The Time Warp Mechanism for Database Concurrency Control

D. Jefferson
Computer Science Department
University of California - Los Angeles

A. Motro
Computer Science Department
University of Southern California

Abstract

In this paper we introduce the Time Warp mechanism as a new method for concurrency control in distributed database systems. Its major distinguishing features are first, a unification of transactions and data as two forms of the more general notion of *object*, and second, the use of object rollback as the fundamental tool for synchronization instead of blocking or abortion.

1. Introduction

In this paper we introduce the Time Warp mechanism as a new method for database concurrency control, differing from other kinds of mechanisms [11, 5, 1, 2], in at least five respects. (1) It adheres to an object-oriented approach to database design in which there are no formal distinctions among "transaction" objects, "data" objects, and "system" objects. (2) It uses neither locking nor abortion-and-retry to resolve access conflicts. Instead, it uses a more powerful synchronization tool: general rollback. (3) It is free of deadlock and starvation. (4) It is free of restrictions on transaction form such as sequentiality, two-phase structure, or predeclaration of data items to be accessed. And, (5) it introduces the notion of the *virtual time*, an artificial time scale from which transaction timestamps are drawn. The virtual time of a transaction is never changed after it is assigned, and transactions are *guaranteed* be committed strictly in virtual time order. One disadvantage of the Time Warp mechanism is that the principle of commitment in strict virtual time order may require delaying a transaction with a late time stamp to wait for one with an earlier time stamp, even if the two transactions do not conflict. It is thus vulnerable to response time degradation due to long transactions.

A concurrency control algorithm is generally considered correct if it merely executes transactions atomically; that is the point of the serializability criterion for correctness [6]. The order of transaction commitment is considered to be nondeterministic, and is determined in practice by timing considerations such as computation and communication delay and the outcomes of races (e.g. for timestamp assignment or for control of locks). In the Time Warp mechanism the order of commitment is defined when a transaction is assigned a time stamp, which in this paper is the moment of initiation. Although transaction are committed in strict virtual time order, the body of transaction is executed concurrently with other transactions, and there may be concurrency within a transaction as well.

The Time Warp mechanism was originally developed at the Rand Corporation as a synchronization method for distributed discrete event simulation [7]. In this paper we have adapted it for use in databases.

2. Data and Transaction Model

We view a database system as a collection of *objects*, sequential processes that execute concurrently and communicate through timestamped messages. Each object is capable of both computation and communication, and has an associated a "virtual clock" and several queues to be described later. All data items, all executing transactions, and most of the database management system processes are packaged as objects, and the concurrency control mechanism does not make any formal distinctions among them. The defining feature of a *data object* is that it has a particularly simple program that responds only to *read* and *write* messages, whereas other objects may have much more complex behavior. Similarly, the only thing that distinguishes *transaction objects* is that they happen to receive messages directly from external users. For simplicity we assume that there is no object creation or destruction, although allowing these operations would cause no fundamental problems.

All actions in the database are synchronized with respect to a temporal coordinate system called *virtual time* and each transaction occurs "at" a unique virtual time. Every observable computational *event* in the database (state change, message send or receive, etc.) can be thought of as occurring "at" some virtual time t and some virtual place x (i.e. object x). An *action* is defined to be the set of all events occurring at a single virtual time and place, and thus (t,x) are the *coordinates* of an action in the space-time continuum of database activity. Actions at one virtual time may only causally affect actions at the same or later virtual times. A *transaction* is defined to be the set of all computational actions that take place during a single instant of virtual time, i.e. it is a slice of computation at one virtual time across all objects.

A virtual time value is similar to a "pseudotime" defined for the multiversion concurrency control mechanism of Reed [10, 3]. Although any total ordering would suffice we will assume, as in [10], that virtual times have two components: (1) the high-order part is the real time at which the entire transaction was initiated; (2) the low-order part is a unique identification of the user (or terminal) that initiated it. They are treated together as a single unstructured value; the division into two parts is simply to guarantee that each transaction has a unique timestamp.

A user initiates a transaction by sending a message to a transaction object, as shown in Figure 2-1. The message is stamped with a virtual time, which then becomes the time stamp of the entire transaction. Upon receipt of a message a transaction object initiates a distributed computation that may involve many other objects and messages. All messages and computation that derive directly or indirectly from the initiating message are considered part of the same transaction and are given the same timestamp. The fact that all communication and computation associated with a transaction takes place during one "instant" of virtual time is what makes it atomic.

CH2261-6/86/0000/0474$01.00 © 1986 IEEE

An object progresses through virtual time by processing the messages that are directed to it in time stamp order. Each object has a *virtual clock* variable that records its "current" virtual time.

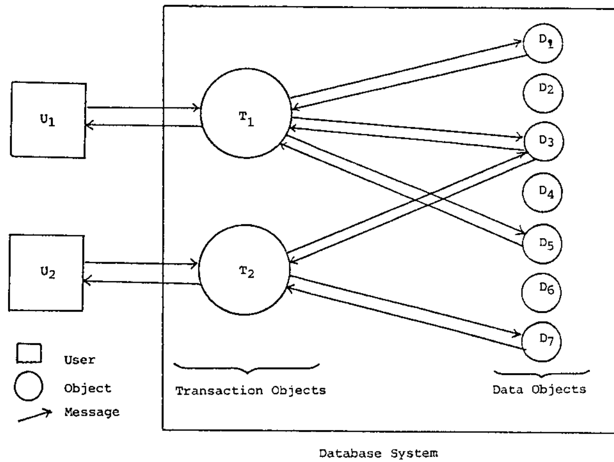

Figure 2-1: Transaction and Data Objects

Whenever an object begins processing a message, its virtual clock is set to the value in the message's timestamp. Hence, the virtual clock at each object always reads the virtual time of the transaction it is executing. When an object is not executing any current transaction, (i.e. when it has no more messages to process) its virtual clock reads *+infinity*. The virtual clocks of different objects are not tightly synchronized; ordinarily some virtual clocks will be ahead while others are behind.

Every object is required to process its input messages in increasing timestamp order. However, there is no guarantee that messages will arrive in timestamp order, and there is no way to know if a particular message is the "next" one to be processed. Therefore an object must be prepared to roll back whenever a message arrives out of order. The major innovation of the Time Warp mechanism (Section 3) is its clean, efficient implementation of rollback in a distributed environment.

We assume that an object communicates with other objects using the primitives *request* and *answer*. There are two corresponding *types* of messages, 'Q' (request) and 'A' (answer). The call

request(dest, text)

sends a type-'Q' message, where *dest* is the name of the destination object, and *text* is a sequence of values carrying the content of the message. The *request* primitive blocks the calling object until the corresponding answer message arrives and then returns the value in the text field of the answer message as the value of *request*. The timestamp of the request message is the value in the sender's virtual clock. Reading a data object is done through a call such as

var := request(DataObject, 'READ') .

Similarly, a write request can be done by the call

var := request(DataObject, ('WRITE', value)) .

The call

answer(uid, text)

sends a type-'A' message in reply to the request message with unique identifier *uid*. Its implied destination is the sender of the request, and its timestamp is, once again, the value in the sender's virtual clock (which is always the same as the timestamp on the request). Messages carry unique identifiers so that it is clear which answer goes with which request.

We assume that there are several functions available to parse messages, defined as follows:

first(msg),second(msg) return the appropriate elements of the message text

uid(msg) returns the unique identifier of a request message

The following procedure is a template for the body of a data object. The vast majority of all objects in the database will be of this type. It is called each time an input message is to be processed, with the message itself acting as a parameter. The variable *datavalue* is declared to be *own* (in the Algol-60 sense), meaning that its value is preserved across calls to this procedure. It acts as the *state* of the process, and it holds the "data" that is stored in this data object.

```
X: object(msg:message);
  own datavalue;
  begin
    case first(msg) of
      "WRITE":    begin
                    datavalue := second(msg);
                    answer(iud(msg), "" )
                  end;
      "READ":     answer(uid(msg), datavalue)
    endcase
  end
```

The behavior of a data object can thus be summarized as follows: (1) for a *read* request, it sends an answer message containing the data; (2) for a *write* request, it stores the value from the request in its state variable, and sends an empty answer message; and (3) for any other incoming message it does nothing. For simplicity we omitted error checking.

There is no corresponding template to describe the behavior of all transaction objects. The following is an example of a transaction object named *INCR* that receives user requests of the form (*d*, *v*) and then increments the data object named *d* by the value *v*. We assume the state of object *d* is a single numeric value, and again we omit error checking.

```
INCR: object(msg:message);
  local v: sequence;
    dummy: string;
  begin
    v := request(first(msg), "READ");
    dummy := request(first(msg),
              ("WRITE", first(v) + second(msg)))
  end;
```

Object *INCR* has no own variables because it has no state that needs to be saved from one transaction to the next.

Another example is the transaction *DOUBLE*, which receives requests of the form (*d*) and doubles the value in object *d*.

```
DOUBLE: object(msg:message);
  local v: sequence;
    dummy: string;
  begin
    v := request(first(msg), "READ");
    dummy := request(first(msg),
              ("WRITE", 2 * first(v)))
  end;
```

These two transaction objects will be used later in the example

illustrating concurrency~~-control.~~—

Because the *request* primitive involves blocking to await an answer message the possibility of deadlock arises. For example, if objects A and B each send request messages to the other with the same timestamp, each will wait for a response and deadlock may result. However, it is important to realize that this kind of deadlock can only arise within a single transaction. Deadlock within a transaction, like infinite loop within a transaction, is the responsibility of the transaction programmer, and not part of the database concurrency control problem. On the other hand, preventing deadlock *between* transactions is an essential part of concurrency control. Under Time Warp such deadlocks are impossible because messages from different transactions bear different timestamps and are synchronized by rollback, with no blocking at all. At no time does any transaction block waiting for any action from another transaction. Without blocking, there cannot be deadlock.

3. The Time Warp Mechanism

In order that distributed transactions appear to execute in timestamp order it is sufficient that messages be processed in timestamp order at each object. During execution of an object (Figure 3-2) messages arrive asynchronously. We distinguish between the moment a message *arrives*, i.e. when it is enqueued, and the moment it is *received*, i.e. when it is read and processed. Since there is no guarantee that messages will arrive in timestamp order, most concurrency control mechanisms based on timestamps resolve at least some conflicts by aborting transactions and re-executing them later with new timestamps. The Time Warp mechanism never aborts a transaction; instead it resolves conflicts using object rollback. Transaction objects, data objects, and all other objects are equally subject to rollback.

The Time Warp mechanism is not *conservative* in the sense of [1], i.e. it does not adopt the policy of blocking objects until they can proceed safely. As long as an object has unprocessed messages in its input queue it continues to execute. It might be considered an *optimistic* mechanism in the sense of Kung and Robinson [9] (although theirs is not a timestamp order mechanism) because it processes incoming messages without worrying about whether there will be a conflict. But unlike their mechanism, when conflicts do occur Time Warp does not resolve them by aborting and retrying whole transactions. Instead, if an errant message arrives with a timestamp earlier than the virtual time in the receiver's virtual clock, the receiver rolls back to a time earlier than the timestamp on the message. The unit of rollback is not the *transaction*. Individual *actions* are rolled back, at the receiving object and possibly elsewhere, but only those that have a causal connection with the errant message. The set of actions undone need not comprise a transaction. Acceptable performance of the Time Warp mechanism rests on the assumption that in practice rollback will be inexpensive and infrequent, and that total time lost in abandoned lookahead paths and rollback is no greater than that lost by other concurrency control mechanisms in blocking or aborting transactions.

3.1. Objects and Messages

To explain Time Warp concurrency control we must first describe the structure of objects and messages. Figure 3-1 shows the format of a message. It contains the following fields:

- the *sender*, identifying the sending object;
- the *receiver*, identifying the destination object;

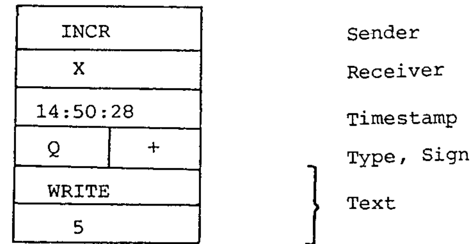

Figure 3-1: The Format of a Message

- the *timestamp*, indicating the virtual time of the transaction that the message is part of;
- the *type*, either 'Q' or 'A', indicating whether the message is a request or an answer message;
- the *uid*, a unique identifier for request and answer messages to allow their correct association;
- the *sign*, either + or -, indicating whether the message is a positive message or a negative message;
- the *text* of the message. We will mostly confine our attention to messages whose text is the specification of a read or write operation to be performed, or the value of a data item transferred, although in general the text may be arbitrary.

Messages (both *request* and *answer*) come in *positive* and *negative* forms. All messages sent explicitly are positive. Negative messages arise only as part of the rollback mechanism, and are completely transparent to the object programmer. The interpretation of a negative message -M is "undo all effects of message M". Two messages that are negatives of one another are said to be *antimessages*.

The structure of an object is shown in Figure 3-2. It has the following components:

- A *name*, which uniquely identifies it;
- A *virtual clock*, a register containing the object's *local virtual time (LVT)* that measures the object's progress through virtual time;
- A *state*, which, depending on the object, may be a single item of data, or an entire process stack;
- An *input queue*, which contains messages from other objects, maintained in timestamp order, and FIFO order within timestamp;
- An *output queue*, which contains messages sent by this object, maintained in timestamp order, and FIFO order within timestamp;
- A *state queue* of past states of this object (including LVT), maintained in LVT order.

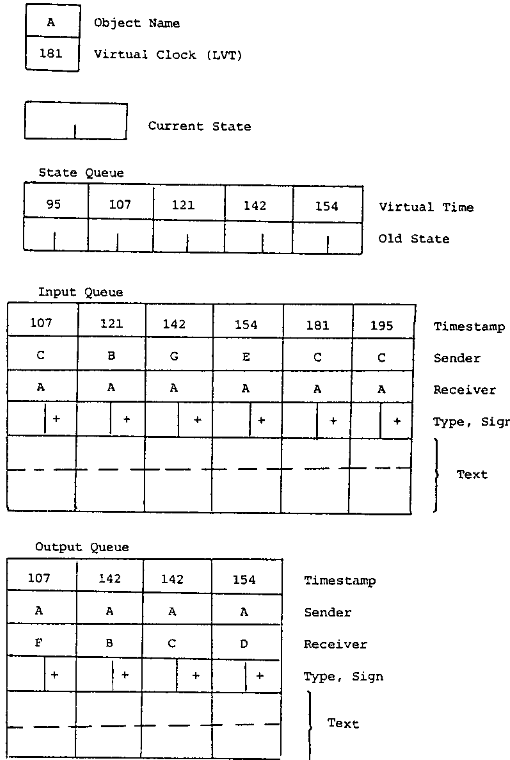

Figure 3-2: The Structure of an Object

The three queues contain information about the past needed to support rollback. There is a special queueing discipline characteristic of the Time Warp mechanism that makes distributed rollback possible. When a message arrives at an object it is inserted into the input queue in timestamp order (and in FIFO order within timestamp). However, if its antimessage is already present in the queue, then both messages are immediately removed and destroyed, and the two antimessages are said to *annihilate* one another. The queueing discipline is partially described by the following algebraic axioms:

$-(-M) = M$

$enqueue(enqueue(Q,M),-M) = Q$

The enqueueing of a message can either lengthen the queue by one message, or shorten it by one. Annihilation occurs regardless of whether the positive or negative message was enqueued first.

3.2. Rollback Mechanism

Object execution is a sequence of *actions*, each of which occurs in one "instant" of virtual time, and is thus atomic with respect to actions at different virtual times. Each action is one object's share of a transaction, and has three parts:

1. The object's LVT is updated to be the timestamp of the "next" input message, i.e. the message in the input queue with minimal virtual receive time strictly greater than the current LVT. If there is no such message, LVT is set to infinity.
2. The object receives the first message (request) in the input queue whose virtual receive time is equal to its LVT, unless its LVT is infinity, in which case the object waits until another message arrives.
3. The object performs the computation called for by the message. This may involve updating the object's state, sending and receiving other messages with the same timestamp, and waiting for answer messages from other objects.

The arrival of a message is like an interrupt. If a positive message has a timestamp greater than or equal to the receiver's current LVT, then it is merely enqueued in its proper place (possibly causing annihilation) and execution continues. However, if it has a timestamp less than the LVT of the receiving object, then it "should" have been received earlier, and the receiving object may have have taken an incorrect computational path. The receiver is therefore immediately rolled back to the state it was in just prior to the virtual time given in the timestamp of the message and it begins to execute forward again. The rollback happens "immediately", i.e. even if the object is in the middle of processing a message. Not only does the mechanism work correctly in this case, but it is actually necessary to do it this way. It would be incorrect to wait until completion of the current action before initiating rollback because the object might conceivably be in an infinite loop *caused* by the fact that the message prompting the rollback was never processed. Only interrupting the process and rolling back immediately can get around that problem.

As an example, suppose in Figure 3-2, a message M with timestamp 135 arrives. Figure 3-3 shows A restored to the state it was in after processing the message with timestamp 121, the last action before virtual time 135. All states from the state queue with virtual times greater than 135 are discarded because they represent a "projected future" that is possibly incorrect. Both positive and negative messages can cause rollback, and rollback occurs even if the message arrival also causes annihilation, as long as the timestamp on the message is less than the LVT of the receiver. When a positive message causes a rollback it is generally because it arrived late compared to those with similar timestamps; when a negative message causes rollback it is generally because the corresponding positive message should never have been sent.

To undo all the incorrect computation done since virtual time 135 it is not sufficient to roll back only object A. As the Figure 3-2 shows, A sent out three messages with timestamps greater 135, some of which might not have been sent if message M had arrived in time to be processed in order. Those messages in the output queue sent after time 135 must be *cancelled*. To cancel a message is to "unsend" it, i.e. move the entire database to a state that it could have achieved if the message had never been sent. This requires undoing all side-effects that derive, directly or indirectly, from this message. It is not at all obvious how to do this because the message may be physically in transit; it may have arrived and be in a queue; or it may have already been processed and caused a further round of side-effects and messages, some of which may be in transit, etc. Furthermore, there is the possibility of message cycles.

At the heart of the Time Warp mechanism is the following observation: *to exactly cancel the effects of a message M it suffices merely to send its antimessage, -M.* To illustrate this, let M be the message with timestamp 142 sent by object A to object B in Figure 3-3, and consider the effect of subsequently sending $-M$. Sending a

negative message, just like sending a positive one, involves (1) enqueueing a copy in the sender's output queue and then (2) transmitting a copy to the receiver's input queue. Enqueueing *-M* in *A*'s output queue causes an immediate annihilation with the copy of *M* already there, leaving *A* with no record that *M* or *-M* ever existed. When *-M* arrives at *B* there are four cases to consider because the timestamp of *-M* (142) may or may not be less than the LVT of *B*, and *-M* may arrive either before or after *M* does. All four cases are handled (recursively) according to the same rule described earlier: *-M* is enqueued in the *B*'s input queue (causing annihilation if *M* is already there) and, in addition, if it has a timestamp less than or equal *B*'s LVT, it causes *B* to roll back and send antimessages.

The simplest case is when the timestamp of *-M*, 142, is greater than *B*'s LVT, and *M* has already arrived. The two messages annihilate (with no rollback) and the situation is just as though neither had ever been sent.

If 142 is less than or equal to the LVT of *B* and *M* arrived before *-M*, then *M* has already been processed by *B*. *-M* is then enqueued, annihilating with *M*; *B* is rolled back; and antimessages are sent to cancel any computation incorrectly invoked in other objects. All record of *M* and *-M* is destroyed, and this time as *B* executes forward it will not receive either message.

In the other two cases when it happens that *-M* arrives before *M*, *-M* is enqueued as usual and is later annihilated when *M* eventually arrives. It may happen that *M* is so late in arriving that *-M* must actually be processed by *B*. Since any action taken to process *M* is certain to be rolled back when *M* arrives, it simply does not matter

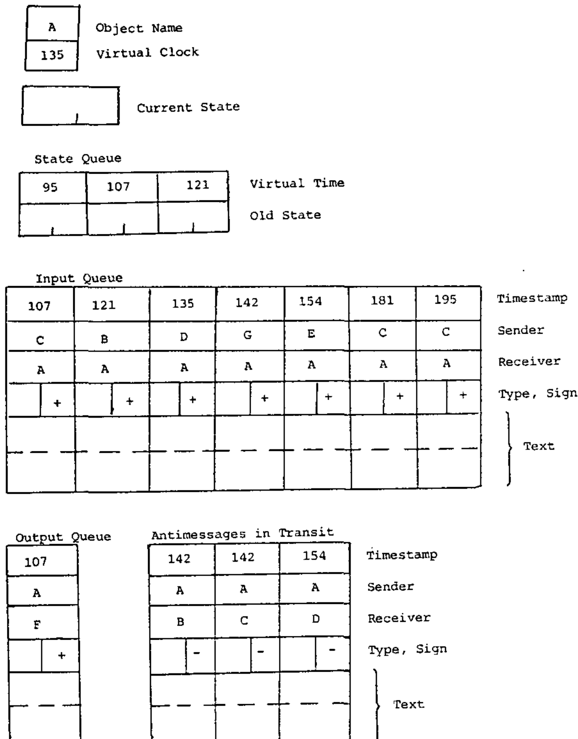

Figure 3-3: Object Rollback

what convention we make in this circumstance. We therefore assume that negative messages are processed as no-ops. As an optimization we could further legislate that the arrival of *M* after *-M* has been received causes annihilation but inhibits the rollback because, since *-M* was treated as a no-op, rollback is unnecessary.

One of the virtues of this rollback mechanism using antimessages is that it mixes smoothly with ordinary transaction execution. It is not necessary to stop or block any database activity to accomplish a distributed rollback, and rollback need not be treated as an atomic action. It is simply an activity that goes on concurrently with forward execution. At equilibrium in a large database one would expect that some constant fraction of all messages would be negative, mixed in among the positive messages.

3.3. Global Virtual Time

So far we have described rollback naively, without facing some of the difficult problems it introduces. In this section we consider some of the issues that prevent people from considering general rollback as a serious tool, particularly in a distributed environment. The first problem, of course, is the problem of output: while it is clearly possible to roll back actions internal to a database, actions that cause observable side-effects on the external world (printing, delivering cash) are irreversible. A second problem is that of storage management: we have been describing the mechanism so far as though the entire history of messages and object states remains in storage, but this is usually impractical. There must be some way to purge messages and states that are no longer needed.

What both of these problems have in common is the need for a virtual time that is an absolute floor on all future rollbacks for all objects in the database. For any instant *r* in real time we define the *global virtual time at r (GVT[r])* to be the minimum of (a) the minimum timestamp of all *unprocessed* messages in the database (including any currently being processed or in transit), and (b) the minimum possible timestamp from any real time clock that is used for generating new timestamps. (If the real time clocks were perfectly synchronized, then this latter value would have *r* in the high order bits and zeros in the low order bits.)

At any moment *r* we are guaranteed that no object will ever again roll back to a virtual time less than GVT[*r*] because (a) no message can cause a rollback to a time earlier than its timestamp; (b) no message ever induces the creation of another message with an earlier timestamp; and (c) no new message from outside the database (beginning a new transaction) can have an earlier timestamp than GVT[*r*]. GVT is nondecreasing with respect to real time and, assuming every transaction eventually terminates normally, GVT is guaranteed to increase. GVT is thus a measure of global *progress* in the database. This definition of GVT in terms of the set of unprocessed messages and real time clocks is mathematically sound, but because of unavoidable delays in communication the exact current value of GVT cannot be known. For the moment let us assume that there is a oracle that can always provide the exact value of GVT instantaneously.

Consider a transaction *T* with timestamp *t*. Whenever GVT exceeds *t* we can be certain that all actions taken by *T* are permanent, and will never be undone. We therefore say that *T* is *committed* when GVT>t. Notice that no specific system action is necessary to accomplish commitment; commitment is not performed, but rather noticed after the fact. Transactions whose timestamps are greater than or equal to GVT are considered to be uncommitted even if they have completed all of their actions, because the Time Warp

mechanism cannot yet be certain that they will not be subject to rollback.

The concept of GVT and transaction commitment provides the solution to the three problems discussed above. To prevent output operations from happenning before it is known that they will not be rolled back, we make it a rule that all output from a transaction with timestamp t is buffered, and not physically performed until the transaction is committed, i.e. until GVT$>t$. While they are buffered, before the commitment of the transactions that sent them, output request messages may of course be annihilated by antimessages sent during the normal process of rollback.

Similarly, once GVT$>t$ we are assured that no saved messages or states with timestamps earlier than t will have any further effect on the future of the database, allowing us to purge the system of all messages and states older than GVT. This is equivalent to saying that whenever a transaction is committed, all storage associated with it can be released.

Although it is not possible to calculate the current value of GVT exactly, it is possible is to estimate it. An algorithm for calculating a slightly out-of-date value of GVT that we will g is described in [8]. Using this algorithm, if we begin calculating g at real time r_1 and the algorithm ends at time r_2 then the value of g returned satisfies GVT$[r_1] \leq g \leq$ GVT$[r_2]$. The run time of the calculation, $r_2 - r_1$, is approximately twice the time it takes to broadcast a message to every processor on the network. Although g is only an estimate for GVT[r], it is guaranteed never to be too high, and thus it is safe to use it in place of GVT[r] for controlling output and storage management.

The Time Warp mechanism must from time to time calculate g and use it for output and storage management. This is done concurrently with normal database activity and must be considered part of concurrency control overhead. The frequency of estimating GVT is a performance parameter that determines tradeoffs between database response time, throughput, and space requirements. If GVT is estimated very often then a high fraction of the network bandwidth and processor cycles may be devoted to overhead, reducing the amount available for executing transactions. On the other hand, if it is estimated infrequently, then transaction commitment may be artificially delayed, resulting in higher response time and less frequent release of storage.

4. Performance Considerations and Lazy Rollback

Probably the primary worry about the Time Warp mechanism is that the number of rollbacks will be prohibitive. The extreme fear is that somehow, in a complex situation, the database will enter an infinite rollback oscillation so that it cannot make progress. We have already addressed this concern when we pointed out that GVT, which measures the progress of database activity, is guaranteed to increase provided that every transaction considered alone terminates normally. But although progress is guaranteed, one might worry that the system, through a cascade of rollbacks, could unwind a large number of actions and then re-execute them, in effect taking 10 steps backward and then 11 steps forward. This is possible in principle when short transactions conflict with very long ones, or when a message gets delayed for a very long time in transmission, but these circumstances cause similar performance problems with other concurrency control mechanisms.

It is important to note that although secondary rollbacks can propagate from an initial one, the propagation is usually quite limited. Whenever a message with timestamp t causes a rollback, all the antimessages produced, directly or indirectly, have timestamps greater than or equal to t, so that no object can be induced to roll back to a virtual time earlier than t. The worst possible case is that every object in the database might have to roll back to time t, which is the Time Warp equivalent of having the entire database locked by one transaction.

The incidence of rollback can be sharply reduced by one optimization that is so important that it should be included in any implementation: *lazy rollback*. With lazy rollback, when an object rolls back from time 100 to time 50, it does not *immediately* send any antimessages. Instead, those messages it had sent between time 50 and 100 simply remain in the output queue. During the subsequent re-execution from time 50 to 100 the new output messages are compared to those still in the output queue. Whenever a new message, e.g. with timestamp 57, is exactly equal to one saved from the last time the object was at time 57, the transmission of the new message to its destination is inhibited because the receiver already has a copy. A new message not found in the output queue is transmitted as usual. If a message in the output queue, e.g. with timestamp 62, has not been resent by the time the object re-executes past time 62 again, then it is deleted and its antimessage is transmitted to the receiver, cancelling the effects of the original. In effect, lazy rollback allows an object to transmit to its receivers the second time around a set of messages and antimessages that represents the *difference* between the first path from time 50 to 100 and the second path. It prevents the wasteful possibility that an antimessage is sent to cancel a computation, and then the original message is re-sent, causing the exact same computation to be repeated.

Determining which transactions affect which others to avoid unnecessary synchronization is called *conflict analysis* [4]. One of the main virtues of lazy rollback is that it prevents unnecessary secondary rollbacks by performing what can be characterized as dynamic conflict analysis. After a rollback it is quite likely that some of the messages sent before the rollback will be regenerated since later transactions do not always depend on the results of earlier ones. This is especially true when the earlier one is a read-only transaction, which cause no side-effects. Thus, if a *read* message happens to cause a lazy rollback of a data object that rollback will not induce any antimessages or any secondary rollbacks. It will be a purely local action, very similar to accessing an earlier "version" of the data in a multiversion concurrency control mechanism [10]. Under lazy rollback the action of sending antimessages is thus decoupled from rollback, and treated as an overhead activity taking place during normal forward execution.

5. An Example of Synchronization

Let us consider a version of the Lost Update problem to illustrate how the Time Warp mechanism synchronizes concurrent transactions. Assume there are two transaction objects, *INCR* and *DOUBLE*, and one data object X as described before, and assume that X's value (state) is a single integer 5. Let us further assume that user *U1* initiates a transaction to increment X by 7, and that the timestamp assigned to the transaction is 37. A moment later user *U2* initiates a transaction to double X, and the timestamp assigned is 39. These transactions are initiated by sending the appropriate messages to objects *INCR* and *DOUBLE*. Each transaction will send a *read* message, wait for the answer, do some arithmetic, then send a *write* message and again wait for the answer.

Under the serializability criterion the two transactions must be

executed atomically, but either order is acceptable, allowing the final value of X to be either 17 or 24. However, under the Time Warp criterion the *INCR* transaction must precede the *DOUBLE* transaction because of its lower timestamp, and thus 24 is the only acceptable final value.

Let us simulate the behavior of *INCR, DOUBLE* and X as the two transactions execute concurrently. Each transaction involves both a *read* and a *write* operation, and each of those requires two messages, and for each message we can distinguish the sending moment from the arrival moment from the receiving moment. Thus, an enormous number of sequences of these events are possible and each may result in a different pattern of antimessages and rollback. We illustrate here one possible concurrent execution. To show the variety of circumstances in which rollback and annihilation can occur we have chosen an event sequence of some complexity.

1. Initially both *DOUBLE* and *INCR* have messages from user *U1* and *U2* respectively. Data object X has no messages in its input queue, and its current value is 5.

2. Although the request to *DOUBLE* has the higher timestamp (42), we assume here that it is the first to receive the request message. It advances its virtual time to 42, and sends a *read* request to X. The request is placed both in *DOUBLE*'s output queue and in X's input queue.

3. X updates its virtual time to 42, receives the *read* request, and generates the answer message. The answer is inserted into both X's output queue and *DOUBLE*'s input queue.

4. At this point *INCR* updates its virtual time to 37, receives its request from user *U1*, and sends a *read* request to X. As usual the request is inserted into *INCR*'s output queue, but when it arrives at X's input queue it causes X to roll back to time 37. As part of the rollback mechanism X sends an antimessage to *DOUBLE*, to cancel the answer message sent in the previous step. The antimessage is inserted into X's output queue and a copy is sent to *DOUBLE*'s input queue.

5. This step is logically part of the previous one, but it seems clearer to separate in in this description. First the negative answer message in X's output queue annihilates with its positive counterpart. Then, if *DOUBLE* has already received the answer to its read request (and was calculating a response) the arrival of the anti-answer message causes it to roll back to the state it was in before receiving the answer. In any case, the negative and positive answers in *DOUBLE*'s input queue annihilate, leaving it in a state in which it has sent the *read* request and is again waiting for an answer.

6. X now receives *INCR*'s *read* request and sends an answer back.

7. Next, X re-receives *DOUBLE*'s read request, advances its virtual time to 42, and sends an answer back. This is the same answer (5) that *DOUBLE* got before, and it is still incorrect.

8. Having completed its arithmetic, *INCR* sends a *write* message to X to update its value from 5 to 12. Because the *write* request has a timestamp of 37 and X is at time 42, X must again roll back and send an antimessage for the answer it just sent to *DOUBLE*.

9. As in step 5, the antimessage pairs in X's output queue and *DOUBLE*'s input queue annihilate, and *DOUBLE* is again left waiting for the answer to its original *read* request.

10. X receives the *write* request from *INCR*, updates its value from 5 to 12, and sends an answer back to *INCR*.

11. X receives (for the third time) *DOUBLE*'s *read* request, updates its time to 42, and sends a new answer message, this time containing the correct value, 12.

12. *DOUBLE* receives the new answer message, doubles the value in it and sends a *write* request to X with the new value (24).

13. Finally, X receives the *write* message from *DOUBLE* and updates its value to 24, as desired.

In this example there were four rollbacks; X rolled back twice and so did *DOUBLE*. This may seem excessive at first glance, but it is important to realize that we deliberately chose a somewhat artificial example. Not only did we assume in Step 1 that the *read* request from *DOUBLE* arrived at X before the one from *INCR* despite the fact that the *INCR* transaction was initiated first, but we even assumed X had time to send the reply to *DOUBLE* before *INCR*'s request arrived. Later we assumed that X had time to resend his *read* request a second time before *INCR* sent his *write* request. In effect we assumed that the *DOUBLE* transaction is *faster*, even though it would probably not be in practice.

A more likely scenario assumes the two transactions execute at the same speed. Then *INCR*'s *read* request would arrive at X, followed by *DOUBLE*'s *read* request, followed by *INCR*'s *write* request. The latter would cause X and *DOUBLE* to roll back once each, for a total of only two rollbacks, after which execution would proceed smoothly.

References

[1] P. A. Bernstein and N. Goodman.
Concurrency Control in Distributed Database Systems.
Computing Surveys 13(2):185-221, June, 1981.

[2] P. A. Bernstein and N. Goodman.
A Sophisticate's Introduction to Distributed Database Concurrency Control.
In *Proceedings of the Eight International Conference on Very Large Data Bases*, pages 62-76. Mexico City, Mexico, 1982.

[3] P. A. Bernstein and N. Goodman.
Multiversion Concurrency Control - Theory and Algorithms.
ACM Transactions on Database Systems 8(4):465-483, December, 1983.

[4] P. A. Bernstein, D. W. Shipman and J. B. Rothnie.
Concurrency Control in a System for Distributed Databases (SDD-1).
ACM Transactions on Database Systems 5(1):18-51, March, 1980.

[5] C. J. Date.
An Introduction to Database Systems, Volume II.
Addison Wesley, 1983.

[6] K. P. Eswaran et al.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19:624-633, December, 1976.

[7] D. R. Jeffesron and H. A. Sowizral.
Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control.
Technical Report N-1906-AF, Rand Corporation, Santa Monica, California, December, 1982.

[8] D. R. Jeffesron and H. A. Sowizral.
Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control.
Technical Report, Rand Corporation, Santa Monica, California, 1983.

[9] H. T. Kung and J. T. Robinson.
On Optimistic Methods for Concurrency Control.
ACM Transactions on Database Systems 6(2):213-226, June, 81.

[10] D. P. Reed.
Implementing Atomic Actions on Decentralized Data.
ACM Transactions on Computer Systems 1(1):3-23, February, 1983.

[11] J. D. Ullman.
Principles of Database Systems.
Computer Science Press, 1982.