted items must be transmitted to and stored at all sites to prevent re-assertion.
- Deleted lines must be kept in their original place in a block, and must remain referenced by their neighbouring lines - this is a precondition for simultaneous insertion detection.
- Moving of text must be performed by deleting all the original lines containing the text to be moved and inserting the same text as new lines, thus preserving the original line ordering.
- If all the above are performed, line ordering within a block is only changed by adding new lines. This makes simultaneous insertion of lines during a network partition a detectable and resolvable situation.

These restrictions will ensure that eventual consistency is achievable, even in the face of continuing modification during a network partitioning. However, they are not always sufficient to ensure that the contents of the document eventually converge on what all of the users actually wish it to be. Although this could be achieved by locking of blocks to ensure that only one person can modify the a block at a time, we believe this restriction is often unnecessary and would restrict usage of the editor. Indeed, under many circumstances, the usage patterns of the editor are likely to be such that large scale simultaneous editing of a block during a network partitioning will not happen because the vocal discussion needed to do so will not be possible. If users are concerned about simultaneous editing of a block during a network partition, they should checkpoint the block to ensure no unseen changes can be made to it. For paranoid users, this check-pointing procedure could be automated, although we do not believe this is normally desirable or necessary.

3 Generalizing the Models

NTE, its data model, and its underlying protocol were all designed to solve one specific task - that of shared text editing. We used general design principles - those of IP multicast, light-weight sessions, and application level framing as starting points. However, the application data model is intended only for text. The data distribution model uses the redundancy achieved through treating a line as an ADU combined with the fact that most successive modifications are to the same line to avoid the need for most retransmissions.

However, the restrictions the data distribution model impose on a data structure consisting of a ordered doubly linked list of application data units can perhaps be generalized some-

what. The imposition of a strict ordering of ADUs, combined with marking deleted ADUs whilst leaving them in position in the ordering, allows the detection of inconsistencies caused by network partitioning in a loose consistency application.

The stacking order of blocks in NTE is a local issue so that overlapping blocks can be edited. In a shared drawing tool, stacking order is a global issue, allowing the overlaying of one object over another to produce a more complicated object. In such a tool, each drawing object (circle, polygon, rectangle, line, etc) is the drawing equivalent of a line of text. The concept of a block does not exist as such, but there is a strict ordering (analogous to line ordering in a block) which is imposed by the stacking order. Thus, the same set of constraints that apply to lines of blocks in NTE should also be applied to the stacking order of drawing objects. We believe many shared applications have similar requirements.

The retransmission mechanism used in NTE is novel, and its requirements are perhaps atypical of shared applications because of the wish to exploit redundancy. For many applications, SRM is a more appropriate choice of retransmission mechanism, as, given a stream of packets with sequence numbers, it is likely to be more timely. However, for applications where retransmission is a relatively rare phenomena due to redundancy or other relaxed consistency requirements, or where we desire a sender controlled system, NTE's retransmission scheme has some possible benefits. Although we do not use the property in NTE, sliding key schemes can be used to ration retransmission requests - this might be useful where the reverse path from receivers to senders is bandwidth limited. In addition, for multicast networks that only support one-to-many multicast with a unicast back channel such as some satellite networks, a sender initiated retransmission request scheme is required.

4 Future Work

The consistency mechanisms implemented in NTE can be generalized as described above, and combined with more traditional deterministic mechanisms and SRM in a single reliable-multicast framework for building shared applications. The goal is that different applications and indeed different objects within the same application can require different reliability modes at different times by relaxing different constraints from section 1.1.

One area that NTE does not address adequately is the issue of congestion control. Although NTE maintains a bandwidth budget, and shapes it's transmissions within this budget, no current mechanism allows us to set this budget effectively and NTE operates with a default budget of 8Kbps. We are investigating mechanisms to allow better congestion control mechanisms within the extended framework to allow higher bandwidths to be used safely when conditions allow.

References

[1] S.Deering and D.Cheriton. “Multicast routing in datagram internetworks and extended LANs.” ACM Transactions on Computer Systems, pp. 85–111, May 1990.

[2] D.D. Clark, D.L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols”, Proc ACM SIGCOMM '90, Philadelphia, Pa, 1990.

[3] S. Floyd, V. Jacobson, S. McCanne, C-G. Liu, L. Zhang, “A Reliable Multicast Framework for Lightweight Sessions and Application Level Framing”, Proc ACM SIGCOMM 1995, Cambridge, Ma.

[4] S. Casner, S. Deering, “First IETF Internet Audiocast”, ACM Computer Communication Review, Vol. 22, No. 3, pp. 92-97, July 1992.

[5] J. Cooperstock and S. Kotsopoulos, “Why use a fishing line when you have a net? an adaptive multicast data distribution protocol,” in Proc. of Usenix Winter Conference, 1996.

[6] K. Lidl, J. Osborne, J. Malcolm: “Drinking from the Firehose: Multicast USENET News”, Proc USENIX Winter 1994, San Francisco, Ca., Jan. 17-21, 1994.

[7] V. Jacobson, “Multimedia Conferencing on the Internet”, Tutorial Notes - ACM Sigcomm 94, London, Sept 1994.

[8] M. Yajnik, J. Kurose, D. Towsley, “Packet Loss Correlation in the MBone Multicast Network” Proc IEEE Global Internet Conf. , London, Nov. 1996.

[9] reference omitted to preserve anonymity

[10] A. Haake, J. Haake, “Take CoVer: Exploiting Version Management in Collaborative Systems”, in Proc. InterCHI'93, Amsterdam, Netherlands, 1993.

[11] J. Munson, P. Dewan, “A Flexible Object Merging Framework”, Proc. ACM CSCW '94, Chapel Hill, North Carolina, 1994.

[12] C. Ellis, S. Gibbs, “Concurrency Control in a Groupware System”, in Proc. ACM SIGMOD '89, Seattle, Wa., 1989

[13] J. Bolot, I.Wakeman, T.Turletti, “Scalable feedback control for multicast video distribution in the Internet”, Proc ACM SIGCOMM 1994, London, UK

[14] W. Dabbous, B. Kiss, “A Reliable Multicast Protocol for a White Board Application”, RR-2100, INRIA, Sophia Antipolis, France, Nov. 1993.

[15] M.R. Macedonia, , D.P. Brutzman, “MBone Provides Audio and Video Across the Internet”, IEEE Computer, Vol.27 No.4, April 1994, pp. 30-36.