

# Copies convergence in a distributed real-time collaborative environment

Nicolas VIDOT<sup>1</sup>, Michelle CART<sup>1</sup>, Jean FERRIÉ<sup>1</sup>, Maher SULEIMAN<sup>2</sup>

<sup>1</sup>LIRMM, University of Montpellier II  
161, rue Ada, 34092 Montpellier (France)  
e-mail: {nvidot, cart, ferrie}@lirmm.fr

<sup>2</sup>ISSAT  
BP 31983, Damas (Syria)

## ABSTRACT

In real-time collaborative systems, replicated objects, shared by users, are subject to concurrency constraints. In order to satisfy these, various algorithms, qualified as optimistic, [3, 5, 13, 17, 14, 15, 18], have been proposed that exploit the semantic properties of operations to serialize concurrent operations and achieve copy convergence of replicated objects. Their drawback is that they either require a condition on user's operations which is hard to verify when possible to ensure, or they need undoing then redoing operations in some situations. The main purpose of this paper is to present two new algorithms that overcome these drawbacks. They are based upon the implementation of a continuous global order which enables that condition to be released, and simplifies the operation integration process. In the second algorithm, thanks to deferred broadcast of operations to other sites, this process becomes even more simplified.

## INTRODUCTION

The purpose of a collaborative system is to facilitate team working, and in particular to enable the manipulation of shared objects by members of a team whilst making them evolve in a coherent way. Usually, a shared object involved in a collaborative activity (shared text edition, shared CAD, electronic conferences, etc.) is subject to concurrent accesses and real-time constraints. The real-time aspect necessitates every user seeing the effects of his own actions on the object immediately, and the effects resulting from the actions of other users as soon as possible. In a distributed system when assuming a non negligible network latency, this high reactivity cannot be achieved without each object being replicated on every site. Consequently, the problem is to conciliate both real-time constraint and consistency preservation of object copies, as they can be modified concurrently by many users. In particular, concurrency control must not use a blocking protocol.

In this context, various algorithms, dOPT [3], ORESTE [5], adOPTed [13], GOT [17], SOCT2 [14], GOTO [18], that exploit the semantic properties of the operations, have

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CSCW'00, December 2–6, 2000, Philadelphia, PA.

Copyright 2000 ACM 1-58113-222-0/00/0012...\$5.00.

been proposed to serialize concurrent operations and thus ensure the convergence of all copies of an object. More precisely, dOPT, adOPTed, GOT, SOCT2 and GOTO exploit a transposition function to transform an operation before integrating it into the history associated with an object copy so as to respect users intentions. The histories associated with the copies of an object are ensured to be equivalent (i.e. resulting in the same final state) though the order of the concurrent operations might be different. In adOPTed, SOCT2 and GOTO the transposition function needs conforming to a condition (named C2) which is hard to verify, and not always possible to ensure. In GOT this condition is relaxed, but it is necessary under some circumstances to undo and then redo some operations.

Given this, our aim is to conceive algorithms which overcome these limitations. The purpose is to ensure copies convergence while respecting the user's intention without either imposing condition C2 on the transformation or needing to undo then redo some operations.

The paper is developed as follows. First we describe the model used as well as the problems raised when considering the consistency of the copies of an object in a distributed collaborative environment. Then we review the problem of operation integration in the history associated with a copy. Afterwards we detail the principles of the two proposed algorithms called SOCT3 and SOCT4. These algorithms are based upon the implementation of a continuous global order which enables condition C2 to be released, and the operation integration process to be simplified. Finally, we compare the existing algorithms and give an overview of the implemented techniques.

## GENERAL PROBLEMS

A distributed collaborative system is constituted from a set of sites interconnected by a supposed reliable network. Each object (i.e. text, graphics, ...) shared by the users is replicated so that a copy of the object exists on every site. Each object can be handled using definite operations. In order to maintain copies consistency, every operation generated and executed on a site must be executed on all other copies as well. This requires every *generated* operation to be *broadcast* to the other sites; after *reception* on a site the operation is *executed* on the local copy of the object. Given a site, a *local* operation is an operation generated on this site whereas a *remote* operation is one that has been generated on another site. In order to guarantee users a minimum response time, operations generated on a site (i.e. local ones) are executed immediately on this site.

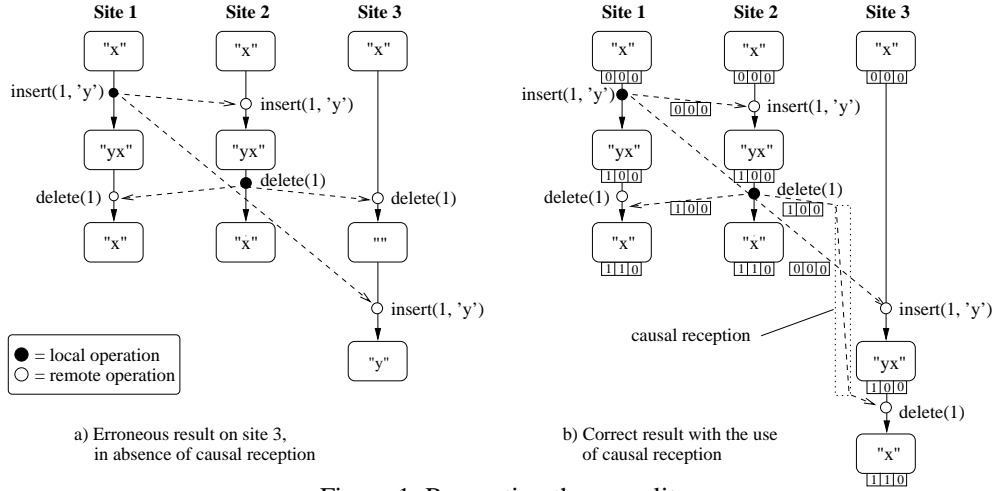


Figure 1: Respecting the causality

This section reviews the three problems encountered when trying to achieve consistency maintenance of object copies, namely: (1) causality preservation, (2) user intention preservation and (3) convergence. A collaborative text editor will be used as an example. Let us assume a text is an ordered collection of sentences, each one being an object represented by a string of characters. The operations defined on this object are:

- $\text{insert}(p, c)$ : inserts character  $c$  at position  $p$  in the string,
- $\text{delete}(p)$ : deletes character at position  $p$  in the string.

In the following, we suppose that users work concurrently and modify the same sentence.

### Causality Preservation

The operation  $op_1$  is said to *causally precede*  $op_2$  (noted  $op_1 \text{ precedes}_c op_2$ ) iff  $op_2$  has been generated on a site after  $op_1$  was executed on this site. Consequently,  $op_2$  is supposed to depend on the effects of operation  $op_1$ . The problem is then to execute the operations broadcast on every site without violating the causality precedence.

Figure 1a depicts an example where causality precedence is not respected. Initially all the copies hold the value "x". After reception and execution of the operation  $\text{insert}(1, 'y')$  generated by the user on site 1, the user on site 2 generates the operation  $\text{delete}(1)$ . In this case  $\text{insert}(1, 'y')$  *precedes*<sub>c</sub>  $\text{delete}(1)$ . If these operations were to be delivered and executed in a different order on another site (e.g. site 3), then their effects might produce an incoherent copy: thus the need to respect causal precedence on all sites. To summarize, given that  $op_1 \text{ precedes}_c op_2$  then on each site the execution of  $op_1$  must precede the execution of  $op_2$ .

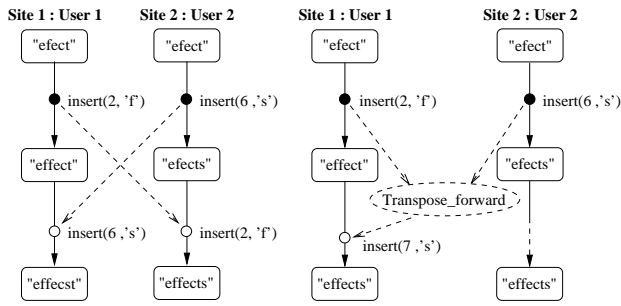
All existing algorithms, dOPT [3], adOPTed [13], GOT [17], SOCT2 [14, 15], GOTO [18] achieve causality-preservation by the use of a state vector  $V_s$  associated with each object at each site  $S$ . A state vector is a variant of the clock vector [8]. More precisely, a state vector component  $V_s[j]$  (with  $1 \leq j \leq N$ ,  $N$  being the number of sites) holds the value corresponding to the number of operations generated by site  $j$ , received and executed by site

$S$ . Each operation  $op$  generated by site  $S_{op}$  is timestamped with the value of the state vector associated to the object on that site, before it is broadcast. This timestamp is noted  $V_{op}$ . When received on site  $S$ , the operation is not delivered unless all the causally preceding operations have been received, that is to say, when the site's state vector is such that  $V_s[i] \geq V_{op}[i]$ , ( $\forall i, 1 \leq i \leq N$ ). As shown on Figure 1b, it is the case of  $\text{delete}(1)$  which is delayed until  $\text{insert}(1, 'y')$  has been received. Finally, given two operations  $op_i$  and  $op_j$ , one can deduce that  $op_i \text{ precedes}_c op_j$  iff their state vectors verify  $V_{op_i}[S_{op_i}] < V_{op_j}[S_{op_j}]$ .

### User Intention Preservation

Operations that are not causally related are said to be concurrent. More accurately  $op_1$  and  $op_2$  are concurrent iff neither ( $op_1 \text{ precedes}_c op_2$ ) nor ( $op_2 \text{ precedes}_c op_1$ ). In this case, neither one depends on the effects of the other. Thus they can be executed in any order on the different sites. Nevertheless, if a site executes  $op_1$  before  $op_2$ , it must take into account the changes made by  $op_1$  when it executes  $op_2$  in order to respect the *intention* of the user who generated  $op_2$ . In the example of Figure 2a, two users work simultaneously on the same object whose state is "effect". The intention of user 1 is to add a 'f' to get "effectf"; this is realized by operation  $\text{insert}(2, 'f')$ . The intention of user 2 is to add a 's' at the end of the word; this is realized by the operation  $\text{insert}(6, 's')$ . When this operation is delivered and executed on site 1, the new state is "effectcs" which is not what user 2 expected. To respect his intention, operation  $\text{insert}(6, 's')$  needs to be transformed. Thus  $\text{insert}(7, 's')$  should be executed instead of  $\text{insert}(6, 's')$ .

The problem described here is due to the fact that the operation which realizes the user intention is relative to a specific state of the object. Should a concurrent operation be executed before it, the object will not be in the required state to execute that operation without violating the user intention. The solution is to transform the operation to be executed to take into account the modifications made by all the concurrent operations serialized before it. This transformation is possible provided that a function specific to the semantics of the operations is defined which gives



a) Non-respect of user 2 intention, on site 1      b) Use of the forward transposition to ensure the respect of user 2 intention, on site 1

Figure 2: Respecting the intention of the user

for all pairs of operations  $(op_1, op_2)$  an operation noted  $op_2^{op_1}$ , defined on the state resulting from the execution of  $op_1$  and realizing the same intention as  $op_2$ . This transformation function introduced in dOPT [3] is also used in [10], adOPTed [13], GOT [17], SOCT2 [14] and GOTO [18] under various denominations. We call it *forward transposition* [14, 15]. Let  $O_i$  be an object state,  $O_i.op$  is the state obtained after the execution of  $op$  and  $\text{Intention}(op, O_i)$  is the intention which is realized by operation  $op$  on object state  $O_i$ . The forward transposition is then formally defined as follows:

$$\text{Transpose\_forward}(op_1, op_2) = op_2^{op_1}, \text{ with:} \\ \forall O_i, \text{Intention}(op_2^{op_1}, O_i.op_1) = \text{Intention}(op_2, O_i).$$

More generally, let  $seq_n$  be a sequence of  $n$  operations; the forward transposition of operation  $op$  with  $seq_n$ , noted  $op^{seq_n}$ , is defined recursively by :  $op^{seq_n} = \text{Transpose\_forward}(op_n, op^{seq_{n-1}})$  with  $seq_n = op_1.op_2\dots.op_n = seq_{n-1}.op_n$  and  $op^{seq_0} = op$ , where  $op.op_j$  represents the execution of  $op_j$  followed by the execution of  $op$ .

It is important to note that the forward transposition requires both operations to be defined on the same state of the object. If this is not the case, the preservation of user intention cannot be guaranteed. This is of particular importance in the case of *partial concurrency* [1], when an operation is concurrent to a sequence of operations. As depicted in Figure 3a,  $op_1$  causally precedes  $op_2$  while  $op_3$  and  $op_1$  are concurrent and both defined on the same state "telephone", but though  $op_3$  is concurrent to  $op_2$ , they are not defined on the same state. This is a typical case of partial concurrency. On site 1, when  $op_3 = \text{delete}(5)$  is received it is forward transposed successively with  $op_1$  and  $op_2$  which gives operation  $op_3^{op_1.op_2} = \text{delete}(7)$  whose execution gives "telephone". On site 2, the forward transposition of  $op_1$  with  $op_3$  gives  $op_1^{op_3} = \text{insert}(5, 'p')$ , but the forward transposition of  $op_2$  with  $op_3$  gives  $op_2^{op_3} = \text{insert}(5, 'h')$  whose execution leads to the incorrect state "telephone" which violates the intention of user 1.

Different solutions have been proposed to apply forward transposition in the right way. In GOT [17] operation  $op_2$  is transformed using the reverse function of forward transposition (called Exclusion\_Transformation), so that  $op_2$  be defined on the same state as  $op_3$  enabling the use of forward transposition. In adOPTed [13] several equivalent

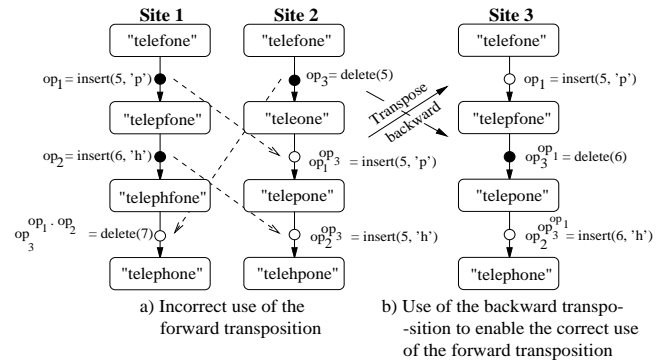


Figure 3: Partial concurrency between operations

histories respecting the causal order are kept on each site, which permits the intermediate states of the object to be retrieved on each site. In SOCT2 [14] and GOTO [18] a new transformation is defined. This function [11] we call *backward transposition* makes it possible to change the execution order of a pair of operations while respecting the user intention. More accurately, the backward transposition of a couple of operations  $(op_1, op_2)$ , executed in this order, gives as a result the couple  $(op_2', op_1')$  which corresponds to their execution in reverse order that leads to the same state, and is compatible with the forward transposition. Formally:

$$\text{Transpose\_backward}(op_1, op_2) = (op_2', op_1'), \text{ with:} \\ op_2' = \text{Transpose\_forward}(op_1, op_2') \text{ and} \\ op_1' = \text{Transpose\_forward}(op_2', op_1').$$

In Figure 3b, when applying backward transposition to the couple  $(op_3, op_1^{op_3})$  we obtain the couple  $(op_1, op_3^{op_1})$ , that is to say the operations  $(\text{insert}(5, 'p'), \text{delete}(6))$ . Operations  $op_2$  and  $op_3^{op_1}$  are now defined on the same state and thus forward transposition of  $op_2$  with  $op_3^{op_1}$  is allowed and gives operation  $\text{insert}(6, 'h')$  whose execution leads to the expected result.

### Copies Convergence

Taking into account causality as well as user intention is not sufficient to achieve executions that guarantee in all cases the convergence of the copies on all sites. In addition, forward transposition is required to verify two conditions [3, 13]. Indeed, consider two concurrent operations  $op_1$  and  $op_2$ , generated on the same state and executed, after being forward transposed, in a different order on two different sites (Fig. 4). The execution of  $op_1$  followed by the execution of  $op_2^{op_1}$  on site 1 must produce the same results as the execution of  $op_2$  followed by the execution of  $op_1^{op_2}$  on site 2. This condition called C1 is formally defined as:

**Condition C1:** Let  $op_1$  and  $op_2$  be two concurrent operations defined on the same state. The forward transposition verify C1 iff:

$$op_1.op_2^{op_1} \equiv op_2.op_1^{op_2}$$

where  $\equiv$  denotes the equivalence of states obtained after applying both sequences from the same state.

Let now consider an operation  $op_3$  generated on site 3 which is concurrent to  $op_1$  and  $op_2$ . It is transposed in turn when received on sites 1 and 2. To guarantee the convergence of all the copies, the transformation of an operation with a sequence of two (or more) concurrent operations must not depend on the order used to serialize these operations. For that, in addition to condition C1, the forward transposition must verify condition C2.

**Condition C2:** Whatever operations  $op_1$ ,  $op_2$  and  $op_3$  are, the forward transposition verify C2 iff:

$$op_3^{op_1:op_2} = op_3^{op_2:op_1}$$

where  $op_i:op_j$  denotes  $op_i.op_j^{op_i}$ .

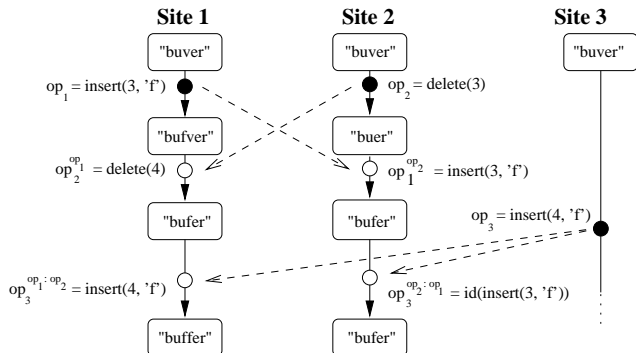


Figure 4: Convergence of the copies

In the example of Figure 4, forward transposition does not verify condition C2, since the forward transposition of  $op_3$  with the sequences  $op_1.op_2^{op_1}$  and  $op_2.op_1^{op_2}$  differs on site 1 and site 2. Conditions C1 and C2 are required in adOPTed [13], SOCT2 [14] and GOTO [18]. Only GOT [17] frees itself from these two conditions by imposing an unique serialization order on all sites. This order, complying with the causal order, makes it necessary to Undo/Redo some operations. In dOPT [3], condition C2 is not required to the detriment of copies convergence.

In this context, our aim is to propose a new solution to convergence of copies which avoids the constraint associated with the verification of condition C2, while guaranteeing that no operation will need to be undone then re-done. Two new algorithms meeting these requirements are presented below.

### OPERATION INTEGRATION

This section presents a framework for describing the various algorithms, and emphasizes the difficulty related to the integration of an operation in the history of a copy. We note that providing the user with the possibility to undo an operation already executed is not in the scope of this paper.

#### Framework Components

Three major functionalities are implemented on each site. They each correspond to a procedure, namely *Local\_Execution*, *Causal\_Reception* and *Integration*.

The *Local\_Execution* procedure is executed consecutively to the generation of an operation on the site. It contains

the local execution of the operation (satisfying in this way the immediate execution constraint) and its broadcast to all the sites including the generator site. The message broadcast for an operation is a triplet  $\langle op, S_{op}, V_{op} \rangle$  where  $op$  designates the operation,  $S_{op}$  the site which generated it, and  $V_{op}$  the timestamp associated with  $op$ .

The *Causal\_Reception* procedure is executed when an operation, either local or remote, is received. Its role is to handle the delivery of operations to the *Integration* procedure with respect to the causal order. For site  $S$ , it corresponds to the following algorithm:

```

procedure Causal_Reception ( $\langle op, S_{op}, V_{op} \rangle$ ) {
  wait_until  $V_S[i] \geq V_{op}[i], (\forall i : 1 \leq i \leq N)$  ;
  deliver ( $\langle op, S_{op}, V_{op} \rangle$ ) ;
  /*delivery of the operation to the Integration procedure*/
   $V_S[S_{op}] := V_S[S_{op}] + 1$  ;
}

```

This procedure is proved [14] to ensure that if  $op_1$  precedes<sub>C</sub>  $op_2$  then  $op_1$  will be delivered before  $op_2$ . Notice that the delivery order of concurrent operations might differ on distinct sites.

*Integration* procedure is called when a local or remote operation is delivered. It performs the local execution of a remote operation (a local operation has already been executed). To take into account the problems evoked earlier on (intention preservation and copies convergence), a remote operation needs to be transformed prior to its execution on the current state of the local copy. This task holds all the difficulties, and this is where the differences among the various algorithms reside.

#### Integration of a Remote Operation

The integration of an operation on site  $S$  aims at enabling its execution on the current state of the copy on  $S$ . It consists in obtaining, by using the history on site  $S$ , the operation whose execution on the current object state realizes the same intention as the operation generated on the remote site.

**Definition:** The history of object  $O$  on site  $S$ , noted  $H_{S,O}(n)$ , is a sequence of  $n$  operations executed on the copy of  $O$  on the site  $S$ , that transformed it from its initial state to its current state.

To simplify notations and without loss of generality, we assume there is only one object. Consequently  $H_{S,O}(n)$  is noted  $H_S(n)$  with  $H_S(n) = op_1.op_2...op_i...op_n$ . The operations order in  $H_S(n)$  is called *serialization order*. Any local operation  $op$  executed on  $S$  is appended to the history of the site (Fig. 5), so in this case,  $H_S(n+1) = H_S(n).op_{n+1} = H_S(n).op$ .

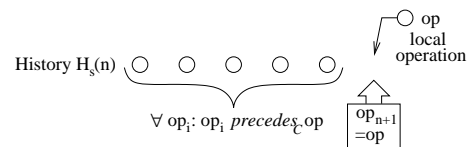


Figure 5: Execution of a local operation

The problem of the integration of a remote operation is depicted in Figure 6a. Thanks to causal delivery,  $H_S(n)$  contains *all* the operations which causally precede  $op$ .

However, it also contains operations that are concurrent to  $op$ . Thus the objective is to obtain, given  $op$  and  $H_S(n)$ , an operation  $op_{n+1}$  whose execution realizes the same intention as  $op$ , knowing that causally preceding and concurrent operations are mixed together in  $H_S(n)$ . In its principle, the integration of an operation  $op$  is composed of two steps (Fig. 6b):

**Step 1:** consists in reordering history  $H_S(n)$  to get an equivalent history where all the operations causally preceding  $op$  precede the operations that are concurrent to  $op$ . The border between these two sequences corresponding to the state on which  $op$  is defined, the partial concurrency problem is then naturally solved.

**Step 2:** consists in transforming  $op$  in order to take into account the concurrent operations already executed.

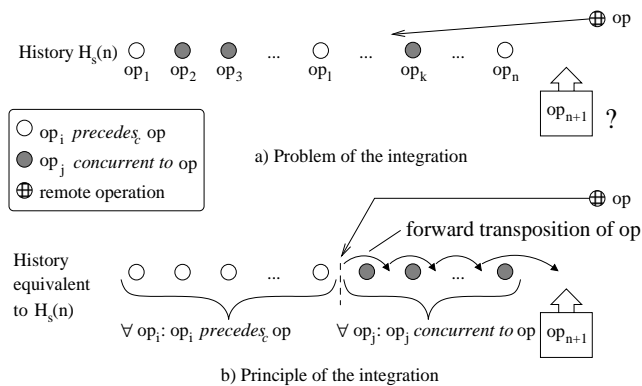


Figure 6: Integration of a remote operation

As concurrent operations may be delivered in different orders on different sites, each site builds its own serialization order based on the delivery order while insuring it is equivalent to the serialization order of the other sites. The algorithm SOCT2 [14] precisely follows the above mentioned schema. Backward transposition is used to reorder the history. Then concurrent operations are taken into account using the forward transposition. The last section gives a comparison of the other existing algorithms.

### Limits of Existing Algorithms: Undo/Redo or Condition C2

Condition C2 ensures the transposition of an operation with a sequence is independent of the order the operations in this sequence are themselves transposed. Verifying this condition is not trivial. With the operations presented in Figure 4 condition C2 is not satisfied. Considering the triplet ( $op_1 = \text{insert}(3, 'f')$ ,  $op_2 = \text{delete}(3)$ ,  $op_3 = \text{insert}(4, 'f')$ ), one cannot achieve equality of  $op_3^{op_1:op_2}$  and  $op_3^{op_2:op_1}$  unless additional parameters are added to the operations signature [15]. Doing so, however, increases the complexity of the verification of C2, especially when the set of operations is large. The set of operations {insert, delete} alone leads to no less than a hundred cases to be checked depending on the various parameters [16]. Moreover, in some cases, condition C2 cannot be satisfied at all and consequently the convergence of the copies cannot be guaranteed. As a result, it is worth trying to get rid of this condition. This supposes that concurrent operations can be

ordered in the same way on all sites. The definition and the use of a global serialization order (noted  $precedes_s$ ) respecting the causal order makes it possible.

In GOT[17], a global serialization order is defined by the sum of the state vector components and in case of equality by a predefined priority on the sites. More precisely, given two messages  $\langle op_1, S_1, V_1 \rangle$  and  $\langle op_2, S_2, V_2 \rangle$ ,  $op_1 precedes_s op_2$  iff  $(\text{sum}(V_1) < \text{sum}(V_2))$  or  $(\text{sum}(V_1) = \text{sum}(V_2) \text{ and } S_1 < S_2)$ . However, concurrent operations are delivered in an order that, although it respects the causal order, does not correspond to this global serialization order. Thus an Undo/Redo schema must be employed: when an operation is received, all the operations following it in the global serialization order but already integrated in the history have to be undone. Next, the received operation is executed and the undone operations are reintegrated in the history.

In the following, we propose the implementation of a global serialization order such that the operations can be delivered in this order. This way condition C2 can be abandoned without having to Undo/Redo any operation.

### CONTINUOUS GLOBAL ORDER AND IMMEDIATE BROADCAST: THE SOCT3 ALGORITHM

Suppression of condition C2 requires the use of an unique global order  $precedes_s$  compatible with the causal order  $precedes_c$ . Moreover, in order to avoid to undo/redo operations the order of operations delivery must be consistent with the  $precedes_s$  order. We propose to satisfy both constraints by using a sequencer to obtain a global and continuous order.

#### Local Execution, Broadcast and Reception of Operations in SOCT3<sup>1</sup>

A sequencer [12] is an object which delivers continuously growing positive integer values, called timestamps. A timestamp is obtained through a call to function *Ticket*. The various methods of implementing a sequencer in a distributed system, namely circulating sequencer [7] or replicated sequencer [2] will not be discussed in this paper. Thanks to the *Ticket* function of the sequencer, each operation generated in the collaborative system is assigned a timestamp. The  $precedes_s$  order follows the order of the timestamps and we show below that it is compatible with the causal order  $precedes_c$ .

To respect the real-time constraint (i.e. immediate execution), a local operation is executed before the *Ticket* function is called. To be more precise, an operation generated on site  $S$  is executed without delay and the quadruplet  $\langle op, S_{op}, V_{op}, -1 \rangle$  is appended to the site history. The value of  $(-1)$  denotes that the operation has not yet been assigned a timestamp. Next, the call to function *Ticket* returns  $N_{op}$ , the timestamp which will be assigned to  $op$ .

The message broadcast for an operation is the quadruplet  $\langle op, S_{op}, V_{op}, N_{op} \rangle$ ,  $op$  being the generated operation,  $S_{op}$

<sup>1</sup> Acronym of: Sériatisation des Opérations Concurrentes par Transposition (in French).

the site which generated  $op$ ,  $V_{op}$  the state vector associated with  $op$  and  $N_{op}$  the timestamp assigned to  $op$ . The reception procedure ensures a sequential delivery of operations with respect to the ascending order of the timestamps. Upon receiving an operation the reception procedure delays its delivery until all the operations with lower timestamps have been received and delivered. The state vector is of no use for the reception procedure, but it enables to determine which operations are concurrent to  $op$  during integration step.

The reception procedure, now called *Sequential Reception*, along with the *Local Execution* procedure are the followings:

*Notations*

$S$  : local site  
 $V_S$  : state vector of site  $S$   
 $op$  : received operation  
 $S_{op}$  : site which generated  $op$   
 $V_{op}$  : state vector associated with  $op$   
 $N_{op}$  : timestamp associated with  $op$   
 $N_S$  : timestamp of the last delivered operation on site  $S$   
 $H$  : representation of  $H_S(n) = \text{table of } n \text{ items in the form } \langle op_i, S_{op_i}, V_{op_i}, N_{op_i} \rangle, H[i].operation, H[i].site, H[i].vector, H[i].timestamp \text{ being the fields of } H[i]$   
 $n$  : number of items in  $H$

**procedure Local Execution** ( $op$ ) {

/\*Execution and broadcast of a locally generated operation\*/  
execute ( $op$ ) ; /\*execution on the current state of the object\*/

$V_{op} = V_S$  ;  
 $n := n + 1$  ;  
 $H[n] := \langle op, S, V_{op}, -1 \rangle$  ;  
 $V_S[S] = V_S[S] + 1$  ;

$N_{op} = Ticket()$  ;  
 $H[n].timestamp := N_{op}$  ;  
broadcast( $\langle op, S, V_{op}, N_{op} \rangle$ ) ;

/\*immediate broadcast to all the sites including  $S^*$ / }

**procedure Sequential Reception** ( $\langle op, S_{op}, V_{op}, N_{op} \rangle$ ) {

wait until ( $N_S = N_{op} - 1$ ) ;  
deliver ( $\langle op, S_{op}, V_{op}, N_{op} \rangle$ ) ;  
 $N_S := N_S + 1$  ; }

The  $precede_S$  order (i.e. timestamps order) is compatible with the causal order  $precede_C$  because, if  $op_1$   $precedes_C$   $op_2$  then  $N_{op_1} < N_{op_2}$ . Indeed, if  $op_1$   $precedes_C$   $op_2$  then the execution of  $op_1$  was performed before the generation of  $op_2$ . The call to *Ticket* for  $op_1$  preceded the one for  $op_2$ , so  $op_1$  was assigned a lower timestamp than  $op_2$ ; thus  $op_1$  will be delivered before  $op_2$  on each site.

### Principle of Operation Integration in SOCT3

The integration of a remote operation follows the previously presented schema. The specificity of the current algorithm is due to the fact that each site maintains a history (equivalent to the real history of the site) noted  $H_S(n)$  in which operations are ordered according to the value of their timestamp.

Formally, let  $H_S(n) = op_1.op_2...op_i.op_{L_1}.op_{L_2}...op_{L_m}$  be the history of the object on site  $S$ .  $H_S(n)$  is such that:

i) for  $(1 \leq j \leq i)$ ,  $op_j$  is an operation, either local or remote,

delivered by the sequential reception procedure; so  $op_1, op_2, \dots, op_i$  have continuous timestamps.

ii) for  $(1 \leq k \leq m)$ ,  $op_{L_k}$  is a local operation executed but not yet delivered by the sequential reception procedure;  $op_{L_1}, op_{L_2}, \dots, op_{L_m}$  may have discontinuous timestamps.

iii)  $n = m + i$ .

As timestamps are assigned to the operations according to the order in which they are generated, the local operations ( $op_{L_1}, op_{L_2}, \dots, op_{L_m}$ ) timestamps verify:  $i < L_1 < \dots < L_m$ .

When operation  $op_{i+1}$  with timestamp  $(i+1)$  is delivered by the reception procedure, two cases are possible:

1)  $op_{i+1}$  is the local operation  $op_{L_1}$ , already executed, so no additional computation is needed;

2)  $op_{i+1}$  is a remote operation from site  $S'$ ;  $op_{i+1}$  needs to be integrated in  $H_S(n)$ , that means:

- determine and execute, on the current state, the operation that realizes the same intention as  $op_{i+1}$ ,
- reorder the resulting history to conform to the ascending order of the operation timestamps.

To find the operation to be executed that realizes the same intention as  $op_{i+1}$ , we apply the principle presented in the previous section. Operations of history  $H_S(n)$  are first re-ordered, using backward transposition, into two sequences of operations  $seq_1$  and  $seq_2$  such that:

- $seq_1$  contains all the operations that  $precede_C$   $op_{i+1}$ ,
- $seq_2$  contains all the operations concurrent to  $op_{i+1}$ ,
- $H_S(n)$  is equivalent by transposition<sup>2</sup> to  $seq_1.seq_2$ , which is noted  $H_S(n) \equiv_T seq_1.seq_2$ .

Thus, the operation to be executed to realize the intention of  $op_{i+1}$  is  $op_{i+1}^{seq_2}$ , that is to say the forward transposition of  $op_{i+1}$  with  $seq_2$ . We obtain:  $H_S(n+1) \equiv_T$

$$H_S(n).op_{i+1}^{seq_2} = op_1.op_2...op_i.op_{L_1}.op_{L_2}...op_{L_m}.op_{i+1}^{seq_2}$$

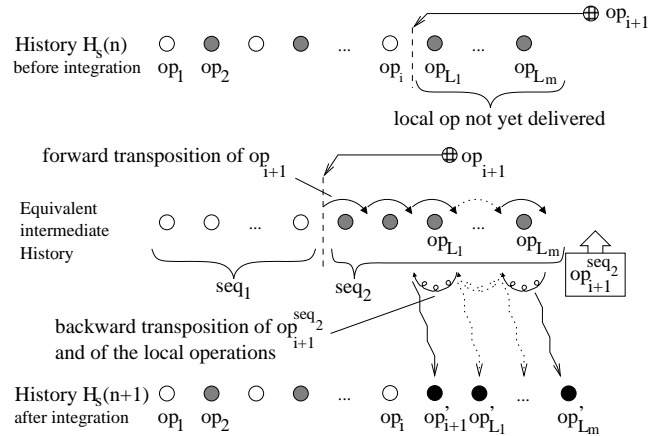


Figure 7: Integration of an operation in SOCT3

For  $H_S(n+1)$  to be well ordered,  $op_{i+1}^{seq_2}$  needs to be put at the right place according to its timestamp. This is done using the backward transposition of  $op_{i+1}^{seq_2}$  with the se-

<sup>2</sup>  $H_1$  is said to be *equivalent by transposition* to  $H_2$ , noted  $H_1 \equiv_T H_2$ , iff  $H_2$  can be obtained from  $H_1$  by applying the backward transposition to the operations.

quence of local operations. The resulting well ordered history:  $H_S(n+1) = op_1.op_2...op_i.op'_{i+1}.op'_{L_1}.op'_{L_2}...op'_{L_m}$  is identical to the one we would obtain if the operations were executed in timestamp ascending order. The proof is given in [16] along with the details of operations  $op'_{i+1}$ ,  $op'_{L_1}$ ,  $op'_{L_2}$ , ...,  $op'_{L_m}$ . The different steps in the integration of a remote operation are summarized in Figure 7.

### The SOCT3 Algorithm

The procedure *Integration* is called when an operation is delivered by the *Sequential\_Reception* procedure. It uses the two following procedures directly taken from SOCT2:

- the *Transpose\_Backward* procedure that uses the *Transpose\_backward* function to realize the backward transposition of two operations in history  $H_S(n)$ ,
- the *Separate* function that rearranges the history  $H_S(n)$  in two sequences  $seq_1$  and  $seq_2$  and returns the length  $n_1$  of  $seq_1$ . This function is applied on a copy of the history  $H_S(n)$  so that the original remains ordered according to the operations timestamps.

Notations remain unchanged. The proof of correctness of SOCT3 is given in [16].

```

procedure Transpose_Backward(j) {
  /*Backward transpose the jth operation and the (j-1)th operation*/
  <opj, Sopj, Vopj, Nopj> := H[j] ;
  <opj-1, Sopj-1, Vopj-1, Nopj-1> := H[j-1] ;
  (opj, opj-1) = Transpose_backward(opj-1, opj) ;
  H[j] := <opj-1, Sopj-1, Vopj-1, Nopj-1> ;
  H[j-1] := <opj, Sopj, Vopj, Nopj> ; }

```

```

function Separate (H, <op, Sop, Vop, Nop>) : integer {
  /*Rearrange the history H and return n1 such that:
  for 1 ≤ i ≤ n1, H[i] precedesc <op, Sop, Vop, Nop> and
  for n1 < i ≤ n, H[i] is concurrent to <op, Sop, Vop, Nop>*/
  n1 := 0 ;
  for i := 1 up to n do
    <opi, Sopi, Vopi, Nopi> := H[i] ;
    if Vopi[Sopi] < Vop[Sop] then /*opi precedesc op*/
      for j := i down to n1+2 do
        /*backward transpose opi down to seq1*/
        Transpose_Backward (j) ;
      endfor ;
      n1 := n1 + 1 ;
    endif ;
  endfor ; }

```

```

procedure Integration (<op, Sop, Vop, Nop>) {
  if S ≠ Sop then /*op is a remote operation*/
    H' := H ; /*copy history H in H'*/
    n1 := Separate (H', <op, Sop, Vop, Nop>) ;
    for i := n1 + 1 up to n do /*forward transpose op with seq2*/
      op := Transpose_forward(H'[i].operation, op) ;
    endfor ;
    execute (op) ;
    VS[Sop] = VS[Sop] + 1 ;
    H[n+1] := <op, Sop, Vop, Nop> ;
    n := n + 1 ;
    for j := n down to Nop + 1 do
      /*backward transpose op in H down to item Nop*/
      Transpose_Backward (j) ;
    endfor ;
  endif ; }

```

## CONTINUOUS GLOBAL ORDER AND DEFERRED BROADCAST: THE SOCT4 ALGORITHM

### Principle

In SOCT4 as in SOCT3, the operations are ordered globally using a timestamp given by a sequencer. They are then delivered on each site in this order thanks to the sequential reception. The originality of SOCT4 comes from the fact that forward transpositions that take into account concurrent operations are now made by the generator sites of the operations. This results in three major advantages:

- a) the receiver site does not have to separate history any more ; thus backward transposition becomes unnecessary,
- b) the received operation can be stored as it is in the history without further transformation,
- c) state vectors are no longer needed.

To achieve this, the broadcast of an operation must be deferred. More precisely, an operation generated on a site S is as usual executed locally without delay to satisfy the real-time constraint, but it is not broadcast until it has been assigned a timestamp and all the operations which precede it according to the timestamp order (i.e. *precedes<sub>S</sub>*) have been received and executed on site S. Moreover, before being broadcast, the operation is forward transposed with all concurrent operations, that is to say with operations received by S after its generation and preceding it in the global order.

As in SOCT3, when a remote operation, let us say  $op_{i+1}$ , with timestamp (i+1) is delivered on site S by the sequential reception procedure, then all operations  $op_j$  ( $\forall j, 1 \leq j \leq i$ ) which precede it in the global order have already been received and executed on the site. Thus, in the absence of local operations, the remote operation can be executed as it is ; otherwise (Fig. 8), if there exists m local operations, noted  $op_{L_1}, op_{L_2}, \dots, op_{L_m}$ , which have been executed in this order and that are waiting to be broadcast, then the integration of  $op_{i+1}$  in the history  $H_S(n) = op_1.op_2...op_i.op_{L_1}.op_{L_2}...op_{L_m}$  (with  $i + m = n$ ) consists of:

1) determining the operation to be executed on the current state. The operation  $op_{i+1}$  is forward transposed to take into account the local execution of concurrent operations  $op_{L_1}, op_{L_2}, \dots, op_{L_m}$ ; the resulting operation is then executed on the local state.

2) reordering the history according to the timestamp order. Local operations waiting to be broadcast,  $op_{L_1}, op_{L_2}, \dots, op_{L_m}$ , are forward transposed, one after another, to take into account the execution of the concurrent operation  $op_{i+1}$ . This way, they will not have to be transposed upon reception. The operation  $op_{i+1}$  is stored without any modification at position (i+1) in the history.

Concerning point 1, given that  $seq = op_{L_1}.op_{L_2}...op_{L_m}$  is the sequence of local operations waiting to be broadcast, the forward transposition of  $op_{i+1}$  to be executed is  $op_{i+1}^{seq}$ . Concerning point 2, every waiting local operation  $op_{L_k}$  must be forward transposed with the forward transposed operation of  $op_{i+1}$ , noted  $op_{i+1}^{seq_k}$  where  $seq'_k = op_{L_1}.op_{L_2}...op_{L_{k-1}}$  is the sub-sequence of the (k-1) local

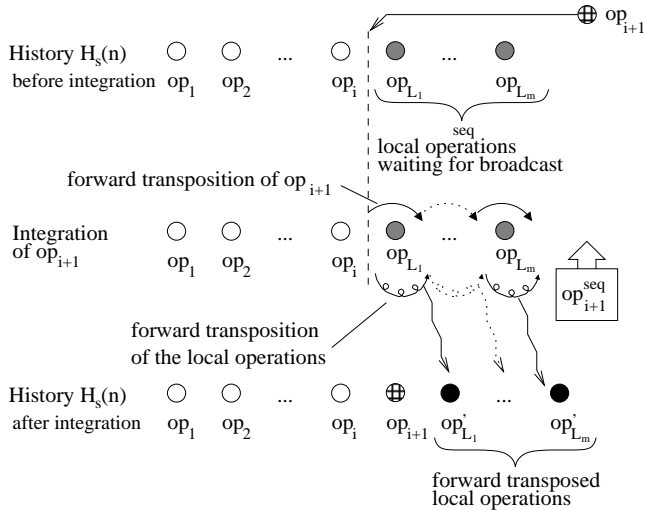


Figure 8: Integration of an operation in SOCT4

operations which precede  $op_{L_k}$ . Operations  $op_{i+1}^{seq}$  and  $op_{L_k}$  are thus defined on the same object state. The forward transposed operations of the  $m$  local operations are  $op_{L_1}^{op_{i+1}}$ ,  $op_{L_2}^{op_{i+1}}$ , ...,  $op_{L_m}^{op_{i+1}}$ .

So far, we have only considered  $op_{i+1}$  as a remote operation. When the operation being delivered is local to the site, it has already been executed and stands in the right place in the history. The transpositions required to take into account the concurrent operations received after its generation have been made before its broadcast. It can thus be ignored.

Unlike SOCT3, in SOCT4 the operations once delivered are no longer needed by the algorithm since they are not involved in the transpositions used.

### The SOCT4 Algorithm

The SOCT4 algorithm is constituted of the following procedures: *Local\_Execution*, *Deferred\_Broadcast*, *Integration* as well as the *Sequential\_Reception* procedure already defined for SOCT3. The same notations as before are used. As stated earlier,  $N_s$  is the timestamp of the last delivered operation on site  $S$  and it is incremented during the execution of *Sequential\_Reception*;  $n$  is the total number of operations (either local or remote) executed and stored in the history (initially  $N_s = n = 0$ ).

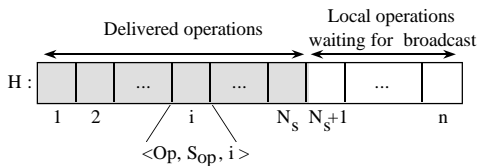


Figure 9: Representation of history  $H_s(n)$

The array  $H[]$  is used to store the operations delivered to site  $S$ , as well as local operations. Delivered operations are stored in the array according to the order of their timestamps. Without loss of generality, we will assume that the position occupied by an operation in  $H$  is identical to its timestamp. Thus an operation timestamped with  $i$  ( $i \leq N_s$ ) and delivered on site  $S$  will be stored in  $H[i]$ . Local

operations waiting to be broadcast, however, may be stored in places  $H[j]$  ( $N_s < j \leq n$ ) that do not correspond to their timestamp. Strictly speaking, the delivered operations might not need to be stored for the algorithm to work.

The *Local\_Execution* procedure is called whenever a local operation  $op$  is generated.  $op$  is first executed and then stored at the end of the history. When  $op$  receives its timestamp in return of the call to *Ticket*, it is checked to determine whether it can be broadcast.

```

procedure Local_Execution ( $op$ ) {
  execute( $op$ ) ;  $n := n + 1$  ;
   $H[n] := \langle op, S_{op}, -1 \rangle$  ;
   $H[n].timestamp := Ticket()$  ;
  Deferred_Broadcast() ; }

```

The *Deferred\_Broadcast* procedure broadcasts to all sites (including  $S$ ) the first local operation on condition that its timestamp is equal to  $N_s + 1$ , which means that all other operations with lower value of timestamp have been delivered.

```

procedure Deferred_Broadcast() {
   $\langle op, S_{op}, N_{op} \rangle := H[N_s + 1]$  ;
  if  $N_{op} = N_s + 1$  then broadcast ( $\langle op, S_{op}, N_{op} \rangle$ ) endif ; }

```

The *Integration* procedure is called when an operation is delivered by the *Sequential\_Reception* procedure. If the operation is local then it is ignored since it has already been executed and stored in the history. If it is remote, the local operations waiting to be broadcast are shifted one place to the right before the triplet  $\langle op, S_{op}, N_{op} \rangle$  is stored in the position  $N_s$  (i.e.  $N_{op}$ ). Then the operation is transposed with the local ones, and the resulting transposed operation is executed. Meanwhile, each local operation waiting for broadcast is in turn forward transposed to take into account this new concurrent operation. The procedure ends by checking if a local operation should be broadcast.

```

procedure Integration ( $\langle op, S_{op}, N_{op} \rangle$ ) {
  if  $S \neq S_{op}$  then
    for  $j := n$  down to  $N_s$  do  $H[j + 1] := H[j]$  endfor ;
     $H[N_s] := \langle op, S_{op}, N_{op} \rangle$  ;  $n := n + 1$  ;
    for  $j := N_s + 1$  up to  $n$  do
       $op_L := H[j].operation$  ;
       $H[j].operation := Transpose\_forward(op, op_L)$  ;
      /*forward transpose local operations*/
       $op := Transpose\_forward(op_L, op)$  ;
    endfor ;
    execute( $op$ ) ;
  endif ;
  Deferred_Broadcast() ; }

```

Due to lack of space, the proof of correctness of SOCT4 is not included in this paper.

### Discussion about the sequencer

In both algorithms, SOCT3 and SOCT4, a sequencer is used to globally serialize operations. The consequence in SOCT4 is that all broadcasts are issued sequentially; it results in a difficult collaboration between users when the propagation delay of an operation on the network is high (e.g. on the order of a minute). This characteristic makes SOCT4 particularly adapted to fast networks. Measure-



ments should enable this aspect to be quantified.

For purposes of clarity, algorithms given here are sequential. However, when exploiting potential concurrency between procedures, the fact that a local operation is waiting for a timestamp does not preclude neither the execution of another local operation nor the integration of a remote operation.

Concerning the lack of robustness of the sequencer, we underline that failure of the sequencer (or the loss of timestamp) does not preclude local functioning. Collaboration is suspended but each user may continue to work separately. Collaboration will be resumed as soon as the sequencer is recovered. The effect is the same when a timestamped operation is not broadcast by a malicious site. In other words, the collaboration provided by SOCT3 and SOCT4 cannot be partial: either all sites collaborate or each one works separately.

### COMPARISON WITH RELATED WORKS

Table 1 gives an overview of SOCT3 and SOCT4 algorithms, as well as the dOPT, adOPTed, GOT, GOTO and SOCT2 algorithms. Many similarities exist among these algorithms regarding the techniques employed and we will take a closer look at the differences that make the originality of each one.

User intention preservation is achieved in all algorithms by transforming an operation with respect to concurrent operations in order to permit its integration. This transformation is employed under various names (L-Transformation, Inclusion\_Transformation, Forward Transposition). Furthermore, some algorithms such as GOT, GOTO,

SOCT2 and SOCT3 implement an additional transformation, called Exclusion\_Transformation or Backward Transposition, which enables the order of execution of two consecutive operations to be changed without violating the user intention. In adOPTed the problem related to partial concurrency is solved by the construction and memorization of a multidimensional graph which enables all the potential serialization orders to be retrieved. Only SOCT4 uses Forward Transposition only, the partial concurrency problem being solved thanks to deferred broadcast.

To ensure the copies convergence, the definition of these transformations must in the general case satisfy two conditions (C1 and C2). Condition C1 guarantees that the operation resulting from the transposition of two concurrent operations will not depend on the order they are serialized in. All the algorithms assume that the transformations verify condition C1. Only GOT algorithm does not impose this condition but fixes a global order and restrains the transformations to be made in this order; this obliges to undo the operations arrived "in advance", that is to say before operations that precede them in the global order. Condition C2 aims at making the transformation of an operation with a sequence independent of the order of the operations in this sequence. Algorithm dOPT does not use this condition but is unable to ensure copies convergence. Similarly, it does not solve the partial concurrency problem, as these two problems were not yet identified when it was written. Complying with condition C2, when possible, remains hard to verify, and is thus worth replacing with a global serialization order as in GOT, SOCT3 and SOCT4.

		dOPT	adOPTed	GOT	GOTO	SOCT2	SOCT3	SOCT4
<b>Constraints</b>	<b>Intention preservation</b>	dOP Transformation	L-Transformation and multidimensional graph	Inclusion Transformation and Exclusion Transformation	Inclusion Transformation and Exclusion Transformation	Forward Transposition and Backward Transposition	Forward Transposition and Backward Transposition	Forward Transposition
	<b>Causality preservation</b>	State vectors	State vectors	State vectors	State vectors	State vectors	Timestamps	Timestamps
	<b>Copies convergence</b>	Condition C1 (but convergence is not achieved)	Condition C1 and Condition C2	Non continuous global order and Undo/Redo	Condition C1 and Condition C2	Condition C1 and Condition C2	Condition C1 and Continuous global order	Condition C1 and Continuous global order
<b>Remote operation</b>	<b>Broadcast</b>	Immediate	Immediate	Immediate	Immediate	Immediate	Immediate (as soon as timestamp is assigned)	Deferred, in timestamp order
	<b>Delivery</b>	Causal order	Causal order	Causal order	Causal order	Causal order	Continuous global order	Continuous global order
<b>History</b>	<b>Memorized order</b>	Execution order	Several equivalent orders respecting the causal order.	Global order (= execution order)	Optimized causal order	Optimized causal order	Continuous global order ( $\neq$ execution order)	Continuous global order ( $\neq$ execution order)
	<b>Memorized operation (at the time of its integration)</b>	Executed operation	Received operation and some transformed operations	Executed operation	Executed operation	Executed operation	Transformed operation conforming to the timestamp order	Received operation

Table 1. Comparative table of the techniques used by the algorithms

In GOT, the global serialization order is not continuous, which entails undoing and then redoing some operations for integrating a late operation at the right position in the history. In SOCT3 and SOCT4, a global continuous order is achieved by using a sequencer which associates a timestamp to each operation. The delivery and therefore the integration of remote operations can be made according to the order of the timestamps.

Causality preservation is achieved in all the algorithms but SOCT3 and SOCT4 by using state vectors which implement a reception procedure that ensures that the operations are delivered following an order compatible with the causal order. In SOCT3 and SOCT4, the use of a sequencer to obtain continuous timestamps not only respects the causality, but also gives a continuous global order free from condition C2.

Concerning the broadcasting of an operation, this is done immediately in all algorithms apart from SOCT4, where it is deferred until all preceding operations in the global order have been received. This simplifies the integration and gets rid of the backward transposition needed in SOCT2, SOCT3, GOT and GOTO.

All algorithms but SOCT4 rely on the memorization by each site of the operations it received or generated. SOCT4 only needs to know the operations that are waiting to be broadcast, that is to say those which have been generated locally but have not yet been delivered. The management of the history on each site is thus simplified.

## CONCLUSION

This article reviews problems raised by the convergence of copies in a distributed real-time collaborative environment that exploits the semantic properties of operations. In this context, to ensure the convergence of the copies while respecting the user intention, we have proposed two new algorithms, called SOCT3 and SOCT4. By implementing a continuous global serialization order these algorithms remove a particularly restrictive condition required by the transformation used in other existing algorithms, and simplify the process of integration of an operation in the history associated with a copy of an object on each site, without the need to undo and then redo any operation. The deferred broadcast of operations to other sites goes a step further in this simplification, by making backward transposition as well as the state vectors unnecessary. A comparison with the existing algorithms concludes this article, and gives a synthetic overview of advantages and drawbacks of the different techniques implemented in each one. Our current challenge consists in providing the user with the possibility to undo an operation already executed without requiring condition C2.

## REFERENCES

1. Allison C.: *Concurrency Control for Real Time Groupware*, CE94: Concurrent Engineering Research and Applications. A global Perspective, Pittsburg, August 1994, pp. 163–170.
2. Banino J.S., Kaiser C., Zimmermann H.: *Synchronization for distributed systems using a single broadcast channel*, Proc. 1<sup>st</sup> Int. Conf. on Distributed Computing Systems, Huntsville, October 1979.
3. Ellis C.A., Gibbs S.J.: *Concurrency Control in Groupware Systems*; Proc. ACM Int. Conf. on Management of Data (SIGMOD'89), Seattle, May 1989, pp. 399–407.
4. Ellis C.A., Gibbs S.J., Rein G.L.: *Groupware : Some issues and experiences*; Commun. ACM, January 1991, vol.34, n° 1, pp. 39–59.
5. Karsenty A., Beaudouin-Lafon M.: *An Algorithm for Distributed Groupware Applications*; Proc. 13<sup>th</sup> Int. Conf. on Distributed Computing Systems (ICDCS'93), Pittsburgh, May 1993, pp. 195–202.
6. Lamport L.: *Time, Clocks, and the Ordering of Events in Distributed System*; Commun. ACM, July 1978, vol. 21, n° 7, pp. 558–565.
7. Le Lann G.: *Algorithms for distributed data sharing systems which use tickets*; Proc. 3<sup>rd</sup> Workshop on Distributed Data Management and Computer Networks, Berkeley, August 1978.
8. Mattern F.: *Virtual time and global states of Distributed Systems*; Proc. Int. Workshop on Parallel and Distributed Algorithms, Elsevier Pub., 1989, pp. 215–276.
9. Nichols D., Curtis P., Dixon M., J. Lamping: *High-latency, low-bandwidth windowing in the Jupiter collaboration system*; Proc. ACM Symposium on User Interface Software and Technologies, November 1995, pp. 111–120.
10. Palmer C.R., Cormack G.V.: *Operation Transforms for a Distributed Shared Spreadsheet*; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'98), Seattle, November 1998, pp. 69–78.
11. Prakash A., Knister M.J.: *Undoing Actions in Collaborative Work*; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'92), November 1992, pp. 273–280.
12. Reed D.P., Kanodia R.K.: *Synchronisation with eventcounts and sequencers*; Commun. ACM, February 1979, vol. 22, n° 2, pp. 115–123.
13. Ressel M., Nitsche-Ruhland D., Gunzenhäuser R.: *An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors*; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'96), Boston, November 1996, pp. 288–297.
14. Suleiman M., Cart M., Ferrié J.: *Serialization of Concurrent Operations in Distributed Collaborative Environment* ; Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97), Phoenix, November 1997, pp. 435–445.
15. Suleiman M., Cart M., Ferrié J.: *Concurrent Operations in a Distributed and Mobile Collaborative Environment* ; Proc. 14<sup>th</sup> IEEE Int. Conf. on Data Engineering (IEEE / ICDE'98), Orlando, February 1998, pp. 36–45.
16. Suleiman M.: *Sérialisation des opérations concurrentes dans les systèmes collaboratifs répartis* ; PhD thesis, University of Montpellier 2, July 1998.
17. Sun C., Jia X., Yang Y., Zhang Y.: *A generic operation transformation schema for consistency maintenance in real-time cooperative editing systems*; Proc. ACM Int. Conf. on Supporting Group Work (GROUP'97), Phoenix, November 1997, pp. 425–434.
18. Sun C., Ellis C.S.: *Operational Transformation in Real-Time Group Editors : Issues, Algorithms and Achievements* ; Proc. ACM Int. Conf. on Computer Supported Cooperative Work (CSCW'98), Seattle, November 1998, pp. 59–68.
19. Weihl W.E.: *Commutativity-Based Concurrency Control for Abstract Data Type*; IEEE Transactions On Computers, vol. 37, n° 12, December 1988, pp. 1488–1505.