# Hybrid Time Warp (HYT):
# A Protocol for Parallel Database Transaction Management

John A. Miller

Aideen M. Dennis

Department of Computer Science

415 Graduate Studies

University of Georgia

Athens, Georgia 30602-7404

Phone: (706) 542-3440

E-Mail: jam@cs.uga.edu

# Hybrid Time Warp (HYT):
# A Protocol for Parallel Database Transaction Management

**Abstract**

The Time Warp protocol has been used very successfully for parallel and distributed simulation. This paper considers several variants of Time Warp adapted for database transaction management. More importantly, it develops promising variants of a hybrid of Time Warp (TW) with Multiversion Timestamp Ordering (MVTO), called Hybrid Time Warp (HYT). The variants of both TW and HYT are produced by combining these concurrency control protocols with various recovery and commit protocols as well as introducing some optimizations. Some of the optimizations considered for improving the performance of TW and HYT for database transaction management include: differentiation of transaction objects from data objects, intra-transaction parallelism, lazy cancellation/reevaluation and jump forward, reduced state saving overhead, elimination of cascaded rollbacks, and most importantly early commits.

**Keyword**: Transaction Management, Concurrency Control, Recovery, Parallel and Distributed Databases, Parallel and Distributed Simulation.

## 1. Introduction

The combination of newer more demanding database applications, along with the increasing use of multiprocessor machines (both loosely and tightly coupled) produce a need and an opportunity for improved database performance [DeWi92]. As levels of concurrency get higher and higher, the impact on performance of transaction management protocols becomes more and more important. The use of common concurrency control protocols such as Two-Phase Locking may lead to an unacceptable performance bottleneck [Kim90]. Possible solutions include: (1) finding a useful weaker correctness condition than serializability, (2) finding higher performance

This research was partially supported by a University of Georgia Faculty Research Grant.

protocols to enforce serializability, and (3) exploiting special semantics of operations. Approaches (1) and (3) are relatively complicated for a naive user to work with, so a higher performance protocol for enforcing serializability is an attractive alternative.

One way to achieve better response times and throughput in a parallel/distributed environment is to avoid protocols that block processors. This claim is based on the intuition that it is better to use available processors, even if the work using those processors turns out to be incorrect and must be thrown away, than to block processing when processors are available. This argument assumes that the useful work done by transactions in the forward direction outweighs the non-productive work that has to be reversed. Simulation results presented in [Mill86, Mill92, Mill94] indicate that this assumption is justified. A second way to improve performance [xxxxyy, Liu92] is to have multiple versions of objects (rather than updating an object, new versions of the object are created). This reduces the kinds of conflicts that can occur (e.g., write-write conflicts are no longer a problem).

The departure point for this paper is the work done by David Jefferson. He adapted his *Time Warp* protocol [Jeff82, Jeff85] used for Parallel Discrete-Event Simulation (PDES) to serve as a database concurrency control protocol [Jeff85, Jeff86] for Parallel Transaction Processing Systems (PTPS). Interestingly enough, there are close similarities between protocols for database concurrency control and protocols for synchronization of simulation events. Because of the large body of successful work in PDES, it is naturally a fertile ground for migration of ideas to related fields such as operating systems or databases [Jeff85, Fuji90a, Fuji90b]. Time Warp, in particular, is one of the more successful optimistic protocols used for PDES.

A similar protocol, which has been found to have good performance [Mill86, Mill92, Mill94] enforcing serializability in parallel systems, is the *Multiversion Timestamp Ordering* protocol [Reed78, Reed83]. Protocols such as Multiversion Timestamp Ordering and Time Warp have the following advantages: (1) high effective concurrency levels, (2) deadlock free operation (because there is no or limited blocking), and (3) efficient operation unless there is a conflict. These types of protocols resolve conflicts by partially rolling back (undoing some

operations) or completely aborting transactions. The fact that transactions are not blocked leads to higher effective concurrency levels. Thus, if aborts or partial rollbacks do not occur with great frequency, these protocols should exhibit excellent performance.

We extend Jefferson's work on Time Warp, by developing several other variants of Jefferson's Time Warp protocol [Mill91a, Mill92, Mill94]. These variants include some minor changes/optimizations. More importantly, we develop a hybrid protocol that combines the best features from the Time Warp (TW) and MultiVersion Timestamp Ordering (MVTO) protocols. We call this protocol the HYbrid Time warp (HYT) protocol.

In this paper, the protocols are presented in detail: Their strategies, algorithms and data structures are given. The correctness of two of the protocols is also shown. (Performance profiles of these protocols are presented in [Mill91a, Liu92, Mill92, Mill94, Soma94b].) Specifically, Section 2 defines appropriate correctness conditions and discusses the execution model; Sections 3, 4 and 5 discuss the MVTO, TW and HYT protocols, respectively; and finally, Section 6 gives conclusions.

## 2. Database Correctness Conditions

In this section, we define the basic correctness conditions for multiversion databases operating within a parallel or distributed environment. In the Appendix, we also define the primary correctness condition used in parallel and distributed simulation, and compare it to the database correctness conditions.

Abstractly, the execution model used for this paper as well as the corresponding simulation studies is simply that of active and/or passive objects that communicate by message passing. Parallelism is achieved through the use of multiple physical processors (CPU's and IO Processors). In practice, this abstract model could be mapped to a shared-memory (tightly-coupled) or distributed memory (loosely-coupled) multiprocessor or a distributed system with high performance communications (e.g., an ATM network). Studies within the PDES field have shown the protocols such as Time Warp are sensitive to message overhead [xxxxyy].

We define an active object to be an object (instance of a class) containing its own execution thread. It can respond to incoming messages as well as execute its own script. Threads compete for physical resources such as CPU's and IO processors. The greater the number of physical resources, the less important resource contention becomes, and the more important data contention becomes. A passive object simply provides methods (or conventional equivalents such as read and write operations) that can be executed by active objects.

Without loss of generality, we define a database *DB* as a finite set of objects (active or passive).

$$DB = \{ x, \ldots, z \}$$

Although our definitions fit object-oriented database the most cleanly, we use the word object in a generic sense to represent a tuple in a Relational Database, a record instance in a Network or Hierarchical Database, or an encapsulated object in an Object-Oriented Database. (In the section of the paper on Time Warp, we will need to narrow the scope of our definition of object.) Each object consists of an ordered list of versions.

$$x = (x_i, \ldots, x_j)$$

At any given time, there will also be a finite set of active objects (or processes) that access the database *DB*, performing *operations* that retrieve/update information in *DB*. (This set may be a subset of *DB* or a distinctly different set, see Section 4.) The access to *DB* can be achieved in the traditional way by directly accessing the state of the objects in *DB*, or by following the object-oriented paradigm, i.e., by sending messages between objects. From an abstract point of view, there is little difference between the two. In this paper, the operations may be read, write or commit, or one of the two exception operations, abort or partial rollback. (Some systems provide additional operations, and those which support semantic concurrency control [Garc83], allow operations such as increment to be interleaved, and thus achieve enhanced performance [Allc83, Garc83].)

For the sake of consistency, a sequence of interrelated operations are grouped together to

form a transaction. As a unit of work, a transaction should fully complete or not execute at all. A transaction should also not interfere in destructive ways with other transactions executing at the same time. A transaction can be formally define as follows [Bern87]:

*Definition 1. Transaction:* A transaction $T_i$ is a partial order $<_{T_i}$ of operations (read, write and commit).

A *write* operation on object $x$ by transaction $T_i$, $w_i(x)$, produces a new version, $x_i$, which is inserted into $x$'s ordered list of versions. A *read* operation of object $x$ by transaction $T_i$, $r_i(x)$, retrieves the value of some version, $x_j$, that has already been written (future versions may not be read). We also need to guarantee that the effects of successfully completed work are permanent. A *commit* operation by transaction $T_i$, $c_i$, signals its successful completion. After $T_i$ has committed, its versions must remain in the database until they are no longer needed (see Section x). This will require the use of sufficiently non-volatile memory (e.g., disks, solid-state disks, flash memory, or battery-backed CMOS RAM).

### 2.1. Serializability

The principal correctness condition governing the concurrent (or parallel) execution of transactions is serializability. Serializability maintains the consistency of data, by guaranteeing that if the individual transactions are correct then their combined effect will be correct [Papa77, Ullm82]. Serializability is maintained by constraining the ordering of operations so that their effect is equivalent to a serial execution of the transactions. Since we are dealing with multiversion databases in this paper, we will use the specific criterion of One-Copy Serializability (or 1SR). Our definitions are a composite of those found in [Papa86, Bern87, Cell88]. We begin by defining the notion of a schedule (or ordering of operations). In parallel and distributed database systems, operations carried out by different processors may overlap in time. This combined with difficulty of synchronizing clocks in distributed systems, means that for some operations it may be impossible to determine which came first. Consequently, schedules in such systems are

defined to be partial (rather than total) orders.

*Definition 2. Schedule:* A schedule $S$ over a set of transactions $T = \{T_1, T_2, \ldots, T_n\}$ is a partial order $<$ of all of the operations within the transactions. The ordering of operations within transactions must be preserved in $S$.

*Definition 3. Version Function:* A version function $h$ determines the version read (written) by each read (write) operation. If $r_k(x)$ reads version $x_j$, then $h(r_k(x)) = x_j$. Assuming no overwrites, then $h(w_k(x)) = x_k$, i.e., it is an identity map.

*Definition 4. Serial Schedule:* A schedule $S$ is serial if for any two transactions $T_i$ and $T_j$, either all of the operations of $T_i$ precede those of $T_j$ or vice versa.

*Definition 5. Standard Serial Schedule:* A serial schedule $S$ is standard if $h$ maps to the most recently written version. $h$ is then said to be a standard version function.

*Definition 6. Committed Projection:* The committed projection of $h$ is the subset of $h$ where all of the reads (domain elements) belong to committed transactions.

*Definition 7. One-Copy Serializable (1SR) Schedule:* A schedule $S$ is 1SR if the committed projection of its version function $h$ is equal to the the committed projection of the (standard) version function $h'$ in some standard serial schedule $S'$, and both $S$ and $S'$ have the same transactions.

### 2.2. Recoverability

Recovery issues [Gray81, Hadz83, Agra85, Bern87] are also vitally important to the correct operation of databases. In particular, recoverability [Hadz83] guarantees that after a failure (either transaction failure or system crash) the database can be recovered to a consistent state, at which point things can carry on as usual. Recoverability is maintained by not allowing a transaction to commit until all the data it has read is committed, After a failure, we recover the database to a state including the effects of exactly those transactions that committed before the failure. Following the definition given by Hadzilacos [Hadz83], we formally define

recoverability as follows:

*Definition 8. Recoverability:* A schedule $S$ is recoverable if, for every transaction $T$ that commits, $T$'s commit follows the commit of every transaction from which $T$ has read.

Recovery from catastrophic failures (e.g., a media failure) are not considered in this paper.

## 3. Multiversion Timestamp Ordering (MVTO) Protocol

Under the Multiversion Timestamp Ordering protocol, each transaction is timestamped upon initiation (see [Bern87] for techniques to generate good unique timestamps). Each version of an object is also timestamped with the timestamp of the transaction writing it. Additionally, it is important to keep track of the latest reader of each version. Hence, the following information needs to be maintained for the successful operation of the protocol:

$ts(T_i)$ = timestamp of transaction $T_i$
$ws(x_k)$ = write-stamp of version $x_k$
$rs(x_k)$ = read-stamp of version $x_k$

where $T_i$ is the $i^{th}$ transaction to initiate ($ts(T_i) < ts(T_j)$ for $i < j$). The write $w_i(x)$ by transaction $T_i$ produces a new version $x_i$ with write-stamp $ws(x_i) = ts(T_i)$. This version is inserted into the ordered list $x$ according to its write-stamp. Transactions will read the most recent version of an object. Specifically, the most recent from its time frame; versions from the future may not be read. Consequently, when transaction $T_i$ reads from object $x$, the version with the largest write-stamp not exceeding the timestamp of $T_i$ is returned.

The Multiversion Timestamp Ordering protocol supports high concurrency since reads can be executed in a relatively unrestricted way. Except for the concern for recoverability, there are no restrictions at all (simply the appropriate version is chosen and read). Writes, however, may cause a transaction to be aborted. In particular, if transaction $T_i$ attempts to write a version of object $x$ that interferes with an existing information transfer, then this write cannot be allowed to proceed. This occurs under the following conditions,

$$ws(x_k) < ts(T_i) < rs(x_k)$$

where $x_k$ is the version that immediately precedes the version $T_i$ is attempting to write. In other words, $T_i$ is attempting to insert a new version immediately after $x_k$, but since the read-stamp on $x_k$ is greater than the timestamp of $T_i$ this cannot be allowed. The problem is that a transaction reading version $x_k$ should have really read the version that transaction $T_i$ is attempting to write. For example, consider the following schedule:

$$SCH1: \quad w_2(x)\, r_4(x)\, w_5(x)\, r_7(x)\, r_9(x)$$

If transaction $T_8$ attempts to write a version of object $x$, it must be aborted since transaction $T_9$ has already read the version written by transaction $T_5$.

Since writes may be aborted, some type of recovery protocol is necessary to maintain consistency. One that appears particularly well suited is the Realistic Recovery protocol (see Appendix). It will block each read until the version it is attempting to read is committed. This will introduce some extra delay into the system, but the amount of blocking will be rather small in comparison to the Pessimistic Recovery protocol [Grif85] or the commonly used Two-Phase Locking concurrency control protocol [Mill86]. (Another possibility is to use the Optimistic Recovery protocol [Grif85, Mill94]. This approach was used in the study by [Liu92] with the result being that the overall performance is marginally worse than using the Realistic Recovery protocol.) For a proof that the Multiversion Timestamp Ordering protocol produces 1SR schedules, see [Bern87].

## 4. Time Warp (TW) Protocol

The Time Warp protocol has been studied extensively for parallel and distributed simulation. This same Time Warp protocol can be adapted for use as a database concurrency control protocol [Jeff85, Jeff86].

The Time Warp protocol fits in nicely with the object-oriented paradigm. Therefore, we will now narrow our definition of the database $DB$. The objects in $DB$ will now be encapsulated

objects. In general, there are two main categories of objects of interest, Transaction Objects and Data Objects. Combining the two results in a third, Dual Objects.

1. Transaction Object -- An active object with an embedded transaction. All activity in the database is initiated by such objects. After the transaction has committed the containing object may be terminated or it may start up a follow-on transaction (i.e., such objects can be used to provide a session-like capability [Garz88]). Transaction objects send messages (e.g., read and write) to data objects, and receive replies from the data objects (e.g., ACK's, NAK's, or data). Being active they have their thread of control, and in some implementations they could be multi-threaded.

2. Data Object -- An object which is capable of handling incoming request messages (e.g., reads and writes). It is possible to implement data objects either as passive or active objects. The advantage of making them active is that it increases the uniformity of the system, all objects are active. These active objects will sleep until the next operation request message comes in. Because of the potentially large number of data objects in some systems, it may be preferable to implement them as passive objects. There are numerous ways in which threads belonging to active objects could execute a passive object's method, e.g., the thread could belong to a single-threaded or multi-threaded transaction object, or to an agent for a group of data objects.

3. Dual Object -- Some applications may be simplified if an object can contain data that other objects wish to access, and have its own behavior in which it accesses other objects (see [Mill91b] for an example). In this case the object will combine the capabilities of both data and transaction objects.

The Time Warp protocol allows objects to advance their state forward as rapidly as possible, until a message (from some other object) is received whose receive-time is in the past. Such a message is referred to as a straggler. (Note, in PDES events may be scheduled to occur in the future so that the receive-time may be greater than the send-time; the two times are identical for databases.) When a straggler message *arrives*, the receiving object must be rolled back to this

time so that the straggler message can be *processed*. Determination of past, present and future are dealt with on a local basis. Each object keeps tracks of its own Local Virtual Time or *LVT*. The *LVT*'s of different objects need not agree. Indeed, the motivation behind Time Warp is to allow objects to individually advance their local clocks (*LVT*'s) as rapidly as possible. Since the *LVT*'s can be reversed by rollbacks, it is also important to know the time beyond which rollbacks cannot go. This is important for commitment of transactions and purging of old versions. This time is referred to as Global Virtual Time or *GVT*.

Let us motivate the applications of Time Warp, by looking back at our example for Multiversion Timestamp Ordering. In schedule $SCH1$, transaction $T_7$ also reads the version written by transaction $T_5$. Unlike the read by transaction $T_9$, this presents no problem for transaction $T_8$ in its write attempt. To maintain serializability, it suffices to have transaction $T_9$ change its read from transaction $T_5$'s version to transaction $T_8$'s version. Hence, we need to be able to partially rollback a transaction. The simplest approach is to back up transaction $T_9$ to the point of the read of object $x$ (i.e., the *read-in-question*) and then begin redoing the transaction. However, if a dependency analysis of transactions is performed, then it is possible to significantly optimize the rollback operation.

### 4.1. Intra-Transaction Dependencies and Rollbacks

Within transactions, many different kinds of dependency patterns are possible. In parallel and distributed databases, operations of a single transaction can be executed in parallel, unless there is a dependency between two operations. Even if operations are not executed in parallel, dependency patterns can still be exploited. Assuming that within a transaction there are no loops (or that loops have been unfolded), but that if statements are allowed, a transaction can be represented as a Directed Acyclic Graph (or DAG). The nodes in the DAG are operations while the arcs represent dependencies. An example of such a DAG is shown in Figure 1, while the corresponding pseudo-code for the transaction is shown in Figure 2. (A compiler for the transaction may also produce the DAG in addition to the object code.)

Figures 1 and 2:

When a transaction is rolled back, it suffices to simply rollback the read-in-question and any operations dependent upon it. More specifically, the following suffices:

1. "Redo" the read-in-question. In addition, handle erroneously included/excluded reads: (a) "undo" any reads that were previously performed, but should not have been as their guard condition now evaluates to false; and (b) "do for the first time" any reads that were previously skipped because a condition that uses the value of the read-in-question evaluated to false. We say such reads are conditionally dependent on the read-in-question (e.g., referring to Figure 1, $r(z)$ is conditionally dependent on $r(u)$).
2. "Undo" and then "redo" writes that are either conditionally or value dependent on the read-in-question. A write is value dependent, if the value obtained by the read-in-question is used in the write's calculation (e.g., referring again to Figure 1, $w(y)$ is value dependent on $r(y)$).

Note that in Time Warp, since transactions are only partially rolled back, their internal structure (including if statements) becomes important. Choosing to partially roll back rather than abort as is done with the Multiversion Timestamp Ordering protocol, has appeal since aborting is a more drastic operation. However, when the writes of a transaction are "undone", they may cause other transactions to be partially rolled back leading to *cascaded rollbacks*. This would suggest that as the rate of conflict gets high, the performance of this protocol would degrade rapidly.

In our prior modeling studies, we have examined the performance of the protocols under two distinct scenarios: (1) High Dependency Scenario: This scenario assumes for the sake of simplicity (both in modeling and in actual protocol implementations) that all of the writes are value dependent, and that no conditionally dependent reads need to be performed. (2) Low Dependency Scenario: More optimistic scenarios are certainly possible. For example, one or a few writes could be dependent on the read-in-question, while the other operations are independent.

### 4.2. Time Warp Data Structures

Implementations of Time Warp for simulators will maintain three important queues: an input queue, a state queue and an output queue. Timestamped event messages *arrive* at an object and are placed in the input queue in timestamp order. When an event message is *processed*, it causes an event to occur at the object which may change its state and/or generate other event messages to be sent out. (Note, messages may arrive in any order, but are processed in timestamp order.) The event messages to be sent out are copied into the output queue, so that if need be they can be readily cancelled with antimessages. The state queue simply saves the internal state of the process to facilitate rollback. The state may be saved before every event or less frequently to reduce space requirements [Fuji90a, Fuji90b].

Implementations for databases can maintain the same three queues. Messages arriving at an object are placed in the object's Input Queue (*IQ*) in timestamp order, and messages sent out by an object are saved in its Output Queue (*OQ*) to facilitate rollback operations. The specifics of processing of a transaction operation depend on the type of database involved.

1. Persistent versus Volatile State. The state of each object may have a persistent component and a volatile component. Since we are dealing with multiversion databases, each change to the persistent state must be saved in sufficiently non-volatile memory before commit. As far as the volatile state is concerned, when or whether to save it involves a tradeoff. The purpose of saving the volatile state is to be able to roll back a computation to specific points. In a simulation, these computations can be very complex, so that saving this information is important. (Unfortunately, this potentially large amount of state savings is considered to be one of the weak points of optimistic simulation mechanisms such as Time Warp [Fuji90b]. However, some recent results using adaptive memory management have potential for diminishing this problem [Das94].) In databases, the computations within transactions are typically less complex with just a few local variables. Consequently, the size of the state is likely to be small.

a. For each Transaction Object, volatile state information can be saved in its Volatile State Queue (*VSQ*), and can be saved at each possible rollback point, i.e., before each read message is sent out. Transaction objects typically will have no persistent state.

b. For each Data Objects new versions will be placed in the Persistent State Queue (*PSQ*). The versions are ordered according to their *LVT* values. Since versions will be tagged with *LVT*, (or a corresponding vector of timestamps), no volatile state information will need to be saved.

2. Propagation of Transaction Operations. Depending on how complex objects are organized, an update operation sent to (say) a car object may need to be propagated to engine and body sub-objects.

3. Databases with Triggers. If a database has triggers, then when a transaction operation is processed (e.g., an update operation) it may cause a transaction to be spawned to perform additional updates at other objects. In this way transactions can trigger other transactions much like events trigger other events in simulation.

For simplicity, in this paper we assume that transaction operations do not propagate and that triggers are not used.

A few more remarks on reducing state savings overhead are in order. There are a couple of techniques to reduce the amount of state information that needs to be saved. One is to keep track of which local variables have been modified since the last save point, and only save those. The other is to code (or translate) transactions in a single assignment language [Ungexx, Koch93]. Local variables can be allocated on a stack for a transaction, and this space can be released at commit time. Thus, for transaction objects we could just save the program counter, stack pointer and actively used registers. Resetting the program counter will back up the execution, while resetting the stack pointer will throw away unnecessary state information.

As you might expect the transaction object queues are likely to be relatively short, so simple data structures should suffice. On the other hand, the queues for data objects may be long enough for the advantages of efficient priority queue implementations (e.g., using heaps or splay

trees) to come into play. This could be the case if certain data objects are frequently accessed (hot spots). Simulation results presented in [Soma94a, Soma94b] show that when access patterns are uniformly distributed the queue lengths are quite short.

### 4.3. Jefferson's Original TW Protocol

The adaptation of the Time Warp algorithm to the problem of database concurrency control was originally done by David Jefferson [Jeff85, Jeff86]. However, Jefferson requires that all messages be processed in timestamp order. Thus, the relative order in which reads are processed is significant. This requirement is above and beyond what is necessary to enforce serializability, so one would expect a tradeoff of possibly higher consistency for lesser performance. However, optimizations can be used to enhance performance (see Subsection 4.5).

### 4.4. Anti-Transaction Variant of TW

The thorniest problem in using the Time Warp protocol is that some of the transactions that need to be partially rolled back cannot, since they have already committed. A possible remedy for this assumes that transactions can be effectively reversed through the use of anti-transactions. An anti-transaction is a system spawned transaction that reverses or counterbalances the effects of some transaction. In a more conventional database setting, anti-transactions are referreed to as compensating transactions [xxxxyy]. Therefore, under this variant, conflicts are handled by rollbacks for uncommitted transactions, and by spawned anti-transactions for committed transactions.

### 4.5. Deferred Commitment Variant of TW

In many applications, the use of anti-transactions is infeasible. For example, if money is dispensed from an automated teller machine, reclamation of this money is not possible. The Deferred Commitment variant of the Time Warp protocol avoids the necessity of anti-transactions by delaying the commitment of transactions.

This variant is similar to Jefferson's original protocol [Jeff86], with a few simple optimizations thrown in (e.g., no read-read conflicts, and access to prior versions is not considered to be an exception). Consequently, we will discuss it in more detail. Transaction objects will send messages (either read $M_r$ or write $M_w$) to data objects. These messages are timestamped with the timestamp of the issuing transaction. *Local Virtual Time, LVT*, is set to the timestamp of the message currently being processed. The *LVT* for data object $x$, $LVT(x)$, will be equal to the maximum of all of the read-stamps and write-stamps processed for object $x$. Messages with larger timestamps are enqueued in timestamp order (FCFS for messages with the same timestamp) and must wait their turn. In addition to ordinary positive messages, $M$, anti-messages, $-M$, are sometimes required to reverse the effects of positive messages. Table 1 summarizes the possible messages that can be sent between transaction and data objects. In this table Sign indicates whether it is a positive or negative message; Source Object and Dest Object refer to the sender and receiver, respectively.

| Message | Type | Sign | Source Object | Dest Object |
|---|---|---|---|---|
| $M_r$ | read request | + | transaction | data |
| $M_w$ | write request | + | transaction | data |
| $M_{val}$ | read reply | + | data | transaction |
| $M_{ack}$ | write reply | + | data | transaction |
| $-M_r$ | read request | - | transaction | data |
| $-M_w$ | write request | - | transaction | data |
| $-M_{val}$ | read reply | - | data | transaction |
| $-M_{ack}$ | write reply | - | data | transaction |

**Table 1: Possible Messages.**

Of these possibilities, our protocols will utilize all but the last one, $-M_{ack}$.

When a message first arrives at an object, it should be handled like an interrupt. If it is a positive message with a timestamp greater than or equal to the receiver's *LVT*, then it is enqueued in its proper place and execution of the current interrupted operation resumes. However, if it has a timestamp less than the receiver's *LVT*, the operation currently being processed may be erroneous. If the operations may be of a general nature, then Jefferson argues that the rollback must start immediately and that the current operation should not be permitted to finish, as this could lead to an infinite loop [Jeff86]. If, however, the current operation is a read or write, then it could be allowed to complete without disastrous consequences. This may be advantageous, since the current operation (possibly a lengthy DMA disk transfer) and the operation sparking the rollback may be independent.

Assuming (1) reliable message delivery, and (2) that messages sent from one object to another arrive in the order in which they were sent (e.g., virtual circuits or shared memory), the basic operations (read, write, commit and rollback) work as described in the following subsections.

### 4.5.1. Read

Consider a situation in which transaction object $z$ desires to read data object $x$. The issuing transaction $T_i$ embedded within $z$ sends a read message, $M_r$, to object $x$, as depicted in Figure 3. When this read message first arrives at object $x$, current processing will be interrupted so that a decisions can be made on what to do with the message. Two cases must be dealt with.

`Case 1:` $ts(T_i) < LVT(x)$

`Action 1:` Enqueue the read message, $M_r$, on $IQ(x)$ in timestamp order;
The processing of the read will begin immediately.

`Case 2:` $ts(T_i) \geq LVT(x)$

`Action 2:` Enqueue the read message, $M_r$, on $IQ(x)$ in timestamp order;
The read will be processed when the local clock, $LVT(x)$,
advances to this message (or this message becomes next
among the current events).

Note that Case 1 does not generate a rollback since a read can not initiate a conflict in a multiversion database operating under the 1SR criterion. The $M_r$ message is enqueued in either case to facilitate annihilation with $-M_r$. It is important to keep information about the reads around so that in the event of a rollback the reads can be "undone".

When the time has come to process the read message sent by $T_i$, the following steps need to be taken by data object $x$:

1. Get the appropriate version of $x$ from $PSQ(x)$. Specifically, get the version $x_k$ with the largest timestamp not exceeding $ts(T_i)$, i.e., $ws(x_k) \leq ts(T_i)$.
2. If $ts(T_i) > rs(x_k)$, then $rs(x_k) = ts(T_i)$.
3. Send an $M_{val}$ reply message containing $x_k$ to transaction object $z$ and record this fact in $OQ(x)$.

### 4.5.2. Write

Consider a situation in which transaction object $z$ desires to write data object $x$. The issuing transaction $T_i$ embedded within $z$ sends a write message, $M_w$, to object $x$, as depicted in Figure 4. When this write message first arrives at object $x$, current processing will be interrupted so that a decisions can be made on what to do with the message. Again, two cases must be dealt with.

`Case 1:` $ts(T_i) < LVT(x)$

`Action 1:` Enqueue the write message, $M_w$, on $IQ(x)$ in timestamp order;
The processing of the write will begin immediately.

`Case 2:` $ts(T_i) \geq LVT(x)$

`Action 2:` Enqueue the write message, $M_w$, in timestamp order;
The write will be processed when the local clock, $LVT$,
advances to this message.

When the time has come to process the write message sent by $T_i$, the following steps need to be taken by data object $x$:

1. Get the appropriate version of $x$ from $PSQ(x)$, i.e., the version $x_k$ with the largest timestamp $\leq ts(T_i)$.
2. If $ts(T_i) < rs(x_k)$, then send $-M_{val}$ messages to all readers-in-question.
3. Insert the new version into $PSQ(x)$ immediately after $x_k$.
4. Send an acknowledgement message, $M_{ack}$, to object $z$ and record this fact in $OQ(x)$.

Step 2 is very important in that it is the only means by which rollbacks can be initiated. Note that under Case 2, step 2 should be skipped, since the timestamp can only be smaller than the read-stamp of version $x_k$ if the message is from the past (i.e., a straggler).

Figures 3 & 4

### 4.5.3. Rollback

Rollbacks can only be initiated when a data object, e.g., $x$, receives a write request, $M_w$ from a transaction object, e.g., $z$. If when processing this write request, $x$ determines that it conflicts with existing reads (see step 2 for writes), then data object $x$ will send out anti-messages, $-M_{val}$'s, to all transaction objects $y$ having read the prior version of $x$ and having a timestamp larger than $z$'s. These transactions objects $y$, also known as the readers-in-question, are determined by consulting $OQ(x)$.

The arrival of such an anti-message, $-M_{val}$, at a transaction object $y$ indicates that the transaction has read a version of $x$ prematurely, since a version with a later timestamp which should have been read by these transactions, has been produced. If transaction object $y$ has not yet processed the positive equivalent, $M_{val}$, then these two messages simply annihilate each other; otherwise the transaction embedded within $y$ must be rolled back.

The complexity of the rollback of the transaction embedded within $y$ will largely depend on whether or not write operations are involved in the rollback. We consider the two cases below:

1. To rollback the transaction embedded within $y$, the potentially erroneous read-in-question must be handled. This involves "undoing" the read-in-question (e.g., a read-stamp may need to be reset). However, the read-in-question will have already been "undone" by data object $x$ (when it initially detected the conflict.) The read-in-question will be "redone" by transaction $y$ sending an $M_r$ message to data object $x$. According to the dependency pattern of the transaction, additional corrective action may (or will likely) be necessary. Any reads that are conditionally dependent on the read-in-question must be either "undone" or "done for the first time". Reads that were falsely performed because of an erroneous value for the read-in-question should be cancelled ("undone"), by sending an anti-message, $-M_r$, to the relevant data objects $u$. If these reads are not "undone", then future writes may erroneously produce conflicts, because of an incorrectly large read-stamp. Finally, those reads that were erroneously skipped, must be performed for the first time by sending out $M_r$

messages to the relevant data objects $v$. The "redos" will correct the internal state of the transaction so that its future calculations and writes will be correct. It is especially important to make rollbacks involving only reads as efficient as possible, since typically the majority of transaction operations are reads and some transactions are read-only, i.e., queries.

2. If the transaction embedded within object $y$ has produced a version $v_k$ based upon the old incorrect value obtained by the read-in-question, then this write must be "undone". As the write was to data object $v$, the "undo" can be accomplished by sending a $-M_w$ message to object $v$. When this anti-message arrives, $v$ will check to see if the corresponding positive message $M_w$ is still enqueued waiting to be processed. If so, the anti-message simply annihilates the positive message. If the positive message has already been processed, then $v$ will need to remove the offending version, and anti-messages, $-M_{val}$'s, will need to be sent to all transactions reading this version. This is how cascaded rollbacks occur. Finally, after the anti-messages have been sent, reevaluation may begin to calculate new values to be sent out in positive write messages, $M_w$. (Notice that these new positive messages will not annihilate with corresponding anti-messages, because of the restrictions we are able to impose as a result of our assumptions on the order of message delivery.) Finally, conditionally dependent writes are handled in much the same way as conditionally dependent reads.

If *lazy cancellation* [Jeff85, Jeff86, Fuji90a] is used then anti-messages are not sent out immediately, but rather are sent over time as reevaluation of the transaction occurs. In particular, as new values are computed for the writes, anti-messages that contain these new values are sent to the appropriate data objects. This optimization reduces the message traffic, since previously an anti-message followed by a new positive message had to be sent. Furthermore, lazy cancellation usually involves comparing the old message held in $OQ$ with the newly generated one, and only sending an anti-message if they are different. Incurring the cost of an extra comparison is often desirable [Fuji90b], as it may lead to fewer rollbacks and less message traffic.

### 4.5.4. Commit

Commitment occurs when the Global Virtual Time (*GVT*) exceeds the timestamp of the transaction [Jeff86]. At this time, external outputs may be physically performed and older versions may be purged. (Note, at least one committed version must be maintained, since readers may need this version.) Consequently, transactions are committed in timestamp order. Global Virtual Time (*GVT*) represents the maximum time beyond which rollbacks can not occur. As discussed above, rollbacks are initiated when a write message, $M_w$, is sent from some transaction object $z$ to some data object $x$. All of the anti-messages that propagate from $z$'s initial message will have larger timestamps than $z$ has.

*GVT* is defined as follows [Jeff86]: It is the minimum of:

1. All unprocessed messages (including those in queues awaiting processing, those in transit and those that are to sent out by transaction still in their read-write stage).
2. The current values of all timestamp generators (which typically utilize real-time clocks).

(See [Jeff85, Jeff86, Fuji90b] for specific techniques for computing/estimating *GVT* and generating timestamps.)

For a specification of this protocol that assumes nothing about the order of message delivery see [Para92].

### 4.5.5. Correctness of the Deferred Commitment Variant of TW

In this section, we give concise proofs of the correctness of the Deferred Commitment Variant. More detailed, traditional graph-theoretic proofs may be found in [Rame94].

**Lemma 1:** *It is impossible for* $h(r_k(x)) = x_i$ *if there exists* $x_j$ *such that* $ts(T_i) < ts(T_j) < ts(T_k)$.

Proof. Since read operations select the most recently (with respect to timestamp order) written version, the only potential way for $T_k$ to have read $x_i$ is for the write of version $x_j$ to have not occurred yet. This read will be frozen when $T_k$ commits, but it can not commit since we have

assumed that $T_j$ has a write operation yet to be processed, (i.e., $GVT \leq ts(T_j) < ts(T_k)$). Thus, this read, $r_k(x)$, is subject to rollback, and indeed will be when $w_j(x)$ is processed by data object $x$. Data object $x$ will notice that the read-stamp on version $x_i$ is larger than $ts(T_j)$, so that all premature readers of $x_i$ will be sent anti-messages, $-M_r$'s. Transaction $T_k$ will receive such an anti-message. When $T_k$ redoes its read of object $x$, it will now obtain $x_j$. □

**Proposition 1:** *Every schedule produced by the Deferred Commitment variant of the Time Warp protocol is 1SR.*

Proof. Let us consider schedule $S$ with its version function $h$. We claim that $h(r_k(x)) = x_j$ where $x_j$ is the immediately preceding version written in the standard serial schedule that corresponds to the timestamp order. Consider how this might not be the case. Firstly, $h(r_k(x))$ could map to a version later than $x_j$, but this is not possible since the protocol prohibits reading future versions (i.e., versions with larger timestamps). Secondly, $r_k(x)$ could have read a version earlier than $x_j$, but when $x_j$ is finally written, this read will via the rollback mechanism be "undone" and "redone" so that it now reads $x_j$, as indicated by Lemma 1. Therefore, the committed projection of $h$ for schedule $S$ is identical to the committed projection of the standard version function $h'$ for the standard serial schedule $S'$ which corresponds to executing the transactions one at a time in timestamp order. So by Definition 7, $S$ is 1SR. □

**Proposition 2:** *The Deferred Commitment variant of the Time Warp protocol enforces Recoverability.*

Proof. Transactions commit as *GVT* advances to their timestamp. Since a transaction only reads data from older transactions, at the time *GVT* reaches the transaction, all of the data it has read will have been committed. Therefore this protocol enforces Recoverability. □

Although this variant may introduce a significant amount of delay in waiting for commitment, it should not dramatically degrade throughput [Jeff86]. The main effect would be to

increase the transaction completion time. However, it is possible that a few very long duration transactions could increase the delay encountered by average transactions to an unacceptable level.

## 5. Hybrid Time Warp (HYT) Protocol

Each of the variants of Time Warp discussed so far has its weak points. An alternative to using one of these variants is to use a hybrid protocol. The hybrid would use Time Warp so long as all of the reads-in-question belong to uncommitted transactions, and use Multiversion Timestamp Ordering otherwise. Thus, when a *write of a new version* is attempted by $T_i$ three possible actions may ensue:

1. The first possibility is to *continue* normally -- the read-stamp is smaller than $ts(T_i)$ so there is no conflict.
2. The second possibility is to *partially roll back* the transactions with questionable reads -- there is a conflict and all of the reads-in-question belong to uncommitted transactions. Since rollbacks are less costly than aborts, this is the preferred corrective action.
3. The final possibility is to *abort* transaction $T_i$ -- there is a conflict and at least one of the reads-in-question belongs to a committed transaction. Unless one uses anti-transactions or substantially delays the commitment of transactions, rolling back transactions will not always suffice. If the new write would destroy one of the information transfers where the reader has already committed, then the write will be prevented from occurring by aborting the transaction and making it start over with a new, larger timestamp.

In responding to a conflict, the Hybrid Time Warp (HYT) protocol acts like Time Warp when it applies the second action (partial rollback), while it acts like Multiversion Timestamp Ordering when it applies the third action (abort).

### 5.1. Anti-Transaction Variant of HYT

Avoiding anti-transactions by choosing to abort, leads to additional complexities. To prevent aborts from corrupting the correctness of the database either a recovery protocol is needed, or again anti-transactions must be used (although less frequently). To see why this is so, consider the following case: Transaction $T_2$ reads a version written by transaction $T_1$, after which $T_1$ aborts. It is possible, even though $T_1$ started before $T_2$, that $T_2$ commits before $T_1$ finishes (or in this case aborts). Thus, $T_2$ has committed having read uncommitted (i.e. dirty) data, violating the recoverability condition.

### 5.2. Realistic Recovery Variant of HYT

Coupling a recovery protocol with the HYT protocol will solve this problem. Since we desire to keep blocking to a minimum, the Realistic Recovery (or Optimistic Recovery) protocol would make a good candidate [Grif85]. If a recovery protocol is not used then anti-transactions must be used to clean up the damage.

The *Realistic Recovery* protocol described in [Hadz83] is a good choice for a multiversion database. To describe the Realistic Recovery protocol, let us consider how recovery protocols must operate. Data that has been written by transactions that have yet to commit is termed *dirty data*. From the definition of recoverability, we can see immediately that recoverability must be enforced by not allowing commits of transactions that have read dirty data. To do this, we can block either reads of dirty data or we can block commits of transactions that have read dirty data. In either case we have to block; from prior simulation results [Grif85, Mill86], the better choice appears to be to block reading of dirty data. The Realistic Recovery protocol simply blocks transactions attempting to read dirty data, until that data is committed. Writes, however, are not blocked since they simply produce a new version. (See the Appendix for a more detailed specification of this protocol.) We will now specify in detail how the Realistic Recovery protocol can be combined with the HYT protocol.

### 5.2.1. Read

A read request message $M_r$ sent to $x$ by $T_i$ will be processed when $LVT(x)$ reaches $ts(T_i)$. (Straggler messages will be handled as specified in Section 4.5.1.) When such a read operation is processed, the most recent (from the time frame of the reading transaction) version is selected, call it $x_k$. If version $x_k$ is committed, then the read is dispatched; otherwise the read is blocked until version $x_k$ is committed.

If parallelism is exploited within transactions, then only the read operation will be blocked and not the entire transaction; other operations which are not dependent on this read may proceed. Note that this would still be the case even if only quasi-concurrency is used within transactions (i.e., a transaction is implemented as a process containing several coroutines).

Since reads may be dispatched in groups (or batches) (see Section 5.2.4), they are put on a dispatch list for efficient servicing. For example, once $x_k$ is found (e.g., in a buffer, referenced in battery-backed CMOS RAM, or copied into a buffer from a disk page), values obtained from it can be used to fill up reply messages $M_{val}$ for all the reads which were waiting for $x_k$ to commit.

The blocked reads will be kept track of by inserting them in timestamp order into $PSQ(x)$. Note, putting reads in a $PSQ$ is not necessary for the TW protocol since reads are not blocked. An alternative for the HYT protocol would be to not put them on a $PSQ$ per se, but to hang them off the uncommitted version they are attempting to read.

Since the read-stamp indicates the last transaction to have read a version, it will only be updated once a read is actually dispatched, i.e., waiting readers do not affect the read-stamp of a version. Note that if blocked reads are hung off an uncommitted version, they may need to be moved to hang off a new uncommitted version, while if reads are directly inserted into a $PSQ$, the ordering of the queue will take care of this automatically.

Assuming that a read request message $M_r$ is not involved in an abort or partial rollback, it will go through the following stages:

1. $M_r$ is enqueued in $IQ(x)$;
2. $M_r$ is processed and copied-to/referenced-from $PSQ(x)$;
3. $M_r$ may be blocked waiting for $x_k$ to commit;
4. $M_r$ is dispatched and copied-to/referenced-from $x$'s dispatch-list; a reply message $M_{val}$ will result when this dispatched message is serviced; and finally,
5. $M_r$ is committed.

In the simpliest case, a transaction would be suspended during the interval of time between sending the read request message and receiving the reply. A more sophisticated approach would allow other non-dependent operations (reads or writes) to proceed. Finally, reads may be purged when the write that they read is purged. Alternatively, reads may be purged when the data object receives a commit request message $M_c$ from the transaction. This choice may lead to a few false aborts [Soma94a].

### 5.2.2. Write

A write request message $M_w$ sent to $x$ by $T_i$ similarly will be processed when $LVT(x)$ reaches $ts(T_i)$. When a write operation is processed, it will succeed without conflict or corrective action if it does not interfere with an information transfer that has already occurred. This will be the case if the read-stamp of the previous version $x_j$ is less than or equal to the timestamp of the writing transaction $T_i$, i.e., $wts(x_j) \le rts(x_j) \le ts(T_i)$ The write will be carried out by inserting (in timestamp order) the new version $x_k$ into the $x$'s list of versions ($PSQ(x)$) and marking it uncommitted.

On the other hand, if $wts(x_j) < ts(T_i) < rts(x_j)$, then there is a conflict. The corrective action to be taken is just as specified at the beginning of Section 5. Notice that waiting reads do not produce conflicts. If a read by transaction $T_9$ was waiting for transaction $T_5$ to commit, while in the meantime transaction $T_8$ writes a new version of the same object, then transaction $T_9$ now simply waits for transaction $T_8$ to commit. Indeed, if there are no dispatched reads, then

$wts(x_j) = rts(x_j)$ which makes conflicts impossible. Therefore, the existence of a conflict implies that there is one or more dispatched reads whose timestamps are larger than $ts(T_i)$. If any one of them has committed, then $T_i$ must abort; otherwise all of these reads must be rolled back. If reads are purged when the transaction commits, we must maintain an additional read-stamp $rts\text{-}c(x_j)$ which indicates the timestamp of the latest committed read. In any case, maintaining $rts\text{-}c(x_j)$ allows the "partial rollback versus abort" decision to be made more efficiently.

Assuming that a write request message $M_w$ is not involved in an abort or partial rollback, it will go through the following stages:

1. $M_w$ is enqueued in $IQ(x)$;
2. $M_w$ is processed, marked uncommitted and copied-to/referenced-from $PSQ(x)$;
3. $M_w$ is marked committed; and finally,
4. $M_w$ is purged.

Unlike the situation for reads, the issuance of a write message need not suspend the transaction. It seems overly cautious to suspend the transaction until an acknowledgement $M_{ack}$ comes back from the data object. However, if early commits are allowed then some form of acknowledgement of the writes must be received before the commit can be allowed.

### 5.2.3. Rollback

When a transaction is to be rolled back, just like for the Deferred Commitment variant of Time Warp, the read-in-question will need to be "redone", and a determination will be made as to what other read operations need to be "undone" or "done for the first time". Similarly, writes that are conditionally dependent on the read-in-question will need to be either "undone" or "done for the first time". Finally, any writes that are value dependent on the read-in-question will need to be "undone" and then "redone". As previously discussed, both actions can be handled by one anti-message if lazy cancellation is used.

The amount of rollback work to be done using the HYT protocol is relatively small, since

as shown in Proposition 5 (Section 5.2.7), partial rollbacks will not cascade. Therefore, the only messages that can trigger a partial rollback are $-M_{val}$ messages. Recall that under TW these messages initiate rollbacks, but $-M_w$ messages drive the cascade. Finally, if deferred updates as opposed to immediate updates are used, then $-M_w$ will never need to be used, so that only reads will need to be "undone/redone".

### 5.2.4. Commit

When a transaction commits, the version of each object it has written will be marked as committed. Any reads that were waiting on these versions will now be dispatched. A committed version may be purged when it can be assured that no additional reads will need this version. This will be the case when Global Virtual Time ($GVT$) has swept past this version's and a subsequent version's timestamp.

An optimization can be introduced to allow transactions to commit before $GVT$. This is accomplished by introducing a Two-Phase Commit (2PC) protocol. When a transaction has finished its read-write stage, it seems reasonable to try to commit it. The problem is that it may receive an anti-read message telling it that one of its reads is now invalid and needs to be redone. This is why transactions wait for $GVT$. The alternative is to send a commit request message, $M_{c-ok}$, to all data objects read. If responses come back saying that no rollback has been initiated, $M_{c-yes}$, then the transaction can commit. It is not necessary to send a no vote, since the fact a data object cannot vote yes implies that it has already or is about to send a $-M_{val}$ message. Clearly, once a data object responds affirmatively, it must consider the read to be committed when making a decision on whether to abort a new incoming write.

Transaction $T_i$ carries out the commit by sending $M_c$ messages to all data objects it has written (sent a version to). This allows waiting reads to be dispatched.

After $T_i$ commits, it can purge its input queue and eliminate all but its $M_c$ messages from its output queue. It will then send a commit final, $M_{cf}$, message to the fossil collection coordinator which is responsible for calculating/estimating $GVT$ as well as initiating purges.

### 5.2.5. Fossil Collection/*GVT* Calculation

The fossil collection coordinator maintains a data structure (e.g., a balanced search tree) to facilitate calculating/estimating *GVT*. Each node in the tree represents a transaction and contains the transaction's object id, its timestamp, and a flag indicating whether it has committed. The nodes are ordered by timestamp and are placed in the tree when a start message $M_s$ is received from a transaction. The node is marked uncommitted. Upon receiving the $M_{cf}$ message, the coordinator will mark $T_i$ as committed. Then the minimum timestamp transactions will be examined and removed until an uncommitted one is found. An $M_{ip}$ message will be sent to each transaction so removed. This informs the transaction that it may initiate purging, which is accomplished by the transaction sending $M_p$ messages to all the data objects it has written. After the transaction has initiated purging, all its space may be freed up. The data objects will then purge all versions prior to the one written by the initiating transaction.

Using this very simple approach, it is possible for an $M_s$ message to arrive at the coordinator with a timestamp less than *GVT*. This can happen because a timestamp generator is behind the others, or an $M_s$ is in transit. If this happens, the transaction must be aborted and restarted with a new larger timestamp. In cases where this leads to too many aborts, a tunable parameter could be subtracted from *GVT*.

A more accurate although more costly alternative ...

<< Somal is working on this >>.

### 5.2.6. Abort

When a transaction aborts, all of the versions which it has written must be removed from relevant version lists (*PSQ*s). This is accomplished by sending $-M_w$ messages to all involved data objects. Any reads that were waiting for the transaction to commit will now be assigned to the immediately preceding version. If that version is already committed, then these reads will be dispatched. Notice that *cascaded aborts* are not possible, since transactions are not allowed to read uncommitted (dirty) data.

A more detailed specification of the HYT protocol using Realistic Recovery may be found in [Soma94a].

### 5.2.7. Correctness of the Realistic Recovery Variant of HYT

In this section, we give concise proofs of the correctness of this variant of HYT. Again, more detailed, traditional graph-theoretic proofs may be found in [Rame94].

**Proposition 3:** *Every schedule produced by the Realistic Recovery variant of HYT is 1SR.*

Proof. Let us consider schedule $S$ with its version function $h$. We claim that $h(r_k(x)) = x_j$ where $x_j$ is the immediately preceding version written in the standard serial schedule that corresponds to the timestamp order. Consider how this might not be the case. Transaction $T_k$ in issuing read operation $r_k(x)$ could potentially read a version earlier (older) than $x_j$, later (younger) than $x_j$, or even later (younger) than itself. These possibilities are shown in Figure 5. Let us address each possibility in turn.

1. Under the first possibility, $h$ could map read $r_k(x)$ to a (committed) version earlier than $x_j$, e.g.,

$$h(r_k(x)) = x_i$$

   where $ts(T_i) < ts(T_j)$. This could happen temporarily, if $T_k$ reads $x$ before the version $x_j$ is written. However, when $x_j$ is finally written/committed this situation will be corrected. If the reading transaction, $T_k$, has not committed, the read-in-question, $r_k(x)$, will via the rollback mechanism be "undone" and "redone" so that $T_k$ now reads $x_j$. Otherwise, if $T_k$ has committed, its read of $x_i$ is frozen. However, since the protocol will abort $T_j$, or for that matter any other version with a timestamp between $ts(T_i)$ and $ts(T_k)$, $h$ still maps to the correct version, as $x_i$ is really the immediately preceding version in the standard serial schedule.

2. Under the second possibility, $h$ could map read $r_k(x)$ to a version later than $x_j$, e.g.,

$$h(r_k(x)) = x_l$$

where $ts(T_l) > ts(T_j)$. Although, $r_k$ could be waiting to read $x_l$, the read would not be dispatched until $T_l$ commits. However, $T_l$ never commits, since by our assertion that $x_j(x)$ is the immediately preceding version in the standard serial schedule, $T_l$ must at some time abort (and be resubmitted with a new larger timestamp), so that $r_k(x)$ will be rescheduled to $x_j$.

3. Under the third possibility, $h$ could map read $r_k(x)$ to a version later than itself, e.g.,

$$h(r_k(x)) = x_m$$

where $ts(T_m) > ts(T_k) > ts(T_j)$. This possibility can never occur at any time, since the protocol prohibits reading future versions (i.e., versions with larger timestamps).

Therefore, the committed projection of $h$ for schedule $S$ is identical to the committed projection of the standard version function $h'$ for the standard serial schedule $S'$ which corresponds to executing the transactions one at a time in timestamp order. So by Definition 7, $S$ is 1SR. □

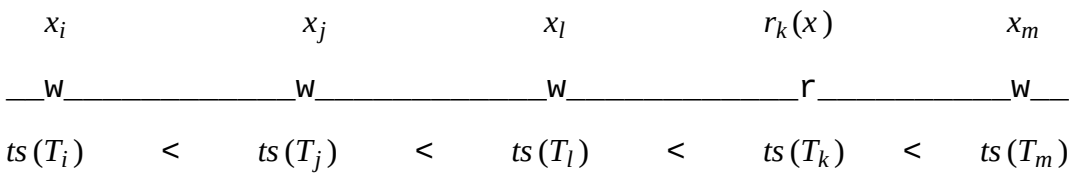


**Figure 5: Reading the Correct Version.**

**Proposition 4:** *The Realistic Recovery variant of HYT enforces Recoverability.*

Proof. All reads are blocked until the version to be read is committed. Therefore, transactions may not read any uncommitted (dirty) data, i.e., the protocol enforces recoverability. □

**Proposition 5:** *The Realistic Recovery variant of HYT eliminates the possibility of cascaded*

*rollbacks.*

Proof. A rollback can only cascade from transaction to transaction when a write is "undone", and another transaction has read this write's uncommitted version. However, since reading of uncommitted versions is prohibited by the Realistic Recovery variant, rollbacks will never cascade to other transactions. □

**Proposition 6:** *The Realistic Recovery variant of HYT is deadlock-free.*

Proof. For the HYT using Realistic Recovery, only reads can be blocked. A read by transaction $T_i$ can be blocked by waiting for an earlier transaction $T_j$ to commit its writes. Transaction $T_j$ itself can be blocked waiting on other transactions to commit, but each of these must be earlier than $T_j$ (i.e., have smaller timestamps). Consequently, it is impossible for a circular wait condition to develop, so this protocol is deadlock-free. □

## 6. Conclusions

In the field of simulation, the Time Warp protocol has been shown to be one of the better protocols for parallel and distributed simulation [Fuji90a, Fuji90b, Jeff91]. In this paper, we have given detailed specifications for hybrids and variants of the Time Warp protocol. These variants are extensions or modifications of Jefferson's Time Warp database protocol [Jeff86]. To reduce the frequency of rollbacks and lessen the delay in waiting for commitment, we have created a hybrid protocol, in the hopes that it would combine the best features of Time Warp and Multiversion Timestamp Ordering.

For two of the protocols, the Deferred Commitment variant of Time Warp (TW), and the Realistic Recovery variant of Hybrid Time warp (HYT), we have shown that they enforce both serializability and recoverability. In addition, we have shown that the Realistic Recovery variant of HYT has the following desirable properties:

1. No Cascaded Partial Rollbacks;

2. No Cascaded Aborts; and

3. No Deadlocks.

(Note that, it is trivially true that Time Warp is deadlock-free, since read and write operations are never blocked.)

Finally, we have considered several issues for improving the performance of these protocols:

1. exploiting language characteristics to make rollback and in particular state savings more efficient,

2. utilizing lazy cancellation to again make rollbacks more efficient and reduce message traffic,

3. utilizing dependency patterns within transactions to reduce the number of operations that need to be "undone" and/or "redone",

4. exploiting intra-transaction parallelism (the more rapidly a transaction can execute the less likely it is to produce a timestamp conflict), and

5. using two-phase commit to allow early commits.

At this point, let us summarize some of our results from our previous performance modeling studies [Mill91a, Liu 92, Mill92, Mill94, Soma94b]. In [Liu92], it was found that multiversion protocols performed substantially better than their single version equivalents, if the system had the resources necessary for high concurrency (e.g, a multiprocessor systems with plenty of processors around to advance the states of transactions). The performance of Time Warp (TW) was compared to Multiversion Timestamp Ordering (MVTO) in [Mill91a, Mill94] assuming a high dependency scenario, while in [Mill92] Time Warp (TW), Hybrid Time Warp (HYT) and Multiversion Timestamp Ordering (MVTO) were compared assuming a low dependency scenario. The combination of results from these papers indicates that Time Warp exhibits performance superior to that of Multiversion Timestamp Ordering. Furthermore, as the degree of dependency within transactions is reduced, the cost of aborting a transaction becomes much

greater than the cost of partially rolling back a transaction, thereby giving Time Warp an even bigger advantage over traditional protocols. On the negative side, Time Warp is susceptible to a type of data contention thrashing. When the concurrency level and probability of conflict are very high, the frequency of rollbacks begins to explode leading to a sharp downturn in throughput. The problem of cascaded rollbacks escalates to the point where more work is done in reversing time than in progressing time. By throttling the concurrency level, though, high performance for Time Warp can be maintained. Results from [Mill92] further indicate that the Hybrid Time Warp protocol performs even better than Time Warp, with higher throughput and fewer corrective actions. Although the difference is not as great as that between Time Warp and Multiversion Timestamp Ordering, it appears to be consistent. When a write produces a conflict, a decision must be made to either rollback the reader(s) or abort the writer. Evidently, HYT makes the right choice more often than either Time Warp or Multiversion Timestamp Ordering, thus accounting for its better performance.

## 7. References

[Agha86] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, MA, 1986.

[Agra85] R. Agrawal and D.J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, vol. 10, no. 4, December 1985, pp. 529-564.

[Allc83] J.E. Allchin, *An Architecture for Reliable Decentralized Systems*, Ph.D. Thesis, Georgia Tech, September 1983.

[Bern81] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, vol. 13, no. 2, June 1981, pp. 185-222.

[Bern83] P.A. Bernstein, N. Goodman and V. Hadzilacos, "Recovery Algorithms for Database Systems," Technical Report-10-83, Harvard University, March 1983.

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

[Bili89] A. Biliris, "A Data Model for Engineering Design Objects," *Proceedings of the IEEE Second International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Gaithersburg, MD, October 1989, pp. 49-58.

[Cell88] W. Cellary, E. Gelenbe and T. Morzy, *Concurrency Control in Distributed Database Systems*, North-Holland, Amsterdam, 1988.

[Chan79] K.M. Chandy and J. Misra, "Distributed Simulation: A Case Study in the Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, May 1979, pp. 440-452.

[Chan89] K.M. Chandy and R. Sherman, "Space, Time, and Simulation," Proceedings of the SCS Multiconference on Distributed Simulation, March 1989, pp. 53-57.

[DeWi92] D.J. DeWitt and J.N. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, vol. 35, no. 6, June 1992, pp 85-98.

[Fuji90a] R.M. Fujimoto, "Optimistic Approaches to Parallel Discrete Event Simulation," *Transactions of the Society for Computer Simulation*, vol. 7, no. 2, 1990, pp. 153-191.

[Fuji90b] R.M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, October 1990, pp 30-53.

[Garc83] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed System," *ACM Transactions on Database Systems*, vol. 8, no. 2, June 1983, pp. 186-213.

[Garz88] J.F. Garza and W. Kim, "Transaction Management in an Object-Oriented Database System," Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, IL, June 1988, pp. 37-45.

[Grah84] M.H. Graham, N.D. Griffeth and B. Smith-Thomas, "Reliable Scheduling of Transactions on Unreliable Systems," *Proceedings of the 1984 Conference on Principles of Database System*, Waterloo, Canada, 1984, pp. 300-310.

[Gray81] J.N. Gray, et al., "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, vol. 13, no. 2, June 1981, pp. 223-242.

[Grif84] N.D. Griffeth and J.A. Miller, "Performance Modeling of Database Recovery Protocols," *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, Silver Spring, MD, October 1984, pp. 75-83.

[Grif85] N.D. Griffeth and J.A. Miller, "Performance Modeling of Database Recovery Protocols," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 6, June 1985, pp. 564-572.

[Hadz83] V. Hadzilacos, "An Operational Model for Database System Reliability," *Proceedings of the 1983 Conference on Principles of Distributed Computing*, 1983, pp. 244-257.

[Jeff82] D.R. Jefferson,

[Jeff85] D.R. Jefferson, "Virtual Time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, July 1985, pp. 404-425.

[Jeff86] D.R. Jefferson and A. Motro, "The Time Warp Mechanism for Database Concurrency Control," *Proceedings of the Second International Conference on Data Engineering*, Los Angeles, CA, February 1986, pp. 474-481.

[Jeff87] D.R. Jefferson,

[Jeff91] D.R. Jefferson and P. Reiher, "Supercritical Speedup," *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, April 1991, pp. 159-168.

[Kim90] W. Kim, *Introduction to Object-Oriented Databases*, The MIT Press, Cambridge, MA, 1990.

[Koch93] K.J. Kochut and J.A. Miller, "WarpLog: Time Warped Logical Objects," *Proceedings of the 26th Annual Simulation Symposium*, Arlington, VA, April 1993, pp. 30-39.

[Liu92] X. Liu, J.A. Miller and N.R. Parate, "Transaction Management for Object-Oriented Databases: Performance Advantages of Using Multiple Versions," *Proceedings of the 25th Annual Simulation Symposium*, Orlando, FL, April 1992, pp. 222-231.

[Mill86] J.A. Miller, *Markovian Analysis and Optimization of Database Recovery Protocols*, Ph.D. Thesis, Georgia Tech, August 1986.

[Mill91a] J.A. Miller and N.D. Griffeth, "Performance Modeling of Database and Simulation Protocols: Design Choices for Query Driven Simulation," *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA, April 1991, pp. 205-216.

{Mill91b] J.A. Miller, K.J. Kochut, W.D. Potter, E. Ucar and A.A. Keskin, "Query Driven Simulation Using Active KDL: A Functional Object-Oriented Database System," *International Journal in Computer Simulation*, vol. 1, no. 1, 1991, pp. 1-30.

[Mill92] J.A. Miller, "Simulation of Database Transaction Management Protocols: Hybrids and Variants of Time Warp," *1992 Winter Simulation Conference Proceedings*, Arlington, VA, December 1992, pp. 1232-1241.

[Mill94] J.A. Miller and N.D. Griffeth, "Performance of Time Warp Protocols for Transaction Management in Object-Oriented Systems," *International Journal in Computer Simulation*, vol. 4, no. 3, 1994, 259-282.

[Misr86] J. Misra, "Distributed Discrete-Event Simulation," *ACM Computing Surveys*, vol. 18, no. 1, March 1986, pp. 39-65.

[Papa77] C.H. Papadimitriou, P.A. Bernstein and J.B. Rothnie, "Computational Problems Related to Database Concurrency Control," *Proceedings of the Conference on Theoretical Computer Science*, Waterloo, Canada, 1978, pp. 275-282.

[Para92] N.R. Parate, *Object-Oriented Databases: The Time Warp Protocols for Concurrency Control*, M.S. Thesis, University of Georgia, June 1992.

[Rame94] D. Ramesh *Hybrid Transaction Management Protocol: Combining Time Warp and Multiversion Timestamp Ordering*, M.S. Thesis, University of Georgia, March 1994.

[Reed78] D.P. Reed, *Naming and Synchronization in a Decentralized Computer System*, Ph.D. Thesis, MIT, September 1978.

[Reed83] D.P. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, vol. 1, no. 1, Feb. 1983, pp. 3-23.

[Soma94a] P. Somal and J.A. Miller,

[Soma94b] P. Somal, M.S. Thesis, University of Georgia, June 1994.

[Ullm82] J.D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD, 1982.

# 8. Appendix

## 8.1. Realistic Recovery Protocol

In this appendix, we explain in more detail how the Realistic Recovery [Hadz83, Grah84] protocol works. The Realistic Recovery protocol requires that each object maintain a list of operations that have been requested (some of which may have also been executed) and that each transaction maintain a list of objects at which it has requested write operations. When a read or write operation is requested of an object, the operation is added to the object's list of operations. When a transaction requests a commit or abort, it notifies each object at which it has requested a write. The object may then remove some writes from its list of operations or dispatch some reads. Committed writes must be left on the operation list until they are immediately followed by another committed write. Reads are removed once they have been dispatched. The following summarizes how each operation is handled:

1. Read. Dispatch the read if it is immediately preceded by a committed write operation. Otherwise, add the read to the end of the operation list.
2. Write. Add the write operation (including the value to be written) to the end of the operation list. Mark the write operation as uncommitted.
3. Commit. Notify the objects to mark all write operations of the transaction as committed. If any of these writes is immediately preceded by a committed write, the earlier committed write is removed from the operation list. If any of these writes is immediately followed by one or more read operations, the reads are dispatched.
4. Abort. Notify the objects to remove all write operations of the transaction. If after removal, any committed write is immediately preceded by another committed write, the earlier one may be removed. And if any committed write is immediately followed by one or more reads, the reads may be dispatched.

### 8.2. Local Causality

*Local causality* is the primary correctness condition that must be enforced in parallel and distributed simulations. The executions of events by (logical) processes must be constrained so that cause-and-effect relationships are not violated. (An event is an instantaneous occurrence that typically changes the state of the simulation.) If in reality, one event causes another event to occur, then in any simulation the former event must be executed first. A good formal definition of this correctness condition is given by Fujimoto [Fuji90a]:

*Definition 9. Local Causality Constraint:* A discrete event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging timestamped messages, obeys the local causality constraint if and only if each LP processes events in non-decreasing timestamp order.

Note that the role of time in simulation is more significant than it is for databases. Unless a database has a historical element, time is used simply as a tool to order the execution of transaction operations. Local causality is similar in character to serializability. The chief difference is that local causality demands that the concurrent execution of events must be equivalent to one given total ordering, while serializability only demands equivalence with any one of a collection of total orderings (corresponding to serial schedules). An additional difference is that local causality does not distinguish between conflicting and non-conflicting events. For example, if the order of two query (read-only) events [Fuji90b] is swapped, this will have no detrimental effect, yet formally it violates local causality. However, some parallel simulation mechanisms can take advantage of this, e.g., Time Warp with lazy reevaluation or jump forward [Fuji90b]. Events and Transactions are analogous in one respect, they are both execute instantaneously in virtual (or simulated) time. However, they model different notions in the real world: A real-world event is instantaneous; whereas, transactions exist over time, albeit relatively short.