An Algorithm for Distributed Groupware Applications

Alain Karsenty and Michel Beaudouin-Lafon
Laboratoire de Recherche en Informatique (CNRS URA 410)
Université de Paris-Sud - Bâtiment 490
91405 ORSAY Cedex - FRANCE

Abstract

CSCW is a new and rapidly growing field. An important problem is concurrency control: current algorithms impose constraints on the user or are not general enough to be used by other applications. We present an algorithm that gives the best response time for the interface and, using the semantics of the application, reduces the number of undo and redo operations when conflicts occur.

Introduction

Computer Supported Cooperative Work (CSCW) is a rapidly growing field. Groupware applications support groups of people working together and range from electronic mail to real time multi-user editors that allow users in different locations to edit the same document simultaneously. Although a number of groupware systems have been implemented, including some commercial products [3, 10, 5], many architectural and structural issues remain. Some problems that are already difficult, such as the implementation of distributed systems, pose additional problems for groupware. For example, current tools attempt to make distribution as transparent as possible. Yet groupware requires a combination of transparency and awareness, to let users be aware of other users in the system. Because these requirements do not match, much of the research on distributed systems cannot be applied directly to groupware. Groupware developers are thus unable to use the tools designed for implementing distributed systems and must develop such tools themselves.

We are investigating real-time (or synchronous) groupware and are particularly interested in extending the usual interaction paradigms to include the dimension of the group [2]. This work led us to implement a specific system for managing the distribution of the application. This paper reports on the architecture and concurrency control algorithm used in this system. The algorithm is based on the semantics of the application and can be used by the developers of other groupware systems. We begin by introducing the notion of a purely replicated architecture and then present GroupDesign, a shared drawing tool implemented with this architecture. We then present the main parts of the algorithm that implement the distribution. We conclude with a review of related work and a discussion.

The purely replicated architecture

We address real-time groupware systems that allow a group of users to edit a shared document. We advocate a fully distributed architecture for such systems, both to optimize the response time and to make the system more fault-tolerant. Our architecture replicates the application at each site and does not need a centralized component. Because each site runs the full application, the whole system continues to work even if one site crashes. We call this a *purely replicated architecture*. We can view the system as a set of interactive applications that communicate with each other, rather than as a single distributed system. This is particularly true in a heterogeneous environment where the different replicas are different programs [12]. Our architecture acts as a super-architecture over existing architectures and is compatible with the current models and architectures used for interactive systems.

We have identified several requirements for the implementation of real-time groupware. First, it must provide immediate response time. User commands must be handled immediately without waiting for either authorizations or acknowledgments from the other sites. In our architecture, whenever a user issues a command, the local state of the user's replica is immediately updated, thus introducing an inconsistency with the states of the other replicas. The command is then sent asynchronously to the other sites. The state will become consistent again when the message has been received and handled by all other sites. The property of immediately handling local commands insures the best possible response time, namely that of a single-user application. Moreover, the fact that

the protocol is asynchronous means that distant sites connected with a low bandwidth network will not impair the global performance of the system. Each site can go at its pace without affecting the whole system. This requirement is fundamental for the success of groupware systems: users will not use groupware that slows down their individual tasks.

The second requirement is that the replicated architecture should neither enforce nor favor any particular coordination scheme, such as floor control. It is quite obvious that enforcing floor-taking simplifies the problem of concurrency control, while running an open floor makes it more difficult. Our architecture runs an open floor and supports a wide range of coupling mechanisms [7].

Finally, the protocol must be as application independent as possible. It should be possible to provide a software package that encapsulates the management of the replication. Our experience in implementing GroupDesign from an existing application shows how this is possible.

GroupDesign allows a set of users, each on a Macintosh, to create and modify a diagram simultaneously. A diagram is a set of pages that contain structured graphics which can be edited, as in MacDraw. We implemented the algorithm described below on top of an existing extensible drawing tool called MetaDesign [14]. We could add the replication algorithm without modifying the existing application, using Apple System 7's new features for inter-process communications (High Level Events) [1]. We developed a set of features that complement the drawing tool in order to support the shared aspect of the editing tasks. The purely replicated architecture made these features easy to implement; a more detailed description can be found in [2].

In the rest of the paper, we use GroupDesign to illustrate the purely replicated architecture and to explain the concurrency control algorithm. We begin with a description of the ORESTE (Optimal RESponse TimE) concurrency control algorithm for groupware applications and the PROMOTE algorithm included in ORESTE. We first outline the algorithm and then present the model and a formal description of the algorithm based on this model.

ORESTE Concurrency Control Algorithm

The algorithm is based on a semantic model of the application. A document is a set of objects that are identified by a globally unique identification number. Users issue commands which correspond to operations. These operations are first immediately executed on the local site and then broadcasted to the other sites as events. Events contain the operations and other bookkeeping information, including a timestamp. Timestamps define a total ordering of events. The PROMOTE algorithm. relaxes this total ordering by allowing an event to be executed out of order only if the final state of the document is the same as if the events were executed in the total order. Moreover, if the execution of an event does not modify the resulting diagram (we say that the event is masked), then the event need not be executed.

The algorithms use two sets: L and Q. L is the log of operations which have been executed. It is used in the PROMOTE algorithm whenever conflict occurs and operations need to be undone. Q is the queue of operations that cannot be executed because the object to which they are applied is not yet created.

The ORESTE algorithm has the following properties: (1) operations are immediately executed at the local site; (2) events from a remote site are executed as soon as possible; (3) the size of the log is bounded.

Property (1) ensures the best response time of the interface (first requirement). Property (2) means that the system will not be impeded if an event sent by a site is delayed since even events sent at a later time are executed. Property (3) makes the algorithm realistic, i.e. possible to implement since L does not grow indefinitely.

We now introduce the model of a groupware session, which is defined as a set of sites that communicate with each other. The document is replicated; each site holds a copy that consists of a set of objects.

Objects and operations

An *object* is defined as a 2-tuple $o = (id_O, st_O)$. $id_O \in I$ is the unique identification number of the object and st_O is its state. Ids must be unique over a whole session. The set of all possible objects is called O.

Each site has a function called Obj that maps ids to objects. This function distinguishes between currently existing objects, objects to be created and objects that have been deleted.

$$\begin{aligned} \text{Obj: } I &\rightarrow O \cup \{\varnothing, \perp\} \\ id &\rightarrow (id, st) \text{ if the object exists} \\ id &\rightarrow \varnothing \text{ if the object does not exist yet} \\ id &\rightarrow \perp \text{ if the object does not exist anymore} \end{aligned}$$

$\mathcal{F}$ is the set of functions that can be applied to objects. $\mathcal{F}$ is partitioned into families: $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, ... \mathcal{F}_n\}$. Each family corresponds to a type of operation. For example color change is a family that includes the functions for changing color to red, blue, etc. The functions of family $\mathcal{F}_i$ have a domain of definition $O_{\mathcal{F}_i}$. For example, the family of font change functions applies only to text objects while the family of resizing objects applies only to graphic objects.

An *operation* is defined as a function $f \in \mathcal{F}_i$:

$$\begin{aligned} f: \quad O_{\mathcal{F}_i} &\rightarrow O \\ (id_O, st_O) &\rightarrow (id_O, st'_O) \end{aligned}$$

Two particular families of functions for creating and deleting objects are defined as follows:

$f \in Create: \{\varnothing\} \rightarrow O$

$\varnothing \rightarrow (id_O, st_O)$

$f \in Delete: O \rightarrow \{\perp\}$

$(id_O, st_O) \rightarrow \perp$

In order to undo operations we assume that every function has an inverse:

$$\forall \mathcal{F}, \forall f \in \mathcal{F}, f \circ f^{-1} = \text{Identity}$$

Events and sites

An *event* is a message that contains an operation to be executed by the receiving site. An event is a 5-tuple $e = (f_e, id_e, t_e, site_e, seq_e)$ with:

- f_e: operation;
- id_e: id of the object to apply f_e to;
- t_e: logical time when the event was sent, as defined in [13];
- $site_e$: id of the sending site;
- seq_e: sequence number of the event.

The *total order relationship* of events (ρ) is defined as follows:

$$e_i \rho e_j \Leftrightarrow \left(t_{e_i} < t_{e_j}\right) \vee \left(t_{e_i} = t_{e_j} \wedge site_{e_i} < site_{e_j}\right)$$

A *site* is defined as a 8-tuple $s = (site_S, S_S, t_S, L_S, Q_S, Obj_S, lastSeq_S, lastClock_S)$

- $site_S$: unique id of the site;
- S_S: set of all sites;
- t_S: logical clock of site $site_S$;
- L_S: log of events;
- Q_S: queue of events waiting for execution;
- Obj_S: function mapping an id to an object.
- $lastSeq_S$: array holding the sequence number of the last event received in sequence from each sit
- $lastClock_S$: array holding the timestamp of the last event received in sequence from each site.

Properties of events

- commute (e_1, e_2): events e_1 and e_2 *commute* iff
  $(id_{e_1} \neq id_{e_2})$ or $(id_{e_1} = id_{e_2}$ and $f_{e_1} \circ f_{e_2}(Obj(id_{e_1})) = f_{e_2} \circ f_{e_1}(Obj(id_{e_1}))$)
- mask (e_1, e_2): event e_1 *masks* e_2 iff
  $id_{e_1} = id_{e_2}$ and $f_{e_1} \circ f_{e_2}(Obj(id_{e_1})) = f_{e_2}(Obj(id_{e_1}))$
- conflict (e_1, e_2): events e_1 and e_2 are in *conflict* iff
  not commute (e_1, e_2) and not mask (e_1, e_2)
- commuteWithSet (e, S): event e commutes with set S iff
  $\forall e_i \in S$, commute (e, e_i)

Quiescence

The system is said to be quiescent when all the events sent by the sites have been received and handled. We assume that the network is reliable: events that are sent arrive only once, but they need not arrive in the order they were sent. The sequence number contained in the events can be used to implement a reliable protocol over an unreliable network.

Algorithmic notation

The sets L and Q are sorted by increasing timestamp. They are modified by the procedures Insert and Remove. To step through a set we use the notation:

for e **in** L **do** instruction() **endfor**

This expression steps through the set L in timestamp order (from the oldest to the most recent event) and executes "instruction()" for each event. The following steps through a set in reverse order:

for e **in** reverse(L) **do** instruction() **endfor**

```
/* the procedures execute(Event) and undo(Event) must
be defined in the application */
procedure SendEvent (e : Event)
begin
    /* send to everybody else */
    Multicast (e, S_S - {site_S});
    /* update local state */
    t_S := t_S + 1;
    seq_S := seq_S + 1;
    Insert (e, L_S);
end /* SendEvent */

procedure ReceiveEvent (e : Event)
begin
    case Obj_S(id_e) of
    ∅:              /* object does not exist */
        if f_e ∈ Create then
            /* creation of an object */
            execute (e);
            Insert (e, L_S);
            /* execute pending operations in Q */
            for e_i in Q_S do
                if id_ei = id_e then
                    execute (e_i);
                    Remove (e_i, Q_S);
                endif
            endfor
        else
            /* object not yet created: defer operation */
            Insert (e,Q_S);
            return;
        endif
    ⊥:              /* object already deleted: log event */
        Insert (e, L_S);
    (id_e, st_e):   /* object exists: execute event */
        if t_e > t_S then
            execute (e);
            Insert (e, L_S);
        else
            Promote (e);
        endif
    endcase
    Update(e);
end /* ReceiveEvents */
```

Figure 1

The ORESTE Algorithm

Let $S = (site_S, S_S, t_S, L_S, Q_S, Obj_S, lastSeq_S, lastClock_S)$ be the current site and $e = (f_e, id_e, t_e, site_e, seq_e)$ be an event. Figure 1 describes the algorithms for sending and receiving events. Figure 2 describes the PROMOTE algorithm that reduces the number of operations to undo-redo. Figure 3 is the updating procedure that ensures that the size of L_S does not grow indefinitely.

The PROMOTE algorithm

The PROMOTE algorithm (Figure 2) is central to the ORESTE algorithm since it reduces the number of operations to undo and redo when an event does not arrive in the correct order. To minimize or even avoid undo-redo, the PROMOTE algorithm relaxes the total ordering of events into a partial order compatible with the semantics of operations. A partial order is defined as an order that gives the same result as the total order.

The total order can be relaxed in three cases: (1) if objects are independent, operations on distinct objects are independent and can thus be applied in any order; (2) if two operations commute, they can be executed in any order. For example, in a drawing program, moving an object and changing its color attribute gives the same result irrespective of the order in which the operations are applied to the object; (3) if an operation is masked by a more recent operation, then the operation does not need to be executed at all. For example, if a site first receives the destruction of an object and then its modification, the destruction masks the modification and the modification does not need to be executed.

The PROMOTE algorithm tries to commute the received event *e* with all events that occurred at a greater logical clock; if an event *e'* does not commute with *e*, it attempts to commute the sequence (*e*, *e'*) with events that

```
procedure Promote (e : Event)
var
    P : SetOfEvents; /* events that do not commute */
begin
    P := ∅;
    for e_i in L where e p e_i do
        if commuteWithSet (e_i, P) then
            if mask (e_i, e) then
                return;  /* don't execute masked event */
            else
                if not commute (e_i, e) then
                    Insert (e_i, P);  /* e_i will be undone */
                endif
            endif
        else
            Insert (e_i, P);  /* e_i will be undone*/
        endif
    endfor
    /* undo-redo sequence */
    /* if P = ∅, there is nothing to undo-redo */
    for e_i in reverse (P) do undo (e_i) endfor
    execute (e);
    Insert (e, L_S);  /* log e in L_S in the correct order */
    for e_i in P do execute (e_i) endfor
end /* Promote */
```

Figure 2

occurred at a greater logical clock. This process is repeated and stops when either the event is masked or all the events have been treated. The algorithm reduces (but does not minimize) the number of operations to undo. In the best cases, either the event is masked and need not be executed at all or it commutes with all more recent events and can be executed without undoing anything.

Reducing the size of L: the Update procedure.

L is used to undo previous events when "old" events arrive. The Update procedure's main purpose is to discard "old" events from the log L. Otherwise the space required by the algorithm would grow linearly with the number of received events. We can discard events in L whose logical clock is such that no older event will ever arrive. We use two arrays for this:

- lastSeq[i] stores the sequence number of the last event received *in sequence* from site i.
- lastClock[i] stores the logical clock of the last event received *in sequence* from site i.

The array lastSeq is necessary unless the order in which events are sent by a site is the same as the order in which they are received by each other site. If this is not the case, it allows a site to detect missing events and events received more than once.

The last step of the algorithm insures that L cannot grow indefinitely. If a site is idle, it is not sending events. Thus the minimum value of lastClock stays the same and the size of L never decreases. Sending "still alive" events avoids this problem. Furthermore, the system can detect sites that have crashed or are unreachable by monitoring "still alive" events.

```
var
    timeOut; /* time limit for a site to be idle */

procedure Update (e : Event)
begin
    /* update local time */
    t_s := max (t_s, t_e + 1);
    /* update lastClock_s and lastSeq_s */
    Let e_i ∈ L_s be the last event received in sequence
        from the sending site site_e
    if seq_ei = lastSeq_s[site_e] + 1 then
        lastSeq_s[site_e] = seq_e;
        lastClock_s[site_e] = t_e
    endif
    /* discard useless events from the log L_s */
    for e_i in L do
        if t_ei ≤ Min (lastClock_s) then
            Remove (e_i, L_s)
        endif
    endfor;
    /* send alive event if inactive for a long time */
    if t_s > lastClock_s(site_s) then
        SendEvent (AliveEvent)
    endif
end /* Update */
```

Figure 3

Implementation issues

Objects are assigned unique ids as follows: each site s holds the number nc_s of objects it has created and a unique site number $site_s$. When site s creates an object, it assigns the id ($site_s \parallel nc_s$) to it. The function Obj that maps ids to objects can be implemented by an associative table containing live objects and a set containing deleted objects.

The set of deleted objects can grow indefinitely. However, a technique similar to the reduction of the log can be applied to this set: a deleted object can be removed from the set if the logical time at which it was deleted is older than the oldest value in lastClock. The function Obj will never be called for an object removed from the set, since no event referencing such an object will ever be received.

We implement the functions commute and mask used in the PROMOTE algorithm by defining the notion of masking and commutability at the level of functions and families instead of events:

let $f_1, f_2 \in \mathcal{F}$ and let $\mathcal{F}_1, \mathcal{F}_2$ be two families,

- Commute (f_1, f_2) iff
  $\forall o \in O, f_1 \circ f_2 (o) = f_2 \circ f_1 (o)$
- Mask (f_1, f_2) iff
  $\forall o \in O, f_1 \circ f_2 (o) = f_2 (o)$
- COMMUTE ($\mathcal{F}_1, \mathcal{F}_2$) iff
  $\forall f_1 \in \mathcal{F}_1, \forall f_2 \in \mathcal{F}_2$, Commute ($f_1, f_2$)
- MASK ($\mathcal{F}_1, \mathcal{F}_2$) iff
  $\forall f_1 \in \mathcal{F}_1, \forall f_2 \in \mathcal{F}_2$, Mask ($f_1, f_2$)

Indeed, families correspond to types of operations, such as change color, move, etc. It is natural to state that change color and move commute, meaning that any change of color on any object commutes with any move of that object. Such properties of the families are stored in a two-dimensional matrix used to implement the functions commute and mask. Note that in the case where every pair of families either commute or mask events never need to be undone: any event will be either masked or executed without undoing anything.

Correctness of the ORESTE algorithm

We must prove that for any given sequence of events exchanged between sites, the document state is the same at each site when the system is quiescent. This can be proven by showing that the execution by a site of any given

sequence of events is equivalent to the execution of this sequence in the total order. This property is ensured by the PROMOTE algorithm. If an event is received late, i.e. if its timestamp is older than the current logical clock, then the set of precedent events is examined and transformed using commuting and masking rules. By definition these rules ensure that the algorithm yields the same result as the execution in the total order.

As an example, we show the execution of the ORESTE algorithm when three sites simultaneously exchange three events (Figure 4). The total order of events is $e_1 \mathrm{p} e_2 \mathrm{p} e_3$. They have the following properties: *mask* (e_3, e_1) and *commute* (e_1, e_2). Figure 5 shows how each site handles the incoming events in an order that is equivalent to the total order.

Open issue

The PROMOTE algorithm reduces the number of operations to undo and redo when a late event arrives but it does not minimize it. This minimization problem can be formalized as follows:

Let $s = e_1 e_2 \ldots e_n$ be a sequence of events and let $\Re_1$ and $\Re_2$ be transformation rules associated with this sequence:

- $\Re_1 = \left\{ e_{i_1} e_{j_1} \rightarrow e_{j_1} e_{i_1}, \ldots, e_{i_k} e_{j_k} \rightarrow e_{j_k} e_{i_k} \right\}$: commutability rules.
- $\Re_2 = \left\{ e_{i_1} e_{j_1} \rightarrow e_{j_1}, \ldots, e_{i_l} e_{j_l} \rightarrow e_{j_l} \right\}$: masking rules.

Let $Right\left(s, a_i\right)$ be the sequence of events to the right of a_i in s. The problem is to find a polynomial algorithm that returns for a given a_i the shortest sequence $Right\left(s, a_i\right)$ by applying the rules in $\Re_1$ and $\Re_2$ to s.

The complexity of the PROMOTE algorithm is polynomial, which makes its implementation efficient. However, it is not optimal. Let us consider the sequence $s = e_1 e_2 e_3$ with the rules *mask* (e_3, e_1) and *commute* (e_1, e_2). PROMOTE does not reduce the sequence although the following rules reduce it to zero:

$$e_1 e_2 e_3 \xrightarrow{mask(e_3, e_2)} e_1 e_3 \xrightarrow{commute(e_1, e_3)} e_3 e_1$$

It is easy to write an exponential algorithm that finds the optimal solution but we do not know whether a polynomial algorithm exists. For our current needs, the PROMOTE algorithm is sufficient since conflicts do not happen often and consist of undoing short sequences of events. But in a situation where a large number of participants modify the same area of a document, a better PROMOTE algorithm would be useful.

Related Work

Various methods have been used to implement concurrency control algorithms in groupware applications. The first is to not implement anything, e.g. BoardNoter [17] and Commune [15] which are bitmap editors for tightly coupled meetings. They do not need to address conflicts since it is improbable that two users will modify the same pixel simultaneously. Other graphic editors, such as Aspects [3], allow joint editing of a structured graphic

Site 1	Site 2	Site 3
e1 is executed locally. ReceiveEvent(e3) e3 is executed. ReceiveEvent(e2) Since e2 and e3 are in conflict, e3 is undone and e2 and e3 are executed, thus the event are finally executed in the total order	e2 is executed locally. ReceiveEvent(e1) Since commute (e1,e2), e1 is executed. ReceiveEvent(e3) e3 is executed. Since we used the rule of commutation, the sequence of events is equivalent to the execution in the total order	e3 is executed locally. ReceiveEvent(e1) Given the masking rule, mask (e3,e1), e1 is not executed. ReceiveEvent(e2) Since e2 and e3 are in conflict, e3 is undone and e2 and e3 are executed, thus the event are finally executed in the total order

Figure 5

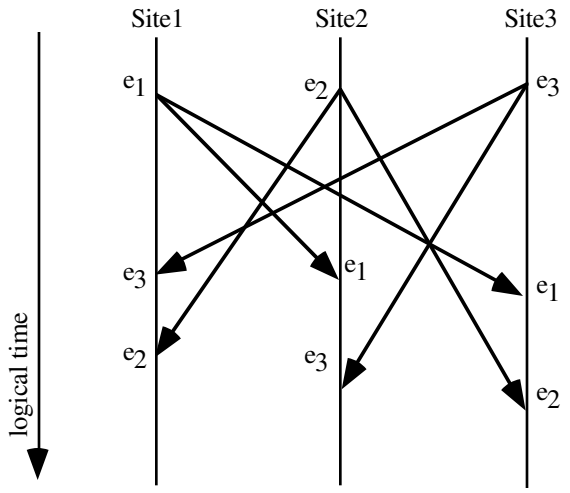

Figure 4

but do not address concurrency control. By not handling conflicts, it is possible to get into an inconsistent state. The chance of conflicts is also higher, since they lack groupware features that help users be aware of each other.

Other groupware applications use a floor control mechanism that allows only one participant to be active at a time. For instance, CaptureLab [10] and Timbuktu [5] use floor control to take over a different computer. MMConf [6] encourages turn-taking floor control; when running an open floor, the events are not guaranteed to arrive in the same order at all sites. This is well-suited to cases where the participants are in the same room or connected via a video link and do not need much parallelism. However, this floor control mechanism would slow down the users of a text editor, since two users could not edit the document simultaneously.

Many algorithms exist for concurrency control in databases [16]. Unfortunately, most results do not apply to real-time groupware: databases are designed to give the illusion of being the only user on the system, whereas groupware systems are designed to make users aware of each other. The most important property of a groupware system, interface response time, is not as important in a distributed database.

If distributed operating systems were widely available, the purely replicated architecture would be easy to implement. For instance the ISIS system [4] is a good candidate for the implementation of replicated systems. It introduces the notion of process groups. Any message sent to a process group is sent to each participant of the group. Several primitives provide different semantics over the delivery of messages. In particular, one semantic is that all sites receive the events in the same order. This simplifies the implementation of the replicas, but one must be aware that such message passing is more expensive than weaker semantics, i.e. sending a message to the process group requires the acknowledge of the receiving sites, thus impairing response time. For our purposes, ISIS has two drawbacks: it runs only on Unix workstations, and it is too powerful for our needs. We seek a lightweight implementation of the replication that fits the precise needs of groupware systems.

Within the domain of real-time multi-user editors, Grove [8,9] is the closest to our system, although it is dedicated to text editing. Grove also uses a replicated architecture, with concurrency control being achieved by a distributed operational transformation algorithm (dOpt) : operations which arrive out of date are transformed so that they can be executed without disrupting the session. If there are n operations, this technique requires n^2 transformation procedures, some of which are not trivial to write whereas our algorithm requires a matrix that holds the operations that commute and mask plus n functions of undo. Also, Grove uses a model based on a text editor and their techniques are not general enough to apply to graphics editor. Finally, to reduce the size of the log, the quiescence is enforced, but this technique might slow down the performance of the system. Our system uses a similar architecture with a simpler concurrency control scheme.

Discussion

We now discuss a set of issues regarding the algorithm and implications for the interface.

Causality between events

The current algorithm does not take into account the causal relation between events [11]. For instance, if a given site sends two events, our algorithm handles them in the order received, which may be the reverse order. The masking and commutability rules define a dependency relation that is less constraining than causality because it takes into account the semantics of the application.

Another issue is how to cope with the simultaneous use of a video link to coordinate tasks. If a user is talking about an object that has not yet been created on another user's computer, it may create confusion. Implementing causality between events might resolve some such cases. We plan to experiment with video links in order to explore solutions to this problem.

Dependence between objects

The independence of objects varies with the application. For example, objects in a drawing tool should probably be independent, but not objects in a text editor. Masking operations within the same family of parameterized functions is generally easy. For example,

ChangeColor (c: Color) defines a family of functions ChangeColor (red), ChangeColor (green), etc. Such functions generally affect a part of the state of the object. Applying two functions from the same family is equivalent to applying the second one: ChangeColor (red) followed by ChangeColor (green) is equivalent to ChangeColor (green). For this to hold, the result of each operation must not depend on the part of the state of the object being modified. For example, MoveBy (10, 10) is not masked by MoveBy (5, 5) because they are relative moves, but MoveTo (50, 50) is masked by MoveTo (100, 100) because they are absolute moves.

Finally, in order for operations in different families to commute, they must manipulate independent subparts of the state of the object. For example, changing the color and moving an object are independent operations and thus commute. However, moving and resizing an object may not be independent. For example, if an object is resized from a corner and its position is determined from the center, resizing will change both the object's size and position. Such dependencies can be avoided by sending two resize events (see next section).

Even though our algorithm is designed to handle independent objects, one need only change the definition of the commute and mask functions to make it handle dependent objects. The basic structure of the algorithm would not change.

Operations vs. commands

Commands issued by the user differ from the actual operations sent via events between sites. As for any interactive system, the command set must be designed based on the task space of the application and its conceptual model. However, the set of operations must be designed based on the optimal performance of the replication algorithm. The optimal situation for this is:

- all objects are independent,
- any pair of operations in the same family $\mathcal{F}_i$ mask each other,
- any pair of operations in two different families $\mathcal{F}_i$ commute.

These properties ensure that any latecoming event can be handled without undoing any previous command, providing the most responsive interactive behavior.

The mapping between operations and commands need not be one to one, especially for "long" commands. In many interactive systems, a command is activated by selecting an object and then selecting an item in a menu. Sometimes, a dialogue box opens in order to specify extra arguments to the command. In this case, the command is actually carried out when the user clicks OK in the dialogue box. Other long commands are issued by direct manipulation, e.g. dragging the mouse. Although the command is actually executed when the "long" interaction finishes, it might be useful to send an operation when the command starts and another operation when it finishes. The start operation serves two purposes: feedback to other users that something is going to happen to the object and partial locking of the object at the other sites. Partial locking prevents other users from starting a command on that object that would conflict with the command in progress. This technique reduces dramatically the likelihood of user conflicts, because each user knows what is being done by others, and not only what has been done by others. This improves the sense of a shared environment.

Mapping one command to several operations can also be used to avoid dependencies. In a previous section, we have shown how to map the resize command to two independent operations (change size and change position).

Conclusion and future work

We have introduced the notion of a purely replicated architecture and presented a concurrency control algorithm for real-time groupware systems. The algorithm optimizes response time for the interface, a critical factor for the success of groupware. Furthermore, by using the semantics of the application, we have shown how to reduce the number of operations to undo when events arrive out of order.

Future work will take three directions. First, we are working on heterogeneous groupware: groupware applications that run on different hardware and software platforms (e.g. XWindow, Macintosh). Second, we plan to implement a multi-user text editor using our algorithm, both to more thoroughly validate our model and to compare our approach with that used to implement Grove [8,9]. It would also allow us to define a more general algorithm and eventually a software layer to be used by groupware applications. Finally, we plan to study session management, e.g. handling of newcomers and storage and retrieval of shared documents in a file system.

Acknowledgments

This work is partially supported by Apple France. We thank MetaSoftware for providing us with MetaDesign and Design/OA and Heather Sacco and Wendy Mackay for enhancing the readability of this article.

References

1. Apple Computer, Inside Macintosh, Volume VI, Addison Wesley, Reading, MA, 1991.

2. Beaudouin-Lafom, M., Karsenty, A., Transparency and Awareness in a Real-Time Groupware System. In *Proc. ACM Symposium on User Interface Software and Technology UIST'92* (Monterey, CA, November 1992).
3. von Biel, V., Groupware Grows Up. In *MacUser*, June 1991, pp. 207-211.
4. Birman, K., Cooper, R., Joseph, T., Kane, and K., Schmuck, F., The ISIS System Manual, June 1989.
5. Coleman, Dale, Timbuktu vs. Carbon Copy Mac: Close Race. In *MacWeek* (September 11, 1990), pp. 181-188.
6. Crowley, T., Milazzo, P., Baker, E., Forsdick, H., and Tomlinson, R., MMConf: An Infrastructure for Building Shared Multimedia Applications. In *Proc. Third Conference on Computer-Supported Coopera-tive Work* (Los Angeles, CA., October 1990) ACM, New York, 1990.
7. Dewan, P., Choudhary, R., Flexible User Interface Coupling in Collaborative Systems. In *Proc. Human Factors in Computer Systems CHI'91* (New Orleans, LA, April 1991), pp. 41-49.
8. Ellis, C.A., and Gibbs, S.J., Concurrency Control in Groupware Systems. In *Proc. ACM SIGMOD'89 Conference on the Management of Data*, (Seattle WA, May 1989) ACM, New York, 1990.
9. Ellis, C.A., Gibbs, S.J., and Rein, G.L., Groupware Some Issues and Experiences. In *Communications of the ACM*, January 1991, 34 (1), pp. 39-58.
10. Elwart-Keys, M., Halonen, D., Horton, M., Kass, R., and Scott, P., User Interface Requirements for Face to Face Groupware. In *Proc. Human Factors in Computer Systems CHI'90* (Seattle, WA, April 1990), pp. 303-312. ACM, New York, 1990.
11. Fidge, C., "Logical Time in Distributed Computing Systems" *IEEE Computer*, August 1991.
12. Karsenty, A, Tronche, C., Beaudouin-Lafon, M., GroupDesign: Shared Editing in a Heterogeneous Environment, *Usenix Computing Systems*, to appear, 1993.
13. Lamport, L., Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM*, July 1978, 21 (7), pp. 558-565.
14. Meta Software Corporation, *Design/OA Manual*, 150 CambridgePark Drive, Cambridge, MA, March 1989.
15. Minneman, S. L., and Bly, S. A., Managing a Trois: a Study of a Multi-User Drawing Tool in Distributed Design Work. In *Proc. Human Factors in Computer Systems CHI'91* (New Orleans, LA, April 1991), pp. 217-224.
16. Son, S. H., Replicated Data Management in Destributed Database Systems. In *SIGMOD Record*, 17(4), December 1988.
17. Stefik, M., Foster, G., Bobrow, D. G., Keneth, K., Lanning, S., and Suchman, L., Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings. In *Communications of the ACM*, January 1987, 30 (1), pp. 32-47.