Concurrency Control in Groupware Systems

C.A. Ellis
S.J. Gibbs
MCC, Austin, Texas

Abstract. Groupware systems are computer-based systems that support two or more users engaged in a common task, and that provide an interface to a shared environment. These systems frequently require fine-granularity sharing of data and fast response times. This paper distinguishes real-time groupware systems from other multi-user systems and discusses their concurrency control requirements. An algorithm for concurrency control in real-time groupware systems is then presented. The advantages of this algorithm are its simplicity of use and its responsiveness: users can operate directly on the data without obtaining locks. The algorithm must know some semantics of the operations. However the algorithm's overall structure is independent of the semantic information, allowing the algorithm to be adapted to many situations. An example application of the algorithm to group text editing is given, along with a sketch of its proof of correctness in this particular case. We note that the behavior desired in many of these systems is non-serializable.

1. Introduction

Real-time groupware systems are multi-user systems where the actions of one user must quickly be propagated to the other users. An example that many find easy to relate to is the multi-player game. Here the movement of one player's token, perhaps a tank or spaceship, triggers updates on the displays of all players. Other examples can be found in the area of *computer-supported cooperative work* [CSCW86, CSCW88]. For instance, in real-time computer conferencing [Sari85], the users, who are often at different locations, communicate through a software medium. This software might allow the users to view and modify a shared graph structure [Stef87] or edit a shared outline [Elli88a]. Our research group has been studying, and experimenting with these types of systems for several years. We now recognize that there are significant challenges in implementing these systems that typically do not arise in other applications. Some of these challenges [Elli88b] reside in the areas of group interfaces, access control, social protocols, and coordination of group operations. For instance, groupware introduces new complexities to the user interface: the interface must depict group activity, and designers must weigh the need for group focus against the potential for distraction. Concurrency control also has novel aspects within groupware as we will demonstrate in this paper.

An invocation of a groupware system is informally called a *session*. Sessions in which we have participated are typically an hour or two in length but may be much shorter or much longer. At any point in time, a session consists of a group of users called the *participants*. The session provides each participant with an interface to a *shared context*, for instance participants may see synchronized views of evolving data. The system's *response time* is the time necessary for the actions of one user to be reflected by their own interface; the *notification time* is the time necessary for one user's actions to be propagated to the remaining users' interfaces.

Real-time groupware systems are characterized by the following:

- highly interactive — response times must be short.
- real-time — notification times must be comparable to response times.
- distributed — in general, one cannot assume that the participants are all connected to the same machine or even to the same local area network.
- volatile — participants are free to come and go during a session.
- ad hoc — generally the participants are not following a pre-planned script, it is not possible to tell *a priori* what information will be accessed.
- focused — during a session there is high degree of access conflict as participants work on and modify the same data.
- external channel — often participants are connected by an external (to the computer system) channel such as an audio or video link.

It is useful to distinguish groupware systems from other multi-user systems. For example, both database management systems and timesharing operating systems support multiple users. However neither of these are groupware since they provide little notification — if one user performs some action, perhaps inserting a tuple or creating a process, other users are not normally notified of the action and may only learn of it by explicitly querying the system.

One example of groupware is GROVE *(GR*oup *O*utline *V*iewing *E*ditor) [Elli88a], an outline editor which we implemented in the Software Technology Program at MCC. It is intended for use by a group of people simultaneously working on a textual outline. GROVE supports multiple views of the outline, each view is displayed in a *group window* (a window replicated over a set of machines). A window may be private, shared, or public. A GROVE group window is shown in Figure 1. Participants can modify the underlying outline by performing standard editing operations (insert, delete, cut, paste, etc.) in the window, they may also open and close parts of the outline (using the small buttons on the left side) or change the read and write permissions of outline items. In addition to displaying views, group windows also indicate who is using the window. The window in Figure 1 will appear on the workstations of the three users shown along the bot-

addition to displaying views, group windows also indicate who is using the window. The window in Figure 1 will appear on the workstations of the three users shown along the bottom border. An important characteristic of this system is that edits performed by any participant are rather immediately seen on the screens of all participants (real-time notification).

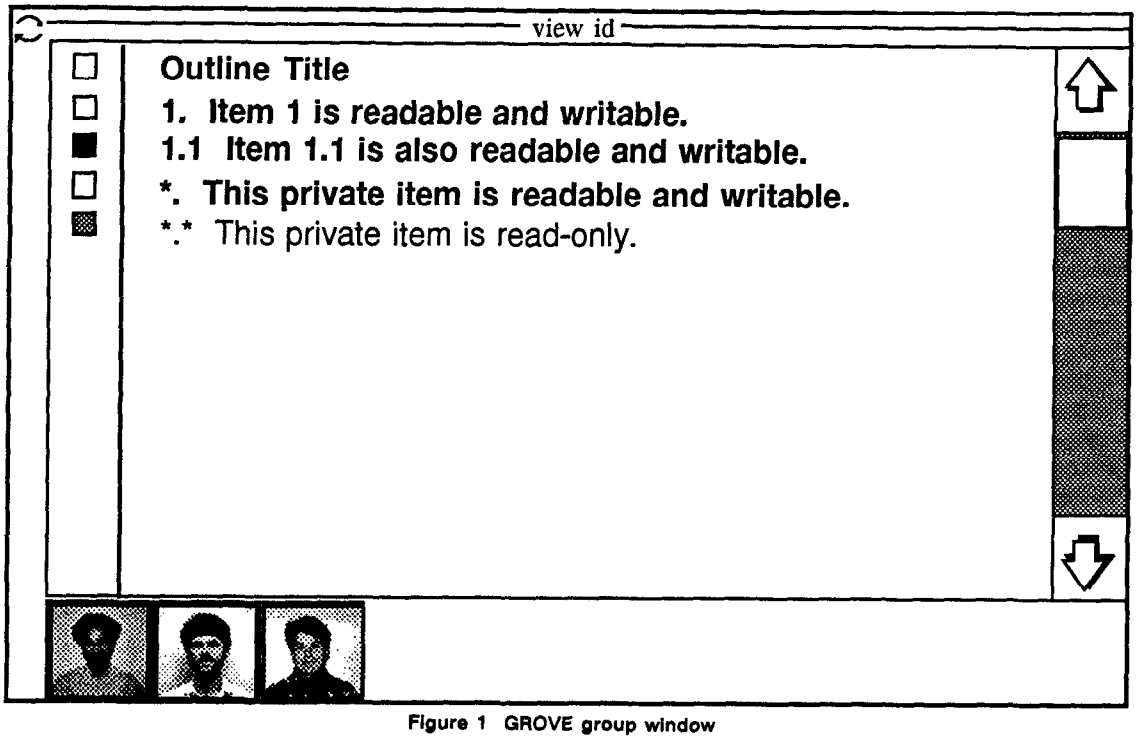

Figure 1 GROVE group window

A very closely related class of groupware systems is what might be called *non-real-time groupware*. Examples are editors such as CES [Grie86], Quilt [Cohe88], or Shared Books [Lewi88]. These editors allow a group of users to work on the same document, however each user typically works on their own section at their own pace. As a result sessions are less focused and are longer in duration (days or even weeks in length); also real-time notification may not be necessary because of the asynchronous nature of participants' actions. Given these distinctions, the remainder of this paper will concentrate upon real-time groupware, although we will simply use the term "groupware systems."

In this paper we present an algorithm for concurrency control in groupware systems. The next section explores issues related to concurrency control in this context and describes problems with alternative approaches. In Section 3 we provide a model for groupware systems. The algorithm is developed in Section 4 and applied to GROVE in Section 5. Discussion of its correctness is presented in Section 6.

2. Concurrency Control Problem

Concurrency control is needed within groupware systems to help resolve conflicts between participants, and to allow them to perform tightly coupled group activities. For example, with a group editor such as GROVE, clearly there is a conflict if one participant deletes a sentence while a second inserts a word into the sentence. The various approaches to resolving these situations, such as explicit locking or transaction processing, that have been developed for database applications do not appear to be suitable in this context. In the following we identify some of the issues related to concurrency control in groupware systems and then discuss the drawbacks of current approaches.

2.1 Issues

WYSIWIS. Although there has been little experience in the evaluation of interfaces to groupware systems [Grud88, Elli89] it appears that some form of a *WYSIWIS* (what you see is what I see) interface [Stef87] is necessary to maintain group focus. If each user sees a slightly different or out-of-date version then the session's cohesiveness is soon lost. WYSIWIS interfaces have two implications on concurrency control. First response times are important — the time taken to access data, modify data, or notify users of changes must be as short as possible. Secondly, if the concurrency control scheme entails the use of modes where actions of one user are not immediately seen by the others, then the effect of these modes on the group's dynamics must be considered and only allowed if they are not disruptive.

Wide-area Distribution. One of the main benefits of groupware systems is that they allow people to work together, in real-time, even though separated by great physical distances. Consequently these systems may be geographically distributed. With current communications technology, transmission times and rates for wide-area networks are significantly worse than those found in their local area counterparts, the possible impact on response time must be taken into account.

Replication. Because the response time demands of groupware systems are so high, the data state is usually replicated for each participant. This allows many potentially expensive operations to be done locally. For instance, consider an editing session where one participant is in Los Angles and the

other Austin. Typically each participant would be working in a windowing environment. If the object being edited is not replicated then even simple scrolling operations require communication between the two sites. The resulting degradation in response time may be catastrophic.

Robustness. Traditionally robustness refers to recovery from unusual circumstances, typically these are component failures — the crash of a site or a communications link. While these are also concerns of groupware systems, there is also a second form of robustness these systems must achieve, in particular, robustness to user actions. For example, the addition of a new user to the set of users issuing transactions on a database is not normally considered a major problem. However, with groupware systems, the addition of a participant may result in what amounts to a reconfiguration of the system. Clearly the concurrency control algorithm must adapt to such reconfigurations and in general recover from "unexpected" user actions (abruptly leaving the session, going away for coffee, ...).

2.2 Other Approaches

Locking. One solution to concurrency control is simply to lock data before it is modified. For instance, in an editor such as GROVE, outline items could be locked whenever a user places their cursor over an item. There are a number of techniques suited to interactive environments which decrease the probability of a lock request being refused. For example, with "tickle locks" [Grie86], a request to a locked resource will be granted if the current holder is inactive. Another technique is to provide participants with visual indicators of locked resources [Stef87] and so decrease the likelihood of requests being issued for locked objects. There are three main problems with locking: First there is the overhead in requesting and obtaining the lock, this may include waiting if the data is already locked. In any case there will be a degradation in response time. Secondly, there is the question of granularity. In a text editor it is not clear just what should be locked when the user moves the cursor to the middle of a line and inserts a character. Should the enclosing paragraph or sentence be locked, or just the word or character? Fine granularity locking is less constraining to the participants but entails greater overhead. The third problem is determining when locks should be requested and released. Using the previous example, should the lock be requested when the cursor is moved or when the key is struck? The system should not burden the user with these questions but it is hard to automatically embed locking into editor commands. For example, if locks are released when the cursor is moved then a user could move to one place to copy some text only to be locked out from pasting it into their previous location.

Transactions. Transaction mechanisms have been used for concurrency control in interactive multi-user systems (for example, CES or Quilt), but, these are loosely-coupled systems and have less demanding response time requirements. For groupware systems there are a number of problems. First there is the complication of distributed concurrency control algorithms based on transaction processing and the subsequent cost to response time. Secondly, if transactions are implemented using locks there are the problems mentioned above, while if some other method is used, such as timestamps, a user's actions may be aborted by the system. (Only aborts explicitly requested by the user should become visible at the user interface.) Generally, transactions are not well suited to interactive use, for instance a user with two transactions active in separate windows on the same object would be presented with two different data states — it would be better if the windows showed the same state.

There appears to be a basic philosophical difference between databases and groupware systems. The former strive to give each user the illusion of being the only user of the system, while groupware systems strive to make the actions of each user visible to the others. Thus the shielding of one user from seeing the intermediate states of another's transaction is in opposition to the goals of interactive groupware systems. There has been some work on "opening up" transactions [Banc85] however the emphasis of this work has been on coordination of nested transactions rather than elimination of the constraints imposed by locking and transactions.

Single Active Participant. Some real-time computer-conferencing systems are intended for situations where only one participant at a time "has the floor" [Lant86]. Access to the floor may be controlled by software or through an external protocol (for example, verbal agreement by the participants). The main problem with this approach is that it is limited to just those situations where having a single active participant fits the dynamics of the session, in particular, it is not suited for sessions with much parallelism among participants. It may overly inhibit the free and natural flow of information among participants. Additionally, if change of floor is left to an external protocol then participants' errors in following the protocol, or non-cooperation (refusal to follow the protocol), can result in two participants believing they have the floor (and potentially issuing conflicting operations).

Dependency-detection. One recent proposal for concurrency control in groupware systems is the dependency-detection model [Stef87]. Dependency detection is based on the use of timestamps to detect conflicting operations; conflicts are resolved by manual intervention. The great advantage of this method is that no synchronization is necessary, non-conflicting operations are performed immediately upon receipt, so response is very good. Mechanisms which involve the user, in general, are appropriate and valuable in groupware applications. However, any method which requires user intervention to assure data integrity is vulnerable to user error.

Reversible Execution. This is another recent proposal for concurrency control in groupware systems. With reversible execution [Sari85], operations are executed immediately but information is kept so that they may be undone later if necessary. Many optimistic concurrency control mechanisms fall within this category [Bern87]. A global time ordering in defined for the operations (this may be provided by a central sequencer or by ordering on timestamps and site identifiers). When it is detected that two operations have been executed out of order, they are undone and re-executed in the correct order. As with dependency-detection, this method is very responsive. It's disadvantages are the need of a global ordering of operations and the unpleasant possibility that an operation will appear on the user's screen and later disappear because it is undone. There are groupware systems in which the forcing of a total ordering is inappropriate. This paper will show an example of operations on a group editor where total ordering produces undesirable behavior.

The model and algorithm which we present next apply to environments where the data is replicated at all sites. Control is distributed and locking is not used.

3. The Model

A groupware system consists of a set of participant systems connected by a communications network. We are particularly concerned with a distributed workstation environment where the local computing facilities associated with each participant are capable of storing a significant subset of the operational

data, performing operations on this data locally, and optionally displaying the data via different user interfaces and different views. We call a participant system a *site*. There is one site per user. Usually a site corresponds to a machine (the user's workstation), however some machines may run multiple sites. We assume that any site can communicate with any other site.

More formally we model a groupware system by a structure of the form G = <S, O> where:

S is a set of sites,

O is a set of parameterized *operators*.

Each site consists of a *site process*, a *site object*, and a unique *site identifier*. The site object is a passive data object. Since the various users will be viewing different site objects, we want to maintain consistency among site objects as much as possible (i.e. site objects should look the same). A site object's structure is application specific; examples could include a document, a hypertext node, or a CAD design object. The set O consists of operators defined on site objects. For example, a simple text processing system might have a character string as its site object at all sites, and then define O as $\{O_1, O_2\}$ where:

O_1 = insert[X; P] insert character X at position P

O_2 = delete[P] delete the character at position P

The operations which can be applied to any site object are particular instances of the operators. In the previous case this would be things like $o = O_1$[x; 3]. Each operation changes the state of the site object, for instance o('abc') is 'abxc'.

The site process performs three kinds of activities:

1) operation generation: an operation is specified by the site's user. The site process encapsulates the operation in an *operation request* which is broadcast to all sites.

2) operation reception: an operation request is received from some site.

3) operation execution: an operation request is executed.

We make the following assumptions:

A1 the number of sites is constant,

A2 messages are received exactly once and without error,

A3 it is not possible for a message to be executed before it is generated.

It is possible to extend this model and our algorithm to account for a varying number of sites over time. This extension will be discussed at the end of this section. The second assumption should be satisfied by the communications protocols so we take it as given.

Given the above model, we now define correctness of a groupware system. First, following Lamport [Lamp78], we define a partial ordering of all operations in terms of the local generation and execution sequences.

Definition. Given operations o and p, generated by sites s and t, then *o precedes p* iff:

1) $s = t$ and the generation of o happened before the generation of p, or

2) $s \neq t$ and the execution of o at t happened before the generation of p.

Definition. The *Precedence Property* states that if one operation, o, precedes another, p, then at each site the execution of o happens before the execution of p.

Definition. A groupware session is *quiescent* iff all generated operations have been executed at all sites, that is, there are no requests in transit or waiting to be executed by a site process.

Definition. The *Convergence Property* states that site objects are identical at all sites at quiescence.

Definition. A groupware system is *correct* iff the *convergence property* and the *precedence property* are always satisfied.

One possible solution is to extend the partial ordering obtained from precedence to a total ordering. If operations are executed in this order at all sites then both the precedence and convergence properties are satisfied and the system is correct. The total ordering may be obtained by the use of a distributed algorithm [Lamp78] or by use of a centralized *coordinator* process. (Using a coordinator, a request is first sent to the coordinator which then broadcasts the request to all sites.) However, in addition to lack of robustness, there are two problems with relying on a total ordering. First, responsiveness is lost since operations are not executed when requested but after some delay. Secondly, if a user requests an operation, their user interface should then be locked until the request has been executed — otherwise a following request will refer to the current state of the site object but will be performed when the site object is in a possibly different state.

We seek a solution where the only constraint on execution order is the precedence property. The problem is that the precedence property does not imply the convergence property. This can be seen by looking at non-commuting operations. For instance, take the system G = <{S_1 S_2,}, {delete}> and the operation orderings shown in Figure 2.

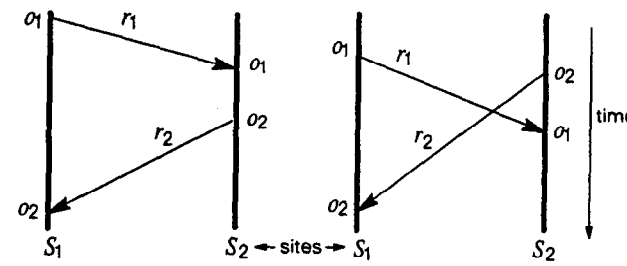

Figure 2 Non-overlapping (a) and Overlapping (b) Operations

In the non-overlapping case, o_1 precedes o_2 and both sites execute the operations in the same order resulting in convergence. However, when the operations "overlap" there is no precedence among o_1 and o_2. The order of execution need not be the same at each site and the system may not converge. For example, if o_1 = delete[3] and o_2 = delete[2] then:

$o_1 \circ o_2$('abcd')) = o_1('acd') = 'ac'

$o_2 \circ o_1$('abcd')) = o_2('abd') = 'ad'

(Here $\circ$ indicates composition of operations.)

Even more troublesome is the case where the deletes overlap and refer to the same position. Here, since neither site sees the other's delete before generating its delete, we would like each site to only delete one character and treat the second delete as redundant. However, there is no serial ordering of the two delete operations which produces the desired result—in either case an extra character is deleted. This is one example of non-serializable behavior. Much of the literature concerning database concurrency and replicated systems defines correctness in terms of serializability. This example suggests situtations where non-serialized correctness criteria are needed.

The approach we propose is that, faced with situations such as the above, instead of executing $o_1 \circ o_2$ at one site and $o_2 \circ o_1$ at the other, we execute $o_1' \circ o_2$ and $o_2' \circ o_1$ where o_1' and o_2' are *transformed* operations. o_1' and o_2' are obtained from o_1 and o_2 respectively, and calculated so that $o_1' \circ o_2$ when applied to a site object has the same effect as $o_2' \circ o_1$. In the following we describe how operations may be transformed and when this should be done.

4. The Algorithm

We will now look at a concurrency control algorithm for groupware systems which is based upon the notion of operation transformation. This algorithm has a number of properties which make it suitable for groupware. First, operations are performed immediately on their originating site, thus responsiveness is good. Secondly, locks are not necessary so all data remains accessible to group members. Finally, the algorithm is fully distributed, and resilient to site failure (in the sense that the remaining sites can continue without interruption).

Intuitively, the algorithm works as follows. When an operation is generated at a site, the site process executes the operation, generates a priority for the operation, and sends information including the operation and its priority to all other sites. When an operation is received at a site (site i), auxiliary information is examined to see if the sending site executed operations which have not yet been executed at site i (i.e., a "future operation"). If so, the operation is queued; if not, the operation is executed. If site i has executed operations which have not yet been executed at the sending site (i.e., a "past operation"), then the priority of the operation is checked and the operation may be transformed before it is executed. In the remainder of this section, we describe this algorithm more precisely. The priority is based upon the site identifiers, and is application dependent. Therefore it will be explained in section 6 in the context of our group editor application.

4.1 Transformation Matrix

Transformation Matrix. The transformation matrix is the key to resolving conflicting operations. Given a groupware system G = <S, O>, then G's transformation matrix, T, is an m x m matrix, where m is the cardinality of the set O. Each component of T is a function which transforms operations into other operations. Given two operations o_i and o_j, with priorities p_i and p_j, and which are instances of operators O_u and O_v, let:

$$o_j' = T_{uv}(o_j, o_i, p_j, p_i)$$
$$o_i' = T_{vu}(o_i, o_j, p_i, p_j)$$

then T is such that following is satisfied:

$$o_j' \circ o_i = o_i' \circ o_j$$

This represents a necessary, but not sufficient condition. Specification of a general, sufficient condition is an elusive endeavor. We consider this specification further in the next section in conjunction with our group editor example.

4.2 Data Structures

State Vectors. Let N be the number of sites in the system (we are assuming that N is constant). Each site has a unique identifier, the site ID (for example its network address). For simplicity assume that the sites are identified by the integers 1 ... N. The state vector of site j, is a N component vector where the i'th component indicates how many operations from site i have been executed by j. Given two state vectors, s_i and s_j, we say that:

1) $s_i = s_j$ if each component of s_i is equal to the corresponding component in s_j,
2) $s_i < s_j$ if each component of s_i is less than or equal to the corresponding component in s_j and at least one component of s_i is less than the corresponding component in s_j ($s_i \leq s_j$ provided $s_i < s_j$ or $s_i = s_j$).
3) $s_i > s_j$ if at least one component of s_i is greater than the corresponding component in s_j.

Requests. Requests are tuples of the form <i, s, o, p> where i is the originating site's identifier, s the originating site's state vector, o is an operation, and p is the priority associated with o.

A request's state vector is used to enforce the precedence property. For instance, in both cases shown in Figure 3, site S_2 can determine from r_2's state vector that according to the dOPT algorithm, it must await the arrival of r_1 before executing r_2's operation.

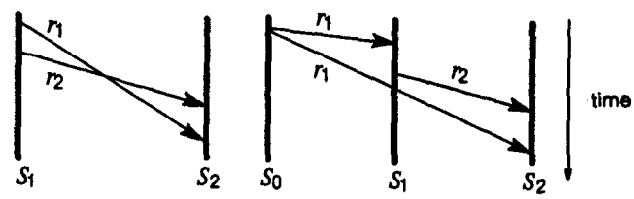

Figure 3 Precedence Detection

Request Queue. Requests waiting to be executed by a site process are kept in the site's request queue. A request, <j, s, o, p>, in Q_i, the request queue for site i, indicates that o has been requested by site j while in state s. An entry is added to the request queue when: 1) the site process receives a request from the network, or 2) the site process receives a request from site's user. Entries are removed from the request queue when the site process determines that the requested operation may be executed. (Note, although we use the term "queue," entries need not be removed in first-in-first-out order.)

Request Log. Each site process maintains a log of requests executed at its site. A request, <j, s, o, p>, in L_i, the log for site i, indicates that site i while in state s, executed operation o (requested by site j). The log is ordered by insertion so it is possible to find the first entry, the most recent entry, and to step through in insertion order.

4.3 The Distributed Operational Transformation (dOPT) Algorithm

We now give a specification of what we call the distributed operational transformation (abbreviated as dOPT) algorithm. The following is the algorithm as executed by the i'th site process.

Initialization:
$Q_i \leftarrow$ empty
$L_i \leftarrow$ empty
$s_i \leftarrow$ <0, 0, ..., 0>

```
Generate Operations:
  receive operation o from the user interface
  calculate the priority, p, of o
  Q_i ← Q_i + <i, s_i, o, p>
  broadcast <i, s_i, o, p> to all other sites

Receive Operations:
  receive <j, s_j, o_j, p_j> from the network
  Q_i ← Q_i + <j, s_j, o_j, p_j>

Execute Operations:
  for each <j, s_j, o_j, p_j> ∈ Q_i where s_j ≤ s_i begin
    Q_i ← Q_i - <j, s_j, o_j, p_j>
    if s_j < s_i
      <k, s_k, o_k, p_k> ← most recent entry in L_i
                where s_k ≤ s_j (or ∅ if none)
      do while <k, s_k, o_k, p_k> ≠ ∅ and o_j ≠ ∅
        if the k'th component of s_j
                is ≤ the k'th component of s_k
          let u be the index of o_j (o_j is an instance of O_u)
          let v be the index of o_k (o_k is an instance of O_v)
          o_j ← T_uv(o_j, o_k, p_j, p_k)
        fi
        <k, s_k, o_k, p_k> ← next entry in L_i (or ∅ if none)
      od
    fi
    perform operation o_j on i's site object
    L_i ← L_i + <j, s_i, o_j, p_j>
    s_i ← s_i with j'th component incremented by 1
  end
```

The initialization section simply sets the site's log and request queue to empty and initializes the site's state vector. The second section receives a local operation (from the site's user), forms a request which is added to the request queue Q_i, and broadcasts the request to the other sites. The third section adds requests from the network to the request queue.

The bulk of the algorithm deals with operation execution. The request queue is examined for requests, $r_j = \langle j, s_j, o_j, p_j \rangle$, which may be executed. Whether r_j can be executed is determined by comparing s_j to the site's state vector s_i; there are three possibilities:

Case I: $s_j > s_i$

The request cannot be executed at this time (site j has executed operations which have not yet been executed by site i) and so is left on the queue.

Case II: $s_j = s_i$

When the two states are equal o_j is executed immediately without transformation.

Case III: $s_j < s_i$

In this case the request can also be executed. However, since r_j refers to a state "older" than site i's current state, o_j must first be transformed. Site i's log, L_i, is examined for requests which were not accounted for by site j (i.e., requests which i has executed but which were not executed on j prior to the generation of r_j). Each such logged request is then used to transform o_j.

The second and third case both lead to the execution of o_j. When this occurs the operation is applied to the site object, added to the log, and the state vector is incremented.

4.4 Quiescence

As mentioned at the beginning of this section we have made a number of assumptions (fixed number of sites, no failures) that could in practice limit the usefulness of this algorithm. Furthermore, as it stands, the algorithm has a continuously growing data structure (the log) which must be scanned when calculating a request's priority and when transforming operations. These problems can be solved by regularly quiescing the system. Quiescence should be enforced periodically (e.g., once per minute) and when the system falls quiet for an extended period of time (e.g., ten seconds). Detection of quiescence is equivalent to the well known distributed consensus problem; algorithms for this problem may also detect site failures (dependent upon the characteristics of the underlying processing and message systems). When the system is quiescent the log can be reset (set to null) and participants can enter and leave the session. Furthermore the session object can be check-pointed by each site.

5. An Example

We now consider the four transformation matrix entries for the insert and delete operations. This transform is applied in cases where the sending site is different from the receiving site. We are thus assured that i is not equal to j; therefore (as we shall see in the next section) the priority p_i does not equal p_j. If we have two operations o_i = insert[X_i; P_i] and o_j = insert[X_j; P_j] originating from sites i and j then T_{11} is defined as:

```
T11(o_i, o_j, p_i, p_j) = o_i'   where
  if(P_i < P_j)
    o_i' = insert[X_i; P_i]
  else if(P_i > P_j)
    o_i' = insert[X_i; P_i + 1]
  else /* P_i = P_j */
    if(X_i = X_j)
      o_i' = ∅
    else
      if(p_i > p_j)
        o_i' = insert[X_i; P_i + 1]
      else /* p_i < p_j */
        o_i' = insert[X_i; P_i]
      fi
    fi
  fi
```

There are two interesting cases in the definition of T_{11}. First, when the arguments of both operations are equal, o_i' is set to null. The reason is that since $i \neq j$ (T is never applied to pairs of operations from the same site) it must be that two different sites have requested the same operation before seeing each others' request, hence it is possible to ignore one request. The second interesting case is when $P_i = P_j$ (i.e., the positions are equal), such conflicts are resolved by using the priority order associated with each request. Priority is used for tie-breaking when similar operations are initiated concurrently. All nodes must order these in the same way. One would think that a priority consisting of the unique site ID would be sufficient, but we will see in the next section that it is not. The priority of an operation used in T is a composite of its predecessor's priority and its originating site ID.

The remaining entries of T are quite simple, we list them here for completeness:

T_{22}(delete[P_i], delete[P_j], p_i, p_j) = o_i' where
if(P_i < P_j)
o_i' = delete[P_i]
else if (P_i > P_j)
o_i' = delete[P_i - 1]
else
o_i' = ∅
fi

T_{12}(insert[X_i; P_i], delete[P_j], p_i, p_j) = o_i' where
if(P_i < P_j)
o_i' = insert[X_i; P_i]
else
o_i' = insert[X_i; P_i - 1]
fi

T_{21}(delete[P_i], insert[X_j; P_j], p_i, p_j) = o_i' where
if(P_i < P_j)
o_i' = delete[P_i]
else
o_i' = delete[P_i + 1]
fi

6. Discussion of Correctness

Lets consider the above examples where the operations are GROVE editor operations being applied to textual outlines. The session object for GROVE is a tree of outline items where each item contains text. Rather than specify T in full for all of GROVE's operators, we will restrict consideration to the simplification where the session object is a character string, and the two operators are O_1 = insert[X; P], and O_2 = delete[P] as previously defined.

Lets examine the workings of our algorithm for the session shown in Figure 4 which has three sites and multiple overlapping operations. We first do this using the site number as the priority. Suppose that initially the site object is the null string at all sites and that site 1 initiates operation insert[x;1]; site 2 initiates insert[y;1]; and site 3, insert[z;1]. At site 3, the z is first appended to the site object, then the insert x operation arrives. Although its priority is lower, it is appended in front of the z (due to temporal precedence). Finally the insert y operation arrives and its priority is compared to both of the previous two operations. Since its priority (2) is lower than that of site 3, it is changed by the transform T to insert[y;2] and is inserted after the x and before the z. The final string is xyz.

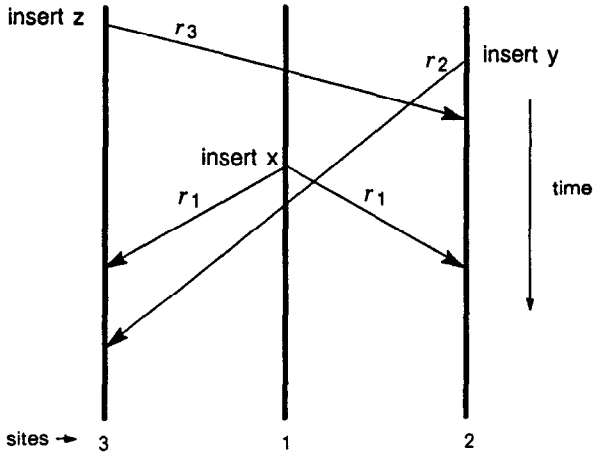

Figure 4 Mixed Priority Example

At site 2, it first prefixes y to its session object, and then prefixes z from site 3. When the insert x operation arrives from site 1, its priority is compared to that of the previously executed insert y operation, and this causes the incoming operation to be changed to insert[x;2]. Note that due to temporal precedence, the operation's priority is not compared to the insert z from site 3. The final string at this site is zxy which is inconsistent with the xyz at node 3.

In Figure 4, the x generated by the operation instigated by site 1 must occur to the left of the z instigated by site 3 because the participant at site 1 saw the insertion of the z, and then explicitly requested an operation which would place an x in the position in front of (to the left of) the z. Transformations and other insert operations could cause other characters to be inserted between the x and the z, but to satisfy our correctness criteria, they should not cause the x to occur after (to the right of) the z in the final string.

Temporal precedence suggests a problem with one of the transforms in the session depicted by Figure 4. At site 2, the position parameter of the insert x operation is incremented before it is executed. The intent is to move the x past the y which has higher precedence, but the actual effect is to move the x past the z which violates the temporal precedence constraint. At this point we might try changing the transform rule so that if the priority of x is less than the priority of y, then we put x to the left of y (rather than to the right as we proposed above.) Although this patch works in this case, it fails in other rather similar cases; the problem is deeper than this and requires a different type of solution.

The fundamental difficulty with this example is the dilemma that our rules require x to be placed before y and y to be placed before z and z to be placed before x. This dilemma arises because node 2 has a precedence that separates a predecessor priority (3) from its successor's priority (1). Our suggested solution is to define the priority of an operation as a list composed of its largest predecessor's priority, concatenated with its own site ID, so that an unrelated operation's priority can never separate them. Note that this priority calculation only needs to be applied to operations with the same position parameter. In this example, the operation insert x would have a priority of (3,1) which is a list containing its site ID preceded by its predecessor's ID. The priority of (3,1) as well as its predecessor (3) is defined to be larger than the priority (2), so this operation will have a very similar priority to its predecessor. We next state the generalization of this which works for an arbitrary directed acyclic graph (dag) of operations, and sketch the outline of its proof.

Priority Calculation: The priority of an operation o_i is calculated at the originating site at the time of execution, and is included when the operation is sent to all other sites. If there are operations in the local log whose original position (before transformation) is equal to the operation being initiated, then the highest of these priorities is chosen and the local site ID is appended to it to form the priority of o_i. If there are no such operations in the local log, then the priority is simply the local site ID.

Priority Comparison: Two priorities are compared element by element, from beginning of the list to the first element in which they differ. Which ever is larger at the element where the lists differ has the higher priority. If one list is a sublist of the other, then one operation is a predecessor of the other.

We choose the longer list to have higher priority so that the successor will be placed to the left of the predecessor in the session object.

Proof Outline: The details of the full proof are beyond the scope of this paper, so we only sketch the main ideas here. Each operation arrives with its state vector and its priority. An invariant which is useful within our proof is the position counter. Each inserted character may be shifted to the left or right during the course of its session, and the net amount of this shift at a particular site is the position counter for that character. Note that this shifting may be explicit by way of transformations applied to the operation when it is being executed, or implicit when a later executed operation inserts or deletes another character at the same or an earlier position in the string.

The proof that the proposed algorithm using the priority list scheme is correct proceeds by showing that (1) the same value will be attained for the position counter of an arbitrary character no matter what ordering of operations is used (consistency), and (2) if o_i occurs before o_j at the same insertion position, then the position counter of o_i will be greater than the position counter of o_j (temporal precedence).

The operations performed in a session could arrive at a passive site (i.e. a site which initiates no operations) in any order. Any arrival ordering of these operations can be obtained from any other ordering by a sequence of transpositions of adjacent operations. Therefore the proof of correctness can proceed by showing that the correctness properties are maintained during any transposition.

We will not examine all cases, but consider the critical case in which two overlapping insert operations at the same position, o_i = insert[x_i; P] and o_j = insert[x_j; P], are transposed. The following discussion argues that the values of the position counters for these inserted characters (denoted by C_i and C_j), and those of other operations, is not altered by the transposition.

Since these operations overlap, we know that neither is a temporal precedent to the other. Assume without loss of generality that the priorities are $p_i < p_j$. Whenever o_i is executed before o_j, the log is searched and C_i is altered from the initial value of P to some new value P_i; later when o_j is executed, x_j is inserted *before* x_i causing x_i to be implicitly shifted to the right. Thus in this case, the value of C_i after o_i and o_j complete is P_i+1. If we now transpose, and if o_j is executed before o_i, then o_i encounters the same log items, but it also encounters o_j in the log. At this encounter, C_i is incremented, so that its final value is P_i+1 in this case also.

Now consider the operation o_j. First suppose o_j is executed before o_i. Let C_j have some value P_j after the execution of o_j. Then when the log is examined during the processing of o_i, since o_j's priority is higher than o_i, C_j is not incremented. In the case where o_j is executed after o_i, it sees o_i in the log, but priority considerations leave C_j unincremented. So the value of C_j is unchanged by the transposition. Obviously, the transposition has no effect upon the counters of other operations in the session, so our argument is complete.

In the case of temporal precedence between the operations being transposed, queuing will necessarily occur so that the order of actual execution is not changed. To verify the temporal precedence property, one must argue from the definitions of the priority list and the priority calculation to infer that if o_i occurs temporally before o_j, then the position counter of o_i will always be greater than the position counter of o_j ($C_i > C_j$).

We have not discussed the case involving insertions at different positions. The position counter of a character is affected by insertion of a character at a smaller valued position (earlier in the string), but not by insertion at a larger valued position. There is no need to use priority numbers in this case. We have also not discussed cases involving deletion. These cases are easier because deletions are frequently commutative. If o_i and o_j are deletions, then o_i followed by o_j is frequently the same as o_j followed by o_i.

7. Conclusions and Future Developments

This paper has introduced the notion of groupware, and presented a novel algorithm for concurrency control within groupware systems. Groupware reflects a change in emphasis from using the computer to solve problems to using the computer to facilitate human interaction. For these systems to be accepted requires fine-granularity sharing of data, rapid response time, and rapid notification time. Therefore, the algorithm presented does not use locking, performs non-serializable sets of operations, and works in a workstation environment with replicated data and distributed control.

This algorithm was presented in the context of GROVE, a group editor which we have implemented within the Software Technology Program at MCC. It also seems applicable to other groupware such as hypertext systems. Notice that in these systems, users frequently apply associative access techniques rather than accessing objects by name. Thus, within a text editing application, users browse and point at the items they want to update; the individual characters are not given separate immutable names or unique addresses.

Future work includes generalizing the characterization of our transform matrix, and extending our proof technique to encompass hypertext and other groupware. We will also continue to incorporate these ideas within groupware systems which we are constructing, and will be constructing in the future. We have been developing and applying the notion of a *team automaton* to model and prove various properties of groupware systems. We hope that this will be useful in our correctness proof extensions. In conclusion, we believe that the new emphasis on groupware suggests a number of interesting and challenging frontiers. Perhaps this paper can help to stimulate work at these frontiers.

Acknowledgements

Thanks to Gail Rein, the other member of the groupware team, who put in much effort to help implement GROVE and its user interface. Thanks also to Ira Forman, whose advice and comments were very useful in our consistency and correctness work. We are pleased to acknowledge MCC and our managers within the Software Technology Program at MCC for creating a stimulating environment within which this work could be done.

References

[Banc85] Bancilhon, G., Kim, W., and Korth, H.F. A Model of CAD Transactions, *Proc. Intl. Conv. on Very Large Data Bases*, pp. 25–33, 1985.

[Bern87] Bernstein, P., Goodman, N., and Hadzilacos, V. *Concurrency Control and Recovery in Database Systems*, Addison–Wesley, 1987.

[Cohe88] Cohen, M., Fish, R., Kraut, R., and Leland, M. Quilt: A Collaborative Tool for Cooperative Writing, *Proc. ACM SIGOIS Conf.*, pp. 30–37, 1988.

[CSCW86] *Proceedings of the Conference on Computer–Supported Cooperative Work* (Austin, Tex., Dec. 3–5). ACM, New York, 1986.

[CSCW88] *Proceedings of the Conference on Computer–Supported Cooperative Work* (Portland, Or., Sept. 23–25). ACM, New York, 1988.

[Elli88a] Ellis, C.A., Gibbs, S.J., and Rein, G. Design and Use of a Group Editor, Rep. STP–263–88, MCC Software Technology Program, Austin, Tex., 1988.

[Elli88b] Ellis, C.A., Gibbs, S.J., and Rein, G. Groupware, Rep. STP–414–88, MCC Software Technology Program, Austin, Tex., 1988.

[Elli89] Ellis, C.A., Rein, G.L., and Jarvenpaa, S.L. Nick Experimentation: Some Selected Results, *Hawaii Int. Conf. on System Sciences*, 1989.

[Grie86] Grief, I., Seliger, R., and Weihl, W. Atomic Data Abstractions in a Distributed Collaborative Editing System, *Proceedings of the 13th Annual Symposium on Principles of Programming Languages*, pp. 160–172, 1986.

[Grud88] Grudin, J. Why CSCW Applications Fail: Problems in the Design and Evaluation of Organizational Interfaces, *Proc. CSCW'88*, 1988.

[Lamp78] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System, *CACM* 21(7), pp. 558–565, 1987.

[Lant86] Lantz, K.A. An Experiment in Integrated Multimedia Conferencing, *Proc. Conf. on Computer–Supported Cooperative Work*, pp. 267–275, 1986.

[Lewi88] Lewis, B. and Hodges, J. Shared Books: Collaborative Publication Management for an Office Information System, *Proc. ACM SIGOIS Conf.*, pp. 197–204, 1988.

[Sari85] Sarin, S. and Grief, I. Computer–Based Real–Time Conferences, *Computer*, 18(10), pp. 33–45, 1985.

[Stef87] Stefik, M., Foster, G., Bobrow, D.G., Kahn, K., Lanning, S., and Suchman, L. Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings, *CACM* 30(1), pp. 32–47, 1987.