1-1-1991

Virtual time synchronization in distributed database systems

Timothy E LeMaster
University of Nevada, Las Vegas

Order Number 1345925

Virtual time synchronization in distributed database systems

LeMaster, Timothy E., M.S.

University of Nevada, Las Vegas, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

VIRTUAL TIME SYNCHRONIZATION IN DISTRIBUTED DATABASE SYSTEMS

by

Timothy E. LeMaster

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science
in
Computer Science

Department of Computer Science

University of Nevada, Las Vegas

August 1991

The thesis of Timothy E. LeMaster for the degree of Master of Science in Computer Science is approved.

Ajoy Kumar Datta

Chairperson, Ajoy Kumar Datta, Ph.D

L. P. Gewali

Examining Committee Member, Laxmi Gewali, Ph.D

Kazem Taghva

Examining Committee Member, Kazem Taghva, Ph.D

Graduate Faculty Representative, Larry J. Paulson, Ph.D

Graduate Dean, Ronald W. Smith, Ph.D

University of Nevada, Las Vegas

August, 1991

ABSTRACT

Distributed systems synchronized by Virtual Time have been topics of recent interest. Virtual Time follows an optimistic philosophy relying on rollback for synchronization instead of abortion or blocking. Although many applications have been suggested as candidates for Virtual Time, few were simulated or implemented. This research reports on the first implementation and results of a Distributed Database Management System synchronized by virtual time. We argue that virtual time is a viable alternate concurrency control method for distributed database systems if its memory overhead can be absorbed.

ACKNOWLEDGEMENTS

This thesis is dedicated to my mother Pat, my father Lou, and to my brother David. They are a continual source of strength and encouragement. I am extremely grateful for the superb guidance I received from my advisor, Dr. Ajoy K. Datta. The quality of this research would not have been as high without his direction. I wish to thank the UNLV College of Engineering systems staff including Jay Nietling, Greg Wohletz, Brian Wohletz, Jeff Gilbreth, and also to Cindy Baum and John Kilburg. Special thanks to committee members Dr. Kazem Taghva, Dr. Laxmi Gewali, and Dr. Larry J. Paulson. Most of all, I thank God for giving me the wisdom and patience required to complete this work and the degree program.

Contents

List of Figures

Chapter 1

INTRODUCTION

It was suggested by Jefferson [33] that virtual time could be used to synchronize a distributed database management system. He described one method in [35]. Some related theoretical work was done by Witkowski [63]. No one has actually implemented such a system to the knowledge of this writer. Therefore, a logical next step would be to implement such a system and measure its costs and performance characteristics. This will assist in determining the practicality of a virtual time synchronized database system. Additionally, the implementation of a traditional alternate method of concurrency control such as MultiVersion Timestamp Ordering (MVTO) is desirable for performance comparison. The purpose of this research is to report on the implementation of a virtual time distributed database system as well as its relative performance against MVTO.

1.1 Distributed Database Management Systems

A distributed database (DDB) can be defined as a set of interrelated databases distributed over a computer network [49]. A distributed database management system (DDBMS) can then be defined as software that manages the distributed database and makes the distribution transparent to any users. One way to visualize the architecture of a DDBMS is shown in Figure 1.1. The four major components of a DDBMS at each site are transactions, transaction manager (TM), database manager (DM), and data. Transactions are generated from

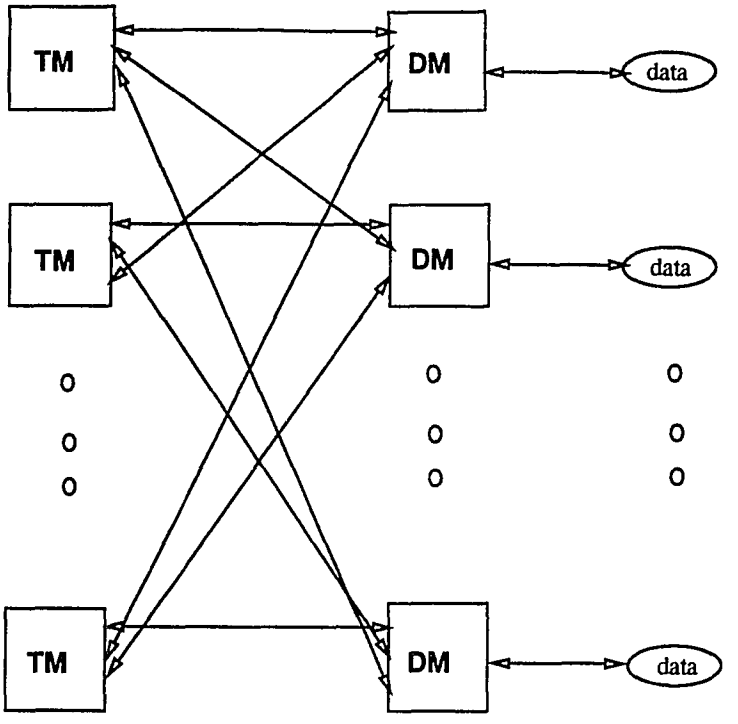


Figure 1.1: One possible architecture for a DDBMS.

users. The transaction manager supervises interaction between users and the DDBMS while the database manager controls access to the data. A fundamental component not shown in Figure 1.1 is some type of recovery manager [37] (this was not considered in this research).

There are many problem areas associated with a DDBMS [49, 16]. Some of these include:

- Database Design
- Query Optimization
- Directory Management
- Deadlock Management
- Reliability
- Operating System Support
- Concurrency Control

Of these, concurrency control is one of the most studied [49]. The focus of this research is on a new method of concurrency control (also called synchronization).

1.2 Concurrency Control Problem

The basic problem in concurrency control is to safeguard the correctness of a database updated by interleaved transactions [51]. The problem is compounded in a distributed environment since the data and user transactions are located at different sites. A concurrency control mechanism is required to insure correct transaction execution while exploiting parallelism as much as possible. The most accepted way to characterize correctness is through serializability. The execution of two interleaved transactions is serializable if it produces the same effect on the database as some serial (non-interleaved) execution of the two transactions [10]. Serialization is discussed in depth in [10, 16, 50, 8, 49, 51, 9]. Another goal of concurrency control is to ensure the atomic transaction execution. Atomicity in this context can be described as follows:

- Each transaction accesses shared data without interfering with other transactions.
- Either all or none of the transaction effects are made permanent.

Some examples follow in the next section to illustrate the need for concurrency control.

1.2.1 Lost Updates Anomaly

A lost update happens when two transactions are allowed to be interleaved without control. The interleaving may allow two different transactions to read the same value. Both may later attempt to write based on that value. One of these will be overwritten and lost. An example follows to clarify this. Two transactions are defined as follows:

T_1: Read(x) T_2:Read(x)

x=x+1000 x=x-500

Write(x)	Write(x)
Commit	Commit

Transaction T_1 adds 1000 to the current value of x while T_2 subtracts 500 from it. Both will write a new value for x. Assume the initial value of x is 2000 and the following interleaved sequence is executed:

T_1:Read(x)	!gets value 2000
T_1:x=x+1000	
T_2:Read(x)	!gets value 2000
T_1:Write(x)	!writes value 3000
T_1:Commit	
T_2:x=x-500	
T_2:Write(x)	!overwrites value 1500
T_2:Commit	

In the above sequence, the value read by T_2 is incorrect since T_1 read it previously and intends on modifying it. T_2 eventually writes the value 1500 over the the value 3000 written by T_1. This causes the T_1 update to be lost.

1.2.2 Inconsistent Retrieval Anomaly

This anomaly could occur when a read oriented transaction is interleaved with a read and write transaction with no concurrency control mechanism in place. Consider the following example for transactions T_1 and T_2 :

T_1: Read(x)	T_2: Read(x)
x=x-1000	Read(y)
Write(x)	Print (x+y)
Read(y)	Commit
y=y+1000	
Write(y)	
Commit	

Transaction T_1 transfers 1000 from x to y. T_2 reads x and y, then prints their sum. Assume the initial values of both x and y are 2000 and the following interleaved sequence is executed:

T_1:Read(x)	!gets value 2000
T_1:x=x-1000	
T_1:Write(x)	!writes value 1000
T_2:Read(x)	!gets value 1000
T_2:Read(y)	!gets value 2000
T_2:Print(x+y)	!prints 3000
T_2:Commit	
T_1:Read(y)	!gets value 2000
T_1:y=y+1000	
T_1:Write(y)	!writes value 3000
T_1:Commit	

In the above sequence, T_1 is half way through the transfer operation when T_2 reads x and y. T_2 prints 3000 as the sum of x and y when in fact it should be 4000. T_1 has not executed the last half of its operations and T_2 is unaware of this fact. This allows T_2 to make inconsistent retrievals. As shown by these two examples, a concurrency control method is required to prevent database inconsistency.

During the 70's and 80's, several different approaches to database concurrency control were introduced [10, 16, 49, 50]. In

recent years, the availability and cost of distributed hardware has made the use of distributed databases practical. The next section presents a new method of distributed database concurrency control based on Jefferson's concept of virtual time[33].

1.3 Virtual Time

Virtual Time was initially conceived by David Jefferson and Henry Sowizral at the Rand Corporation [36]. The original goal of their research was to improve the technology for distributed computer simulation. Virtual Time is implemented with the Time Warp (TW) mechanism [33]. Throughout this text, the terms virtual time and Time Warp are used synonymously.

Distributed synchronization is the key component to a distributed simulation. Distributed synchronization methods can be broken into two broad categories: conservative and optimistic. A conservative approach [18, 19, 46, 53, 25] does not perform an action until complete information arrives to prove it is correct. This approach generally includes some form of blocking to insure this ordering. Blocking can create possible deadlock situations where two or more processes are waiting for messages from each other. All deadlock issues must be resolved in the problem model, adding significantly to its complexity. An optimistic approach [38, 39, 33, 25] gambles that an action should occur and performs it. If it is later determined the action was wrong, a correction must occur which takes into account all direct and indirect effects of the incorrect action. Virtual time is a notable optimistic method.

Virtual time is a temporal coordinate system imposed on a distributed system to define

synchronization and to measure computational progress. This approach relies on rollback as its fundamental synchronization mechanism. Virtual time allows a process to send a message (or request) to any other process at any time. Dynamic process interaction is permitted instead of static declarations. A relaxed communication assumption is used in which messages are not required to be sent and delivered in the same order. Messages are processed by a virtual time process as soon as they are received, regardless of their timestamp and sender. There is no waiting when messages are available for processing. Therefore, deadlock is not possible. If a message arrives with a timestamp less than one previously processed, the process rolls back, undoes previous work, and processes the newly arrived and the previously received messages in timestamp order. The rollback mechanism requires process state histories, commonly called checkpoints, and previous messages be stored in memory to assist in returning to a previous time. This memory and its manipulation costs are the primary overheads in a virtual time system.

Some advantages of the Time Warp mechanism include hardware independence and relative programmer independence. Overall, this optimistic approach offers great potential for speedup since future projections are made instead of blocking. Current limitations include diminishing returns as more processors are added in efforts to obtain greater speedups and unbalanced loading due to the lack of a general load management algorithm. Another potential limitation is the delay incurred from event completion to commitment caused by optimistic processing.

The core of virtual time (Time Warp) is discussed in depth in [40, 36, 33, 34]. Some enhancements and optimizations are described in [4, 27, 26, 23, 24, 28, 29, 44, 58, 60, 61]. Some performance studies and simulation results can be found in [3, 5, 6, 11, 22, 30, 32, 31, 34, 45, 54, 62].

There are several applications that can benefit from an optimistic, virtual time approach. The most predominant of these is distributed simulation. This is by far the most widely practiced use of virtual time. A Time Warp Operating System has been designed to accommodate distributed simulations, but it can support any application synchronized by virtual time [34]. Other applications that have been examined as virtual time candidates include distributed logic programming [42, 20], computer animation [55], and distributed

databases [35, 63].

Virtual Time was suggested as a possible candidate for distributed databases in [33]. It was introduced as a new method of concurrency control in DDBMS in [35]. Some analytical work on distributed databases including virtual time was presented in [63]. Some advantages of virtual time in a DDBMS include lack of starvation or deadlock. Entire transactions are not aborted and restarted. Instead, individual actions within a transaction are rolled back when conflict occurs. Multiversion history information is available since the rollback mechanism requires it. Disadvantages include commitment in strict virtual time order (which may lead to long commit delays) and very large memory overhead.

Jefferson noted five major differences from other concurrency control methods in [35]. They are summarized here and follow:

- 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.
- Rollback is used to resolve access conflicts instead of locking or abort-and-retry.
- Deadlock and starvation are not possible.
- It is free from requirements such as predeclaration, sequentiality, or two-phase structure.
- It uses an artificial time scale (called virtual time) from which timestamps are generated.

History information is stored in two queues. These queues are required to retain copies of all requests sent to and received from an object for some item. This creates a historical view of that item. Transactions are assigned unique timestamps at initiation. When a transaction request is received, it is processed without blocking. A read request is satisfied by returning a certain write value found in one of the queues. A write request is achieved by retaining a copy of the request. If a read or write request arrives late, or out of order, rollback must be invoked. This may cause previously satisfied requests to be undone and recomputed.

Transaction commitment is usually non-deterministic due to delays and race conditions. In virtual time, commitment is defined at transaction initiation (and therefore deterministic). Each transaction is assigned a unique timestamp and transaction are committed in timestamp order. When a virtual time system commits a transaction, it is insuring that all actions within that transaction were performed correctly. Virtual time should be considered serializable. A proof that this is true is beyond the scope of this research.

1.4 MultiVersion Timestamp Ordering

Concurrency control mechanisms can be broken into two broad classes: pessimistic and optimistic [49]. Pessimistic algorithms synchronize concurrent transaction execution early in their life cycle. Optimistic algorithms delay synchronization until the end of execution. Some pessimistic algorithms include two-phase locking (2PL), basic timestamp ordering (BTO), and multiversion timestamp ordering (MVTO), and conservative timestamp ordering. Optimistic ones include some form of verification or certification [38, 16]. Virtual time would also fall in this class.

MVTO was chosen to compare performance for several reasons. They are as follows:

1. It maintains a multiversion history similar to virtual time.

2. It relies on blocking and abortion which is opposite virtual time philosophy.

3. It is a well known method and much literature is available.

MVTO is basically a variant of BTO. The addition of a multiversion history allows all read requests to be successfully completed. This is the primary advantage of this method. Some disadvantages include memory overhead for the multiversion history and the possibility of cyclic restart (or starvation). Algorithms, theory, and rules for MVTO can be found in [16, 10, 8, 49, 57, 7, 9, 51].

MVTO generates a view of some data item as a sequence of versions or histories. Each data item has a read list and a write list associated with it. These lists store the history information. A read list contains timestamps for read requests while the write list holds (timestamps, value) pairs for write requests. All requests are processed on a first-come

first-served basis. Each request has a timestamp associated with it. Requests do not have to be processed in timestamp order. Synchronization may require delaying a request or aborting a transaction if processing the request would create a database inconsistency. A read request is accomplished by retrieving a certain version of the item. A write request is achieved by adding a new version to an item. These lists and versions are transparent to the user accessing the data items.

In the next section, virtual time is formally introduced and described as a concurrency control mechanism for distributed database systems. Section 3 presents the MVTO concurrency control method. In Section 4, the implementation environment is discussed. The performance results for both are presented in Section 5. Finally, a conclusion is offered.

Chapter 2

VIRTUAL TIME

SYNCHRONIZATION

Virtual time is a temporal coordinate system imposed on a distributed system to define synchronization and to measure computational progress. It can be visualized as a global, one-dimensional temporal coordinate system [33]. The axis of this system is overlaid upon the distributed computation. The virtual time axis is typically oriented with real numbers from zero to positive infinity. There are two important properties of virtual time. First, it may increase or decrease with respect to real time. There is no restriction on the amount or frequency by which it changes. Second, the virtual times generated must form a total (or partial) ordering on the relation "less than."

A manager (or process) may send a message to any other manager at any time. There are no restrictions placed upon potential communication paths between processes. No fixed or static declarations are required before execution begins. The communication medium is assumed to be reliable, but messages are not required to arrive in the order they are sent.

Virtual time incorporates several different data structures and control algorithms to produce optimistic processing. The most noteworthy is the use of rollback as the fundamental synchronizer in the system. The data structures include queues for storing messages, a unique process id, a local clock, and a permanent value. The algorithms include a trans-

action manager and a database manager. The former is responsible for generating and submitting requests while the latter processes the requests. The rollback mechanism is used by both manager. The data structures, rollback mechanisms, and control algorithms are presented in this chapter.

2.1 Data Structures

2.1.1 Messages

A transaction request or response is represented by a message as shown in Figure 2.1. The components of a message are as follows:

Sender: value representing sending process's identifier

Receiver: value representing the receiving process's identifier

Timestamp: virtual time assigned to the transaction

Type: type of message

Sign: field indicating whether or not this is an antimessage

Text: type dependent data field

The sender and receiver components contain node addressing information consisting of unique node id values. The timestamp contains a unique virtual time value. This value is created in a two step process. First, the wall clock time in units of seconds is read from the local system clock. The clocks at the local nodes are not synchronized. Virtual time does not require this. It is possible that two nodes read their clocks and record the same value. Therefore, the unique node pid is shifted into the lower 5 bits of the timestamp field to create a unique value. The possible message types are Read, Write, Read-Response, and Commit. The sign component indicates whether the message is an antimessage. In this database system, antimessages are only required for Write and Read-Response messages. Antimessages are transmitted only when a manager is required to rollback. Read operations are not rolled back since their is no causal relationship between them. Commit message are

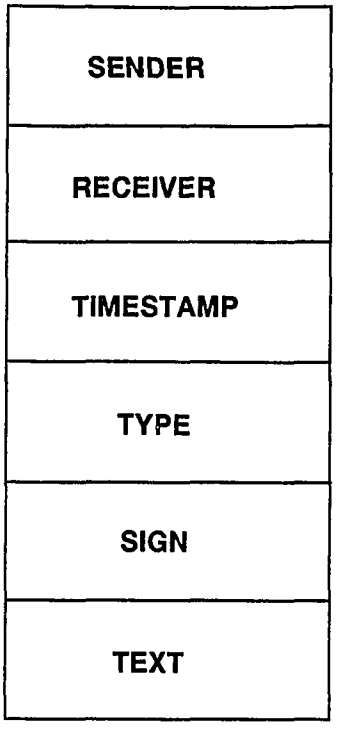


Figure 2.1: Components of a Virtual Time message.

informational only and cannot cause rollback. The text field contains different information based on the message type. A Read message has an item associated with it. A Write message has an item and a value associated with it. A Read-Response message returns data requested from a Read message. This response has a value associated with it.

2.1.2 Database Manager Internals

The Database manager consists of five major components for each item it manages. These components contain all the information needed to support distributed processing and roll-back. They are as follows:

Pid: unique process identifier for a node

Local Clock: current local virtual time of an item

Permanent Value: last committed Write value

Input Queue: processed messages from the virtual past along with unprocessed messages in manager's virtual future for an item

Output Queue: antimessage copies of all Read-Response messages generated and sent by the manager

Figure 2.2 depicts a possible configuration for an item. The Pid is 1. This manager adjusts its local virtual time by setting its local clock equal to the timestamp of the message being processed. In this example, the virtual time is 100 since it is processing the message in the input queue with timestamp 100. The permanent database value is 'p.' The input queue and output queue are ordered by timestamps. The input queue contains eight total messages. There are five processed messages with timestamps in the virtual past (timestamps less than 100). It also contains two unprocessed messages in the virtual future (timestamps greater than 100). Four output messages have been sent by this process since there are four antimessages in the output queue.

The output queue, and one portion of the input queue are necessary to support the rollback function. They create additional space overhead not found in other distributed synchronization methods. Additional time overhead will be incurred when these queues are manipulated. Information must be stored in these queues after event occurrence and on a periodic basis as messages are received and sent. These overheads are the major drawbacks of virtual time synchronization approach.

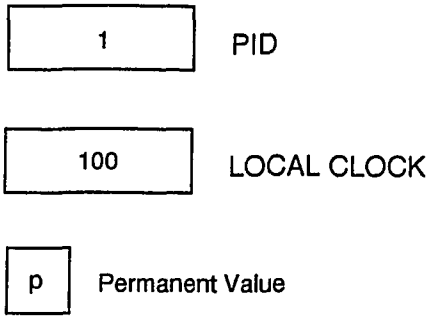
2.1.3 Transaction Manager Internals

The Transaction manager consists of three primary components. These components hold the information required for rollback and distributed processing. They are as follows:

Pid: unique process identifier for a node

Local Clock: current local virtual time of the manager

Output Queue: antimessage copies of all Write messages generated and sent by the manager



INPUT QUEUE

4	3	2	2	3	5	3	3	Sender
1	1	1	1	1	1	1	1	Receiver
50	60	80	80	90	100	120	120	Timestamp
R	R	R	W	R	R	R	W	Type
+	+	+	+	+	+	+	+	Sign
a	b	c	d	e	f	g	h	Text

OUTPUT QUEUE

1	1	1	1	Sender
4	3	2	3	Receiver
50	60	80	90	Timestamp
Rr	Rr	Rr	Rr	Type
-	-	-	-	Sign
z	y	x	w	Text

Figure 2.2: Virtual Time Database Manager where the message with timestamp 100 in the Input Queue is being processed currently.

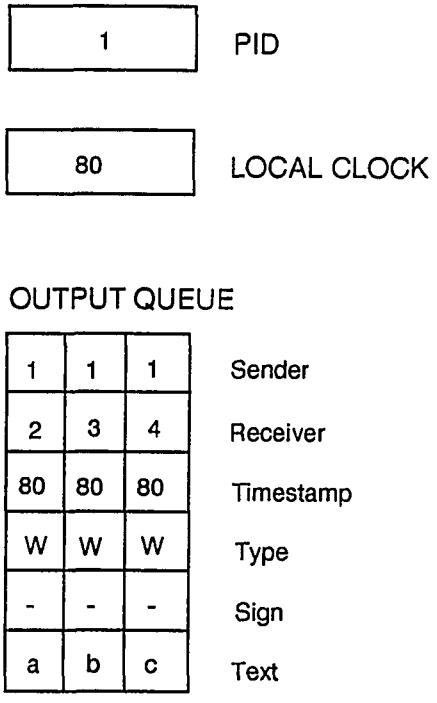


Figure 2.3: Virtual Time transaction manager where the local time is 80 and three Write messages have been issued.

Figure 2.3 depicts a possible configuration. The Pid is 1. This manager adjusts its local virtual time by setting its local clock equal to the timestamp assigned to the current transaction. In this example, the virtual time is 80. The output queue is ordered by timestamps. The queue contains three messages. They represent Writes that have been sent to three nodes. Since the transaction manager generates the Read and Write requests, no Input Queue is needed. If the manager received requests from an actual user, an Input Queue could be incorporated in a manner similar to the preceding description for the database manager input queue.

The output queue is necessary to support the rollback function. It creates an additional space overhead not found in other distributed synchronization methods. Additional time overhead will be incurred when this queue is manipulated.

2.1.4 Clocks

In a virtual time environment, every process object has its own local virtual clock. In a virtual time distributed database system, every transaction manager has its own local virtual clock, and every item managed by the database manager has its own virtual clock. These local clocks are not synchronized with any traditional method. They are allowed to move forward or backward in virtual time. A local clock for a database item moves forward while processing messages and backward when rolling back. The local clock for the transaction manager is adjusted forward after the completion of a transaction. Timestamps are created by reading the local system clock and shifting the unique pid into the lower 5 bits. Although real time-oriented timestamps were used in this research, any ordered set could have been used. The global clock is used primarily for overhead management and commitment. It is not needed very often and is therefore computed and used on a periodic basis. Local and global clocks are fully discussed in the following sections.

2.2 Local Control

Both Time Warp managers have common system tasks that need to be performed. These tasks include managing messages, clocks, queues, events, and rollback. All the tasks are

discussed here with the exception of rollback. A separate section is dedicated entirely to rollback.

2.2.1 Message Management

The only method of communication between managers at nodes is through message passing. When a message for an item arrives at the database manager, it is placed in an input queue. This queue has two distinct parts. The first part contains unprocessed messages for the virtual future. The second part contains processed messages from the virtual past. In Figure 2.2, the current time is 100. Therefore, all messages in the input queue with timestamps less than 100 represent the virtual past while the others with timestamps greater than 100 represent the virtual future. Before a newly arrived message is placed into the input queue, the sign component is examined to determine whether the message is an event message (sign is $+$) or an antimessage (sign is $-$). The two possible types of messages are handled differently.

The first possible type of message is an event message. Upon arrival, the database manager places it in the input queue in timestamp order. If the message's timestamp is in the manager's virtual past, the manager must rollback to a virtual time earlier than the timestamp. Execution begins at this earlier time, insuring that all messages are processed in timestamp order. The rollback mechanism is invoked to undo all the work done after the timestamp. If the timestamp is in the manager's virtual future, nothing else is required since the message has been queued. The database manager then resumes processing messages in the input queue.

The other possibility is that the newly arrived message is an antimessage. This implies that some transaction manager has rolled back and is undoing work that should not have been done. The only purpose of the antimessage is to seek out its corresponding event message and cancel it and any side effects it may have caused. The timestamps of the antimessage and its corresponding message are the same. The only difference between the two messages is the content of the sign field. If the antimessage's timestamp is in the manager's virtual future, the corresponding message is found in the virtual future part of the input queue. It is removed so it will not be processed. If the antimessage's timestamp is in the

manager's virtual past, the manager must be rolled back. A manager must rollback to a virtual time earlier than the antimessage's timestamp. All the work done after the antimessage's timestamp must be undone. In either case, the antimessage and its corresponding message eventually cancel each other out. A section on rollback explains and gives examples of the antimessage and rollback.

The output queue contains antimessages of messages the manager has sent. This queue is ordered by the timestamp of the antimessage. In Figure 2.2, four antimessages are in the output queue. All the work completed by a manager is conditional and could be rolled back. The output queue is used strictly to support rollback. If a manager rolls back, antimessages may be sent out to cancel the original messages.

Initially, messages are saved in the two aforementioned queues. It is not necessary to keep all this information throughout the entire distributed computation. If a process cannot roll back to a virtual time prior to a time associated with a queue entry, the entry's information is not needed. This cutoff point is known as the global virtual time (GVT). It is unnecessary to save any information with a virtual time less than the GVT. The reader may wish to skip ahead for specifics of the GVT.

The database manager can be considered a task that executes the following loop:

```
While(!global_termination)
    Wait(IQ);
    Switch(Message type);
        CASE READ:
            Create Reply
            Find Latest Write
            Set value in Reply
            Create Reply Antimessage
            Insert into OQ
```

```
        Send Reply
    CASE WRITE:
        No Action Yet
End_While
```

As long as unprocessed messages are in the input queue (IQ), the database manager processes them. When a Read message is processed, the input queue is searched for the latest Write. If no Write is found, the value recorded permanently in the database is used. A response message with this value is generated and sent. An antimessage copy is stored in the output queue (OQ) in case rollback is required later. A Write message is simply marked as processed. The physical write operation must be delayed until the write is committed. This occurs when the GVT is larger than the timestamp of the Write. In Figure 2.2, the next message to be processed is the Read (type field = 'R') with timestamp 120. This Read will be satisfied with the Write value found in the Write message (type field = 'W') with timestamp 80 since it is the latest, previous write. The three Read Messages with timestamps 50, 60, and 80 were satisfied with the permanent database value.

The transaction manager can be considered a task that executes the following loop:

```
While(!global_termination)
    Generate Read Set;
    Issue Reads and wait for Replys;
    Generate Write Set;
    Issue Writes;
    Wait for Commit;
End_While
```

This loop runs until a fixed number of transactions are completed. This condition indicates global termination in this research. At any time, the task may get interrupted upon receiving a Read-Response antimessage. This alters the computation path. This subject is discussed in detail in the section on rollback.

2.2.2 Clock Management

The synchronization of clocks in a virtual time environment is not typical of other distributed methods. The local virtual clock for a database manager is allowed to move ahead or behind in virtual time. The clock moves ahead by processing messages in its input queue. It moves behind by getting rolled back by an incoming message timestamped in the virtual past. The local clock for the transaction manager always moves forward as transactions are completed. This clock could also move backward if the characteristics of a different transaction generator required it. The local clocks of all managers are at best, loosely synchronized. Overall, it is expected that the local virtual clock advances ahead in time while occasionally jumping behind.

The local clock for a database manager item is driven by setting its value equal to the timestamp of the message currently being processed. Every time a new message is processed, the clock is advanced. A newly arrived message from the virtual past causes rollback. This forces the clock to jump back in virtual time to a lessor value. The clock is set to $+infinity$ when the database manager for an item has no more input messages to process. This denotes manager termination. This can be a temporary condition since it will unterminate if a new message is received.

2.3 Roll Back

It is possible for a message to arrive out of timestamp order in a virtual time system. The reason for this is the lack of artificial blocking. When this out of order message called a straggler arrives, the manager may need to roll back in virtual time to process the message

in timestamp order. Rollback operations include returning to a previous state and undoing work that should not have been done. In this research, specialized rollback mechanisms for both the database manager and the transaction manager were developed.

2.3.1 Database Manager Roll Back

The rollback mechanism for the database manager handles late arriving Read, Write, and Write antimessages. Each of these messages is handled differently. This diverges from traditional rollback mechanisms [29, 33] and follows a method based on optimized semantics [61]. The possible messages are handled as follows:

Read: Insert into input queue and treat as normal Read message.

Write: Insert into input queue and mark as processed. Send Read-response antimessages for any previously affected Read-response messages.

Write antimessage: Remove corresponding Write from input queue. Send Read-response antimessages for any previously affected Read-response messages.

Rollback is not required for Read messages since this operation has no causal conflict with any other operation. For the late arriving Read, find the latest, previous Write and return this value. When a Write message is received, any processed Read messages with timestamps larger than the late Write, but less than the next future Write (if any) must be reprocessed. This is done by cancelling the Read-response messages and sending out a new response message. The same message is used in this research. When a Write antimessage is received, the corresponding Write messages must be annihilated. Any Read-response message which was satisfied with this value must be cancelled also. The net effect after rollback must be to leave the database in a state as if the late message had arrived in order. Antimessages might get sent to accomplish this, but are not always necessary. Some examples of rollback follow.

Assume the database manager is in the state shown in Figure 2.2. If a straggler Read message with timestamp 85 arrives, the message is inserted into the input queue in timestamp order and processed. A Read-response message is sent, and a copy of it is placed

in the output queue. The state of the queues after this rollback has occurred is shown in Figure 2.4(a).

Suppose the database manager is in the state shown in Figure 2.2. If a straggler Write message with timestamp 50 arrives, the message is inserted into the input queue in timestamp order and processed. Two Read messages with timestamps 60 and 80 were affected by this late Write. The values returned earlier are no longer correct. Therefore, antimessages for both are transmitted. The new value will be sent along in the antimessage. The queue entries are updated with the new value (and retained). Figure 2.4(b) show the queues after this operation has occurred.

Assume the database manager is in the state shown in Figure 2.2. If a Write antimessage message with timestamp 80 arrives, its corresponding Write in the input queue must be removed. One Read message with timestamp 90 needs to be corrected since it used a value that no longer exists. Thus, an antimessage with the correct value is sent. The queue entry is updated with the new value (and retained). The state of the queues after this has occurred is shown in Figure 2.4(c).

2.3.2 Transaction Manager Roll Back

The rollback mechanism for the transaction manager handles Read-response antimessages only since it only receives Read-response messages. There are two cases to consider when a message of this type is received. They are as follows:

No Writes Sent: Record new Read-response value.

Writes Sent: Record new Read-response value. Transmit Write antimessages for all Writes sent. Transmit new Write messages.

If Write messages have been sent, they need to be cancelled. A new value has just arrived that is assumed to have an affect on the contents of the Write value. After cancellation, new Write messages must be sent. If no Writes have been sent, nothing needs to be cancelled. It is assumed the Reads have no interdependencies.

(a)

INPUT QUEUE

4	3	2	2	5	3	5	3	3
1	1	1	1	1	1	1	1	1
50	60	80	80	85	90	100	120	120
R	R	R	W	R	R	R	R	W
+	+	+	+	+	+	+	+	+
a	b	c	d	e	f	g	h	i

Sender
Receiver
Timestamp
Type
Sign
Text

OUTPUT QUEUE

1	1	1	1	1
4	3	2	5	3
50	60	80	85	90
Rr	Rr	Rr	Rr	Rr
-	-	-	-	-
z	y	x	u	w

Sender
Receiver
Timestamp
Type
Sign
Text

(b)

INPUT QUEUE

4	4	3	2	2	3	5	3	3
1	1	1	1	1	1	1	1	1
50	50	60	80	80	90	100	120	120
R	W	R	R	W	R	R	R	W
+	+	+	+	+	+	+	+	+
a	l	b	c	d	e	f	g	h

Sender
Receiver
Timestamp
Type
Sign
Text

OUTPUT QUEUE

1	1	1	1
4	3	2	3
50	60	80	90
Rr	Rr	Rr	Rr
-	-	-	-
z	y	x	w

Sender
Receiver
Timestamp
Type
Sign
Text

transmitted

(c)

INPUT QUEUE

4	3	2	3	5	3	3
1	1	1	1	1	1	1
50	60	80	90	100	120	120
R	R	R	R	R	R	W
+	+	+	+	+	+	+
a	b	c	e	f	g	h

Sender
Receiver
Timestamp
Type
Sign
Text

OUTPUT QUEUE

1	1	1	1
4	3	2	3
50	60	80	90
Rr	Rr	Rr	Rr
-	-	-	-
z	y	x	w

Sender
Receiver
Timestamp
Type
Sign
Text

transmitted

Figure 2.4: Examples of rollback at a database manager. (a) After processing a late Read message with timestamp 85. (b) After processing a late Write message with timestamp 50. (c) After processing a Write antimessage with timestamp 80.

2.4 Global Control

Virtual time success is embedded in Global Control issues. These include measuring global progress, managing memory, controlling message flow, committing irreversible actions, and taking snapshots. The focal point of all these issues is the global virtual time (GVT). It is the single most important virtual time concept because of its global impact, especially on global progress.

2.4.1 Global Clock

Every manger in a virtual time system has a local clock. These clocks contain different virtual times and may jump forward or backward in time. No single local clock can be used as an indicator of global progress. Therefore, a global clock containing the GVT is needed. The GVT is estimated on a periodic basis since it is used primarily for overhead management. The GVT is based on the status of the distributed system.

The GVT at real time r is defined by Jefferson [33] to be the minimum of :

1. All virtual times in the local clocks at time r, and

2. All virtual send times of messages sent but not processed at time r.

The GVT serves as an *absolute floor* past which no process can roll back.

The above definition is not a practical one. The classic problem with this definition is that it requires an instantaneous global snapshot of the distributed system including messages in transit. It has been shown that computing a global snapshot is non-trivial [17]. It would require information from every process at real time r. It is not possible to stop every process at real time r and record this information under the virtual time synchronization policy. Operational definitions have been developed and are discussed in [33, 40].

A specialized GVT algorithm was developed for this implementation. The approach used is a non-traditional one. The transaction manager controls all the timestamps in the system. This manager cannot rollback beyond its current local time. The database manager does not generate new timestamps. It only uses copies of them sent from transaction

managers. Therefore, only the transaction managers need to be involved in the GVT computation. Periodically, every transaction manager sends its current local virtual time to a predefined node. This node, in addition to a transaction manager and a database manager, is responsible for running a GVT algorithm. This node collects the local clock values from the nodes, then periodically computes the GVT and broadcasts it to all nodes. The GVT is computed by selecting the least local virtual time from all the values sent. The GVT value broadcast is slightly out-of-date, but does not requires distributed synchronization.

This algorithm is not a good one when compared with others [33, 44, 58]. Its weaknesses include increased message traffic and a single node responsible for computing this value. It also relies on the characteristics of the transaction and database model used in this implementation. The algorithm did provide acceptable performance results for this implementation.

The frequency of estimating the GVT is a performance parameter that affects response time, throughput, and memory requirements. Small frequency intervals lead to faster response times and better memory utilization but also use more computing time and network bandwidth. Thus, there are tradeoffs involved in selecting this interval value.

2.4.2 Memory Management

Time Warp requires much more memory to support rollback than other synchronization methods. Each manager stores history information in case of rollback. A manager does not need to store all of this information from the beginning of the computation if it cannot possibly roll back that far into the past. The GVT is defined to be lowest possible virtual time to which a process could roll back. Therefore, some messages and states in the input, output, and state queues may no longer be necessary. Any message in the input queue with a timestamp less than the GVT can be discarded. If the message is a Write, its value is recorded as the new permanent value. Similarly, any message in the output queue with a virtual send time less than the GVT can be discarded. The discarded memory can be recovered and reallocated to future requests. The term *fossil collection* [33] is used to describe this memory recovering technique.

Efficient memory management in Time Warp is critical. Applications execute faster

when a maximum amount of rollback information is stored. Fossil collection contributes greatly to reclaiming memory for this use. Other techniques not needed in this research are presented in [28, 29, 40].

2.4.3 Commitment

The nature of rollback in Time Warp restricts a class of operations from being performed immediately. These operations include output to permanent storage. A commitment protocol has been established to manage this class of operations.

A Time Warp system requires that no irreversible action be committed until it can be proven correct. To enforce this requirement, any operation in this class must be buffered (not committed) until the virtual time associated with it is less than the GVT. The buffered operation is proven to be correct once the GVT is greater than its virtual time because rollback can no longer affect it.

The Time Warp commitment protocol is more rigid than other commitment protocols. In this implementation, commitment occurs during fossil collection. When a Write is encountered, the value is written to the permanent database, and a commit message is sent to its transaction manager. The command buffering done prior to commitment may cause a large delay from the time computed to the time committed. This potential delay is not considered a major drawback of virtual time, but specific applications could suffer from it. One of the objectives of this research is to determine whether these delays are large and adversely affect user response time.

2.5 Costs

Three factors contribute to the overall cost of a virtual time system. They are memory requirements, message traffic, and execution time. Each of these can be manipulated by the virtual time implementor to tune the system for optimal performance.

2.5.1 Overhead

As previously stated, large memory overhead is needed to support the rollback mechanism. This memory stores history information used by a manager to return to the virtual past. The output queue and virtual past portion of the input queue comprise this memory overhead. All input and output queue messages must be saved.

Message traffic can be caused by several operations. When rollback occurs, any antimessages transmitted increase message traffic. These antimessages may also cause rollback at their receivers, inducing even more traffic. Estimating the GVT also increases message traffic on a periodic basis. As the total message traffic increases, the probability of rollback also increases because of message delivery assumptions.

Execution overhead involves the above two overheads. Processing time is needed to manage the rollback history memory. When rollback occurs, processing time is required to stop and restart computations. Processing time is required to estimate the GVT and to do fossil collection, commitment, and other GVT related concerns. The processing time added by overhead tasks may be significant enough to impact the overall performance of the system.

2.5.2 System Tuning

The virtual time implementor is responsible for tuning system to achieve maximum performance. The most critical parameter setting is the GVT frequency. This value is commonly expressed in seconds. A low value leads to better response times and memory management, but requires more processing time and increases message traffic. A high value has the exact opposite effect. In this research, the GVT for every simulation run was individually tuned to achieve minimum wall clock execution times.

In examining the cost of Time Warp, the notion of distributed rollback may lead to the immediate conclusion of exorbitant cost. The temporal locality principle is cited as the primary reason disputing this [33]. Applications executed are expected to follow this principle. Under this presumption, rollback is infrequent and the cost is not as high as initially anticipated.

Chapter 3

MULTIVERSION TIMESTAMP ORDERING SYNCHRONIZATION

MVTO is one of several methods available for achieving distributed database synchronization. The MVTO version used in this research is based on Bernstein's algorithm [8]. This method is similar to Reed's method [56, 57].

A manager in a MVTO system may send a message to any other manager at any time. There are no restrictions placed upon potential communication paths between processes. No fixed or static declarations are required before execution begins. The communication medium is assumed to be reliable, and messages are required to arrive in the order they are sent. Note this last condition was not required for virtual time.

MVTO utilizes several different data structures and control algorithms to produce synchronization. Transaction abortion is the fundamental synchronizer in this system. The data structures, abortion mechanisms, and control algorithms are presented in this chapter.

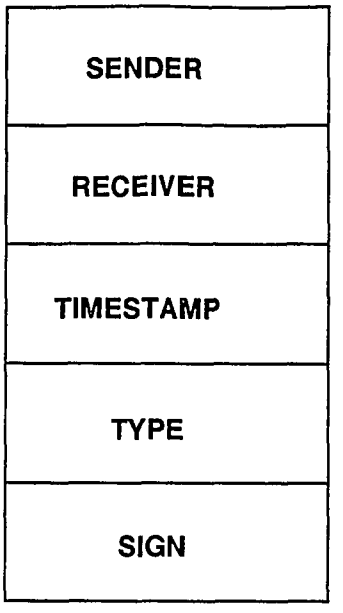


Figure 3.1: Components of a MVTO message.

3.1 Data Structures

3.1.1 Messages

A transaction request or response is represented by a message similar to the one presented for virtual time. The message is shown in Figure 3.1. The only difference with virtual time is the sign field is not needed with this method. The values that the message components contain are the same also except for message type. The possible types for this method include Read, Prewrite, Write, and Abort, and Read-response.

3.1.2 Database Manager Internals

The Database manager consists of five major components for each item it manages. These components contain all the information needed to incorporate MVTO synchronization policy. They are as follows:

Pid: unique process identifier for a node

Read Buffer: unprocessed Read messages

Read List: timestamps of processed Read messages

Prewrite Buffer: accepted Prewrite messages

Write List: timestamps and values of processed Write messages

Figure 3.2(a) depicts a possible configuration for an item. The Pid is 1. All queues and buffers are ordered by timestamps. The Read buffer is empty. The Read list has 3 processed messages. The Prewrite buffer has 1 message while the Write list has 1 message in it. The Read buffer contains Read messages that are not able to be processed yet. The Read list holds the timestamps of processed Read messages. The Prewrite buffer stores Prewrite messages that have been accepted by the Database manager. The Write list contains the timestamp and values pairs which represent versions of the database. Buffers hold work to be done while lists represent work already done. Both types of structures are required to accomplish correct synchronization.

3.1.3 Transaction Manager Internals

The Transaction manager consists of three major components. These components contain all the information needed to support MVTO processing. They are as follows:

Pid: unique process identifier for a node

Clock: current local time of the manager

Output List : addresses of all messages generated and sent by the manager

The Pid is the same as mentioned earlier. The clock contains a value generated when a transaction was initiated. The output list records the nodes involved in the transaction. This list is needed in case the transaction gets aborted by some Transaction manager which is involved. Transaction abortion is discussed fully in a future section.

3.1.4 Clocks

Clocks are only needed by the transaction managers under MVTO policy. Each transaction is assigned a unique timestamp. If a transaction aborts, it is resubmitted with a new

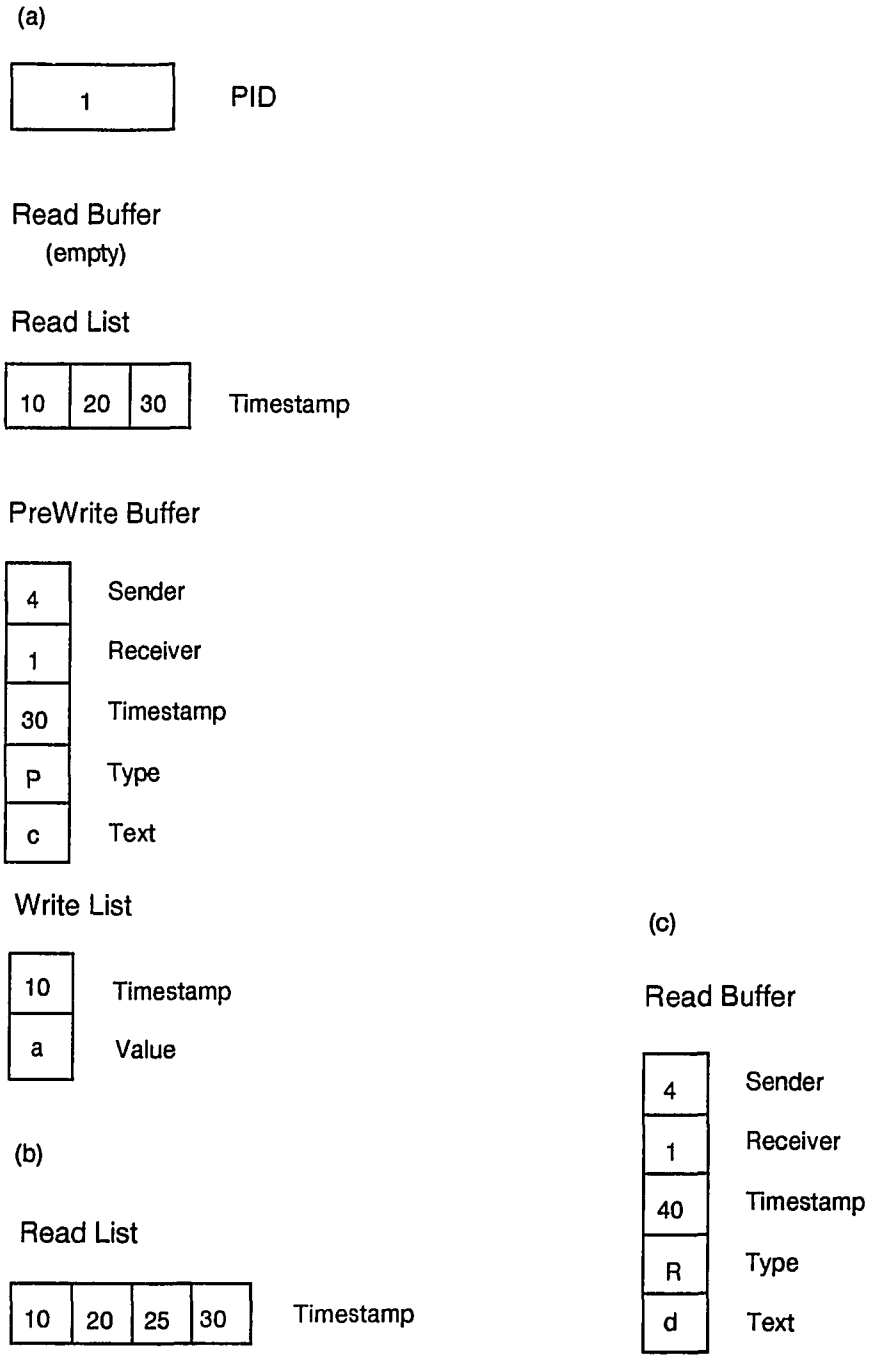


Figure 3.2: Examples of a MVTO database manager processing Read messages. (a) State prior to processing. (b) State after processing a Read message with timestamp 25. (c) State after processing a Read message with timestamp 40.

timestamp. This cycle repeats until the transaction is successfully completed. Timestamps in this system are also based on real-time, but clock synchronization was not considered. The synchronization of local clocks may be important for specific applications. If required, some clock synchronization algorithm would have run along with transaction and database managers at the nodes[17]. Timestamps are created by reading the local system clock and shifting the unique pid into the lower 5 bits. This is the same as the virtual time method.

3.2 Local Control

All MVTO managers have common system tasks that need to be performed. These tasks include managing messages, buffer, lists, and abortion. All the tasks are discussed here with the exception of abortion. A separate section is dedicated entirely to it.

3.2.1 Database Manager Message Management

When a message for an item arrives at the database manager, it is processed immediately. The database manager can be considered a task that executes the following loop:

```
While(!global_termination)
    Wait(Message);
    Switch(Message type);
        CASE READ:
            If(Read In Interval Of PBuf)
              Buffer Read
            Else
              Create Reply
              Find Latest Write
```

```
            Add Timestamp to Read List
            Send Reply
        CASE PREWRITE:
            If(Any Reads In Interval Of Prewrite)
              Send Abort
            Else
              Buffer Prewrite
        CASE WRITE:
            Add (Timestamp,Value) to Write List
            Debuffer Corresponding Prewrite
            Check Read Buffer
End_While;
```

When a Read message is received, it is either processed or buffered. The message must be buffered if it is dependent on a Prewrite that has been accepted. Otherwise, a Read-response message with the latest value of the item is created and sent and the timestamp of the message is added to the Read list. If a Prewrite message is received, the manager must either accept or reject it. The only reason to reject a prewrite request is if time conflict exists with some previously granted Read operation. This is how MVTO resolves conflict. It causes an entire transaction to be aborted. Not only will the request at this manager be resent with a new timestamp, but every other manager in the system that was involved also. If no conflict exists, the Prewrite is accepted and buffered. A Write message is sent to a manager after it is clear that the Prewrite was accepted. (This is accomplished by timeouts in this implementation). When a Write arrives, the corresponding Prewrite is removed from the Prewrite buffer. The Write timestamp and value pair is added to the Write list. The Read buffer is checked next. It is possible that some Reads were buffered because of the

Prewrite that was just satisfied. Some examples follow.

Assume a Read message with timestamp 25 arrives at the database manager shown in Figure 3.2(a). The Read can be satisfied immediately. A copy of the message timestamp is added to the Read list as shown afterwards in Figure 3.2(b). If the timestamp were 40 instead, it would be buffered since the value to be returned depends on a Prewrite at time 30. In this case, the message would be added to the Read buffer as shown in Figure 3.2(c).

Suppose a Prewrite message with timestamp 15 arrives at the database manager shown in Figure 3.3(a). The Prewrite must be rejected since a Read message with timestamp 20 has already been granted. This is presented in Figure 3.3(b). If the Prewrite timestamp were 20, it would be accepted and buffered since there is no conflict. The Prewrite buffer for this case is shown in Figure 3.3(c).

Assume a Write message with timestamp 30 arrives at the database manager shown in Figure 3.4(a). The corresponding Prewrite message in the Prewrite buffer is removed and the Write is added to the Write list. The Read with timestamp 40 can now be processed and then added to the Read list. The final states are shown in Figure 3.4(b).

3.2.2 Transaction Manager Message Management

The transaction manager can be considered a task that executes the following loop:

```
While(!global_termination)
    Generate Read Set;
    Issue Reads and wait for Replys;
    Generate Write Set;
    Issue Prewrites;
    Wait For Timeout Period;
    If(Not Aborted);
        Issue Writes;
```

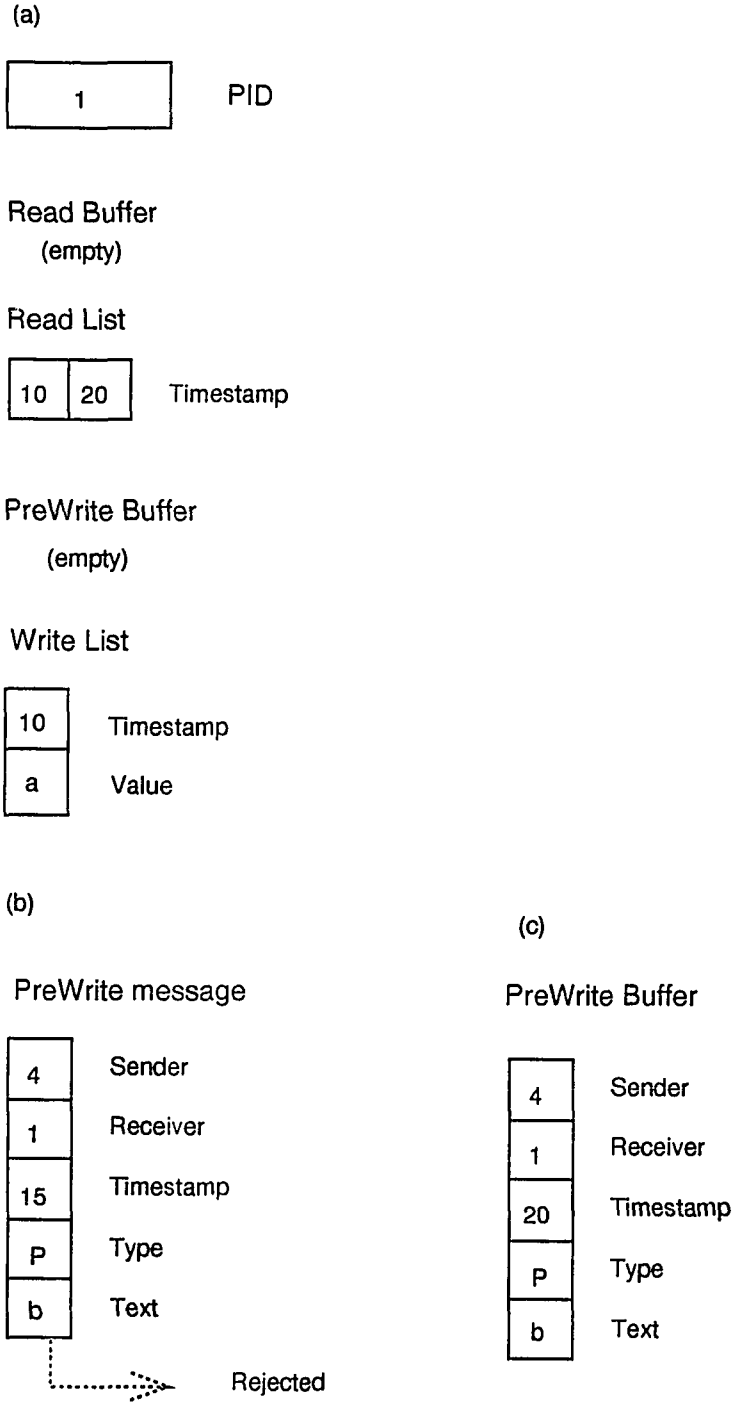


Figure 3.3: Examples of a MVTO database manager processing Prewrite messages. (a) State prior to processing. (b) State after processing a Prewrite message with timestamp 15. (c) State after processing a Prewrite message with timestamp 20.

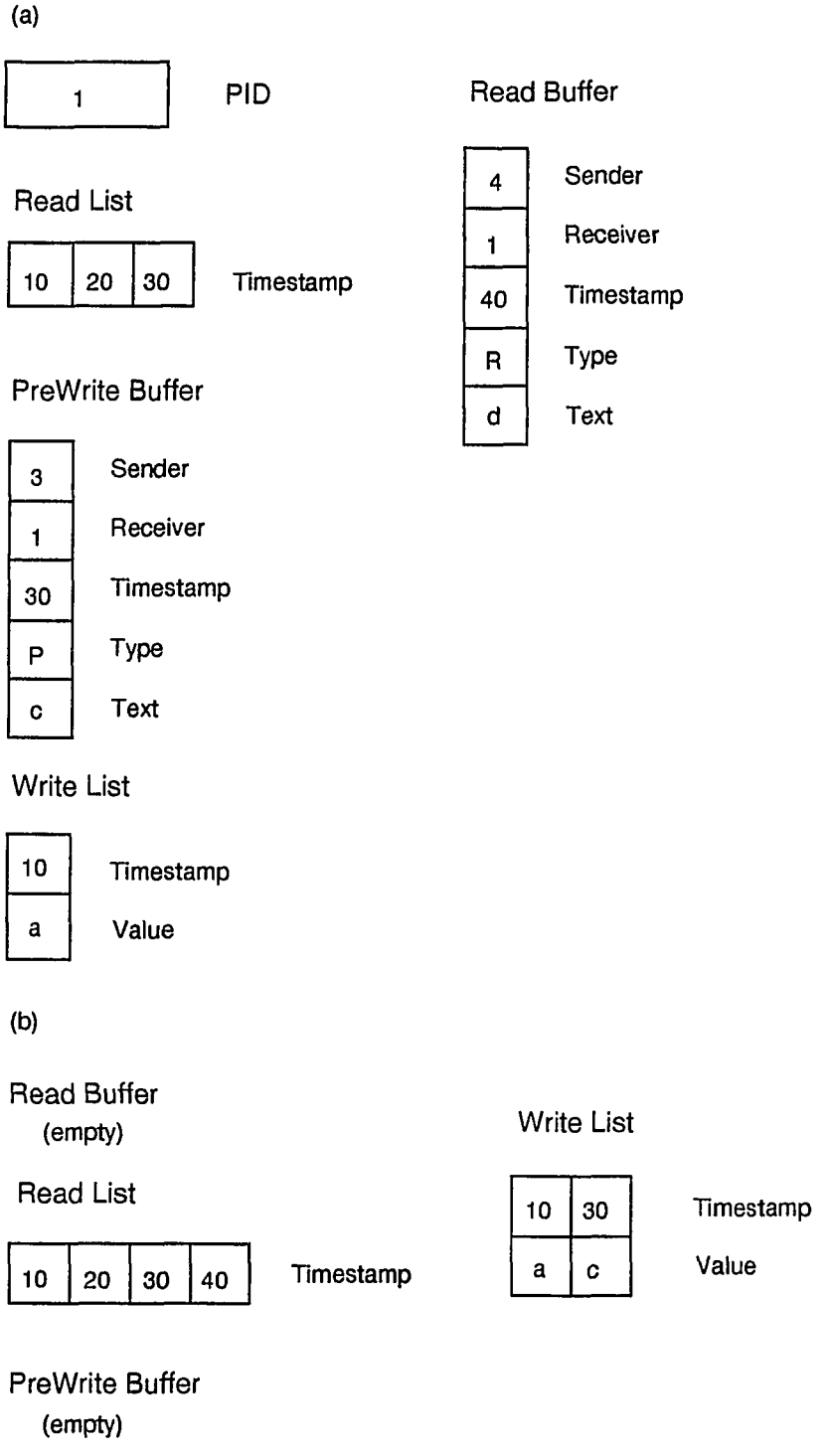


Figure 3.4: Examples of a MVTO database manager processing Write messages. (a) State prior to processing. (b) State after processing a Write message with timestamp 30.

End_While;

The transaction manager requires three phases to submit a transaction. They are a Read phase, a Prewrite phase, and a Write phase. The Read phase is similar to the one described earlier for virtual time. Read requests are sent. The manager blocks until a response is received. All Read requests will eventually get satisfied, but blocking is involved. The Prewrite and Write phase are necessary to support a two phase commit protocol [10, 8]. The manager first issues a Prewrite to a manager it needs to write a value to. If the Prewrite is accepted and no other part of the transaction is aborted, a Write will be issued also. Whenever a Prewrite is accepted, a Write is guaranteed to be accepted. As with the virtual time system, the loop runs until a fixed number of transactions are successfully completed. This condition indicates global termination in this research.

During the Prewrite phase, any database manager which receives a Prewrite may reject it forcing the entire transaction to abort. This is accomplished by the database manager sending an Abort message back to the transaction manager. Once the transaction manager receives an abort message from any database manager, it must cancel all the requests it made and restart the transaction with a new and larger timestamp.

The transaction manager waits for some fixed time period after issuing Prewrites. During this time period, any database managers that need to reject a received Prewrite, then must do so in this time frame. This time length is a critical performance parameter in MVTO system. It must be tuned for optimal algorithm performance.

3.3 Abortion

It is possible for a message to arrive out of timestamp order in a MVTO system since their is no clock synchronization. When an out of order message called a straggler arrives, the manager may need to abort a transaction to resolve time conflicts. When a transaction is aborted, it is given a new and larger timestamp by the transaction manager and is restarted.

It is hoped that the delay and new timestamp will allow the same transaction to complete successfully.

In this system, the database managers make the decision on whether or not a transaction must abort when it processes Prewrite messages. If a Prewrite message conflicts with some previously granted Read request, it must be rejected and the transaction aborted. If no conflict exists, the message is accepted and buffered. By accepting a Prewrite, the transaction manager is guaranteeing that the future Write will not cause any inconsistency in the database.

A database manager rejects a Prewrite message explicitly by sending an Abort message back to a transaction manager. Once a transaction manager receives one of these Abort messages, it must cancel all the requests involved in this transaction and restart it. This process could repeat if many conflicts are detected. This is a primary weakness of MVTO synchronization.

3.4 Global Control

In a MVTO system, a Write message is committed as soon as it is received by the database manager. The transaction manager knows this message is expected since it is preceded by a Prewrite message. The time between these two messages could be considered commitment delay. This delay is not the same as in a virtual time system though.

There are two lists that need to be periodically purged in an MVTO system. They are the Read list and the Write list. These lists would grow indefinitely if some maintenance was not performed on them. The Read and Prewrite buffers do not require attention since messages in them are eventually processed and added to the two lists. In this research, memory utilization was not a primary performance driver. Therefore, the algorithm written to conduct this maintenance may not be very useful in general. The algorithm runs periodically and removes all entries from the lists that are less than some fixed time quanta subtracted from the current clock time. The quanta values selected for this algorithm were not optimal since it was not necessary.

3.5 Costs

Memory, message traffic, and execution overhead are cost factors to be considered with MVTO. Memory overhead is required to support MVTO synchronization. This memory stores timestamp and write information only. These values are needed by the database manager to make abortion decisions. The Read and Write lists comprise the memory overheads. Message traffic is not directly controllable in this system. The only way to reduce it is by reducing the number of transactions that abort. Slight execution overhead is required to update and maintain these two lists. None of this overhead though, is as significant as the virtual time overhead.

System tuning is extremely important in this system also. The most crucial parameter is the timeout period. This value expresses how long the transaction manager must wait after issuing all its Prewrites. In this research, a minimum timeout period was established for every simulation run to achieve maximum performance.

Chapter 4

IMPLEMENTATION ENVIRONMENT

Several factors must be considered when implementing a distributed system. They include the hardware, software, network, algorithm modifications, models, and the run time environment. These are now discussed.

4.1 Hardware

The hardware platform used in this research was Sun Microsystems SPARC Stations 1. They run at a speed of roughly 12 mips. Each station has a memory capacity of 8 megabytes and a local disk of 100 megabytes. They share a network with other workstations, fileservers, and other servers.

4.2 Network

All Sparc Stations used in this research are connected into a local area network via ethernet. The network hardware is based on the IEEE 802.3 standard. This has a theoretical maximum transfer rate of 10 megabits per second.

4.3 Software

4.3.1 UNIX

Each station ran under control of the UNIX operating system. The version used was SunOS 4.0.3c. This is comparable to Berkely Unix BSD V4.3 with some additions including NFS.

4.3.2 ISIS

The ISIS V2.1 toolkit for distributed and fault-tolerant programming [12] was used to implement the MVTO and TW systems. ISIS was built on top of the UNIX operating system. Its purpose is to make it easy to write a program that is distributed, dynamically expandable, and fault-tolerant. Application programs can be written in C, C++, and some versions of LISP, Fortran, and Prolog. ISIS provides many powerful tools to the application programmer.

The run time architecture of ISIS consists of several programs started on each machine where ISIS facilities will be directly accessed. Only two of these programs were required in this implementation. They were *isis* and *protos*. These two are the minimum required to execute ISIS applications. They provide initialization, health monitoring, and communication capabilities.

The application software was written in C. The software was structured in an ISIS format. This format requires the application to be broken into a set of tasks. The program executes as a UNIX process. Within the process, these tasks execute non-preemptively under the control of ISIS. An ISIS task looks just like a C function and shares the same address space and global variables as all other tasks and functions in the process. The main difference between a function and task is that a task can be invoked by the system to respond to an ISIS event (such as message arrival).

Most of the powerful tools offered by ISIS were not taken advantage of in this research. ISIS was primarily used as a message passing facility. Features of ISIS incorporated include process groups, clients, and the bypass facility [12]. Message were passed through the ISIS CBCAST protocol. This provided delivery of messages in the order they were sent.

4.4 Algorithm Modifications

4.4.1 Virtual Time

A traditional implementation of virtual time was not required for this research. In a distributed database system, there is more structure than in a distributed simulation environment. By using this knowledge, a less generalized implementation was constructed.

In this implementation, the following areas diverted from tradition philosophy:

- GVT Computation
- Rollback Mechanism
- History Information

As stated earlier, the GVT algorithm used here was not typical. The performance of a TW system is closely bound to the performance of the GVT algorithm. The assumptions of the algorithm such as only transaction managers are involved and no transaction manager can rollback beyond the current transaction are not true in general. If a different transaction model was selected, a different algorithm would be needed.

Rollback was specialized at both the transaction and database managers. A transaction manager issues a set of read requests followed by a subset of write requests. Since the transaction manager issues only one transaction at a time, it could not be rolled back beyond that transaction. It was assumed a Read-response message that gets rolled back does not affect the other members of the read set, but does for the writes.

Rollback at the database manager was designed following an optimistic method based on semantics of abstract data types [61]. The idea of this approach is to not rollback a manager just because a late message arrives. Instead, the message was processed at the current virtual time if the operation involved did not violate computational correctness. An example of this is a late arriving Read message. In a distributed database system, read operations do not conflict with other reads or writes. On the other hand, write operations do conflict and this is not possible. Another optimization undertaken involved Read-response antimessages. In addition to cancelling the corresponding Read-response message, a new value is returned in the message. Thus, two messages are actually combined into one. This

was done since the new value is always available as a result of the recent write which caused roll back.

Once rollback started at either manager, it was allowed to finish uninterrupted. Normally, the rollback operation would be allowed to be interrupted in case a message farther back in time needed to processed. This diverged from normal rollback, but was easier to implement and oriented towards ISIS message delivery. The rollback mechanisms were designed with this consideration.

The history information stored for rollback includes all entries in the output queue and a portion of the input queue. One structure not found in this implementation is the state queue. This queue contains checkpoint information for a process. Typical entries include information which makes the execution deterministic such as random number generator seeds. In this implementation, the models incorporated did not require information of this type. Different models or assumptions may require this queue. The memory required for storage in this queue could be significant.

4.4.2 MultiVersion Timestamp Ordering

As mentioned earlier, periodic maintenance must be performed on the Read and Write lists. This was done to remove old entries which no longer affect synchronization. In this implementation, this was accomplished by periodically removing entries older than some fixed quanta subtracted from the current time. This quanta value was large enough to insure no synchronization conflict would result. These lists also stored the entire message described earlier and shown in Figure 3.1. This was not necessary, but made the implementation easier. This was not an optimal way to manager memory, but this was not an issue in this research.

4.5 Transaction and Database Models

Several assumptions were made about the nature of transactions and transaction submission. They have a definite impact on how the algorithms were written and on performance. No blind writes were allowed. A transaction could not issue a write operation to a database

item without first reading it. No user was modeled. Instead, the transaction manager generated the user requests randomly [52] and issued them to the database managers. Only read/write (r/w) synchronization was considered. The control variables for this model included transaction size, intertransaction delay, and update probability.

A simple database model was used. Each database manager was responsible for a specified number of items. Whenever a transaction requested read or write access to an item, it was delayed 50 milliseconds to simulate the physical i/o operation. The only control variable used was for the database size.

4.6 Run Time Environment

A dedicated operating system was not available for this research. In light of this fact, several steps were taken to reduce/eliminate any outside affect on results. No users were on the machines involved. No processes other than the application, ISIS, and UNIX were running. No significant activity transpired across the network.

For each test, the same machines were used for MVTO and TW. Also, the same test runs for MVTO and TW were collected back to back. This was done to keep the environment as similar as possible.

A command node which ran on a separate machine controlled execution. This node started all application nodes and controlled global initialization and startup. During the actual test run, it sat idle waiting for global termination. At that point, all the nodes sent their results to the command node. All nodes were stopped by the command node after results were received.

Chapter 5

RESULTS

Several test were designed to compare the relative performance of the MVTO and TW systems. They are as follows:

Transaction Size number of requests within a transaction

Database Size number of items managed by a database manager

Intertransaction Delay delay between two consecutive transactions

Nodes number of nodes running systems

Read Only transactions with no write operations

These tests are not the only ones that could have been selected. Others can be found in various literature including [41, 13, 2, 21, 51, 59, 48, 43]. These five were selected because they provided a solid base in which to compare the two systems. The main influences on these specific choices were [47, 1, 14, 15].

The measurements used in each test comparison are :

- Response Time
- Message Traffic
- Throughput

The primary performance measurement in this research was response time. Every result collection run was tuned to achieve the lowest possible response time for both systems. The response time presented was the average amount of wall clock time per node required for a transaction manager to successfully submit 50 transactions. The value 50 was selected after several runs showed this value produced repeatable results for both systems. Message traffic was determined by counting every message sent by a manager. Throughput is a common measurement, but with varying definitions. In this research, it is defined to be:

Nodes * 50 / Total Response Time of all nodes

This value includes any reprocessing time required for rollback or abortion operations. Results from this test are inversely related to the response time measurement.

Another measurement collected is CPU execution time. This is based on how much CPU time the transaction and database managers took to execute. This included ISIS and UNIX overhead. The results of these tests are discussed in the following sections.

5.1 Transaction Size

The transaction size test holds all parameters constant except the transaction size. This value varied from 5% to 10% of the distributed database size. The number of nodes for this test was 8. The intertransaction delay was 0 and the database size was 10. All read requests were updated with a 25% probability.

The response time for this test is shown in Figure 5.1(a). MVTO performed slightly better when the transaction size is less than 8.5%. After this point, TW had superior performance. This figure suggested that TW is a better choice for large transaction sizes while smaller ones should perform slightly better with MVTO.

The message traffic for this test is presented in Figure 5.1(b). The traffic for MVTO is less than that of TW for transaction size up to 6.25%. After this, TW clearly induced less traffic. Overall, it appears that TW induced the least amount of message traffic.

The throughput for this test is displayed in Figure 5.1(c). This graph is inversely related to the one presented in Figure 5.1(a) since it is based on response time. As the transaction size increases, the throughput for MVTO dropped off dramatically while the throughput

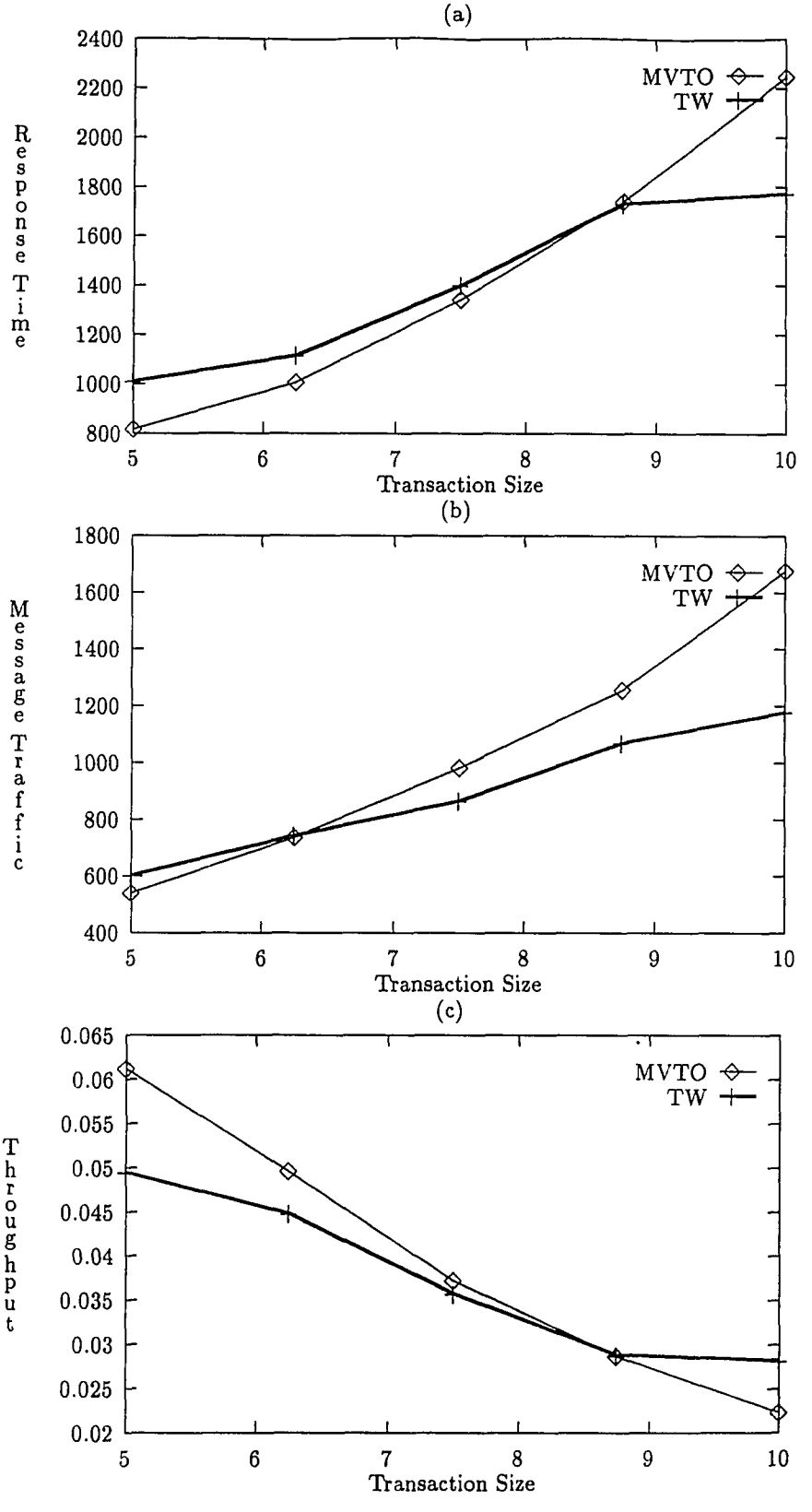


Figure 5.1: (a),(b),(c) Transaction Size Test Results

for TW flattened out.

5.2 Database Size

The database size test holds all parameters constant except the database size. This value varied from 1 to 20. The number of nodes for this test was 8. The intertransaction delay was 0 and the transaction size was 7.5%. All read requests were updated with a 25% probability.

The response time for this test is shown in Figure 5.2(a). Both methods performed very similar until the database size exceeds 12. At this point, TW responded faster. This figure suggests that TW is a good choice when the database size exceeds 12.

The message traffic for this test is presented in Figure 5.2(b). The traffic for MVTO is less than that of TW for database size less than 7. After this, the traffic for TW is less. Overall, it appears that TW generated the least amount of message traffic.

The throughput for this test is displayed in Figure 5.2(c). This graph is inversely related to the one presented in Figure 5.2(a) since it is based on response time. In this test, the values plotted are very close and no distinct advantage can be given to either.

5.3 Intertransaction Delay

The intertransaction delay test holds all parameters constant except the intertransaction delay. This value varied from 0 seconds to 4 seconds. The number of nodes for this test was 8. The database size was 10 and the transaction size was 7.5%. All read requests were updated with a 25% probability.

The response time for this test is shown in Figure 5.3(a). MVTO clearly out performed TW in all cases. This is not a surprising result since the nature of this test, additional delay, is against virtual time philosophy. This graph showed that a TW system suffers drastically by introducing artificial delay. The effects of this delay in the MVTO system was negligible.

The message traffic for this test is presented in Figure 5.3(b). TW always created less traffic than MVTO. This is of little consequence though in light of the primary measurement.

The throughput for this test is displayed in Figure 5.3(c). This graph is inversely related

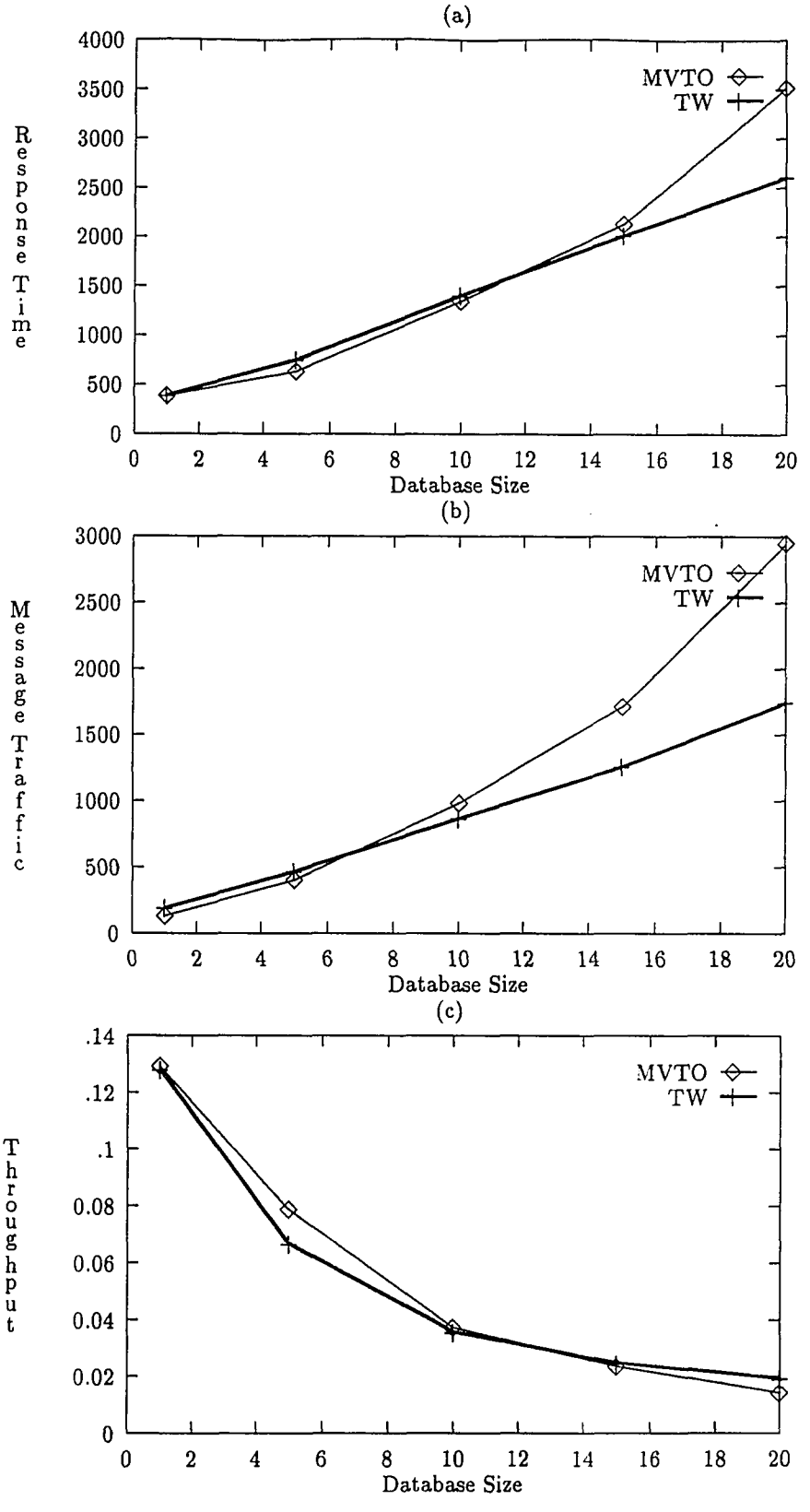


Figure 5.2: (a),(b),(c) Database Size Test Results

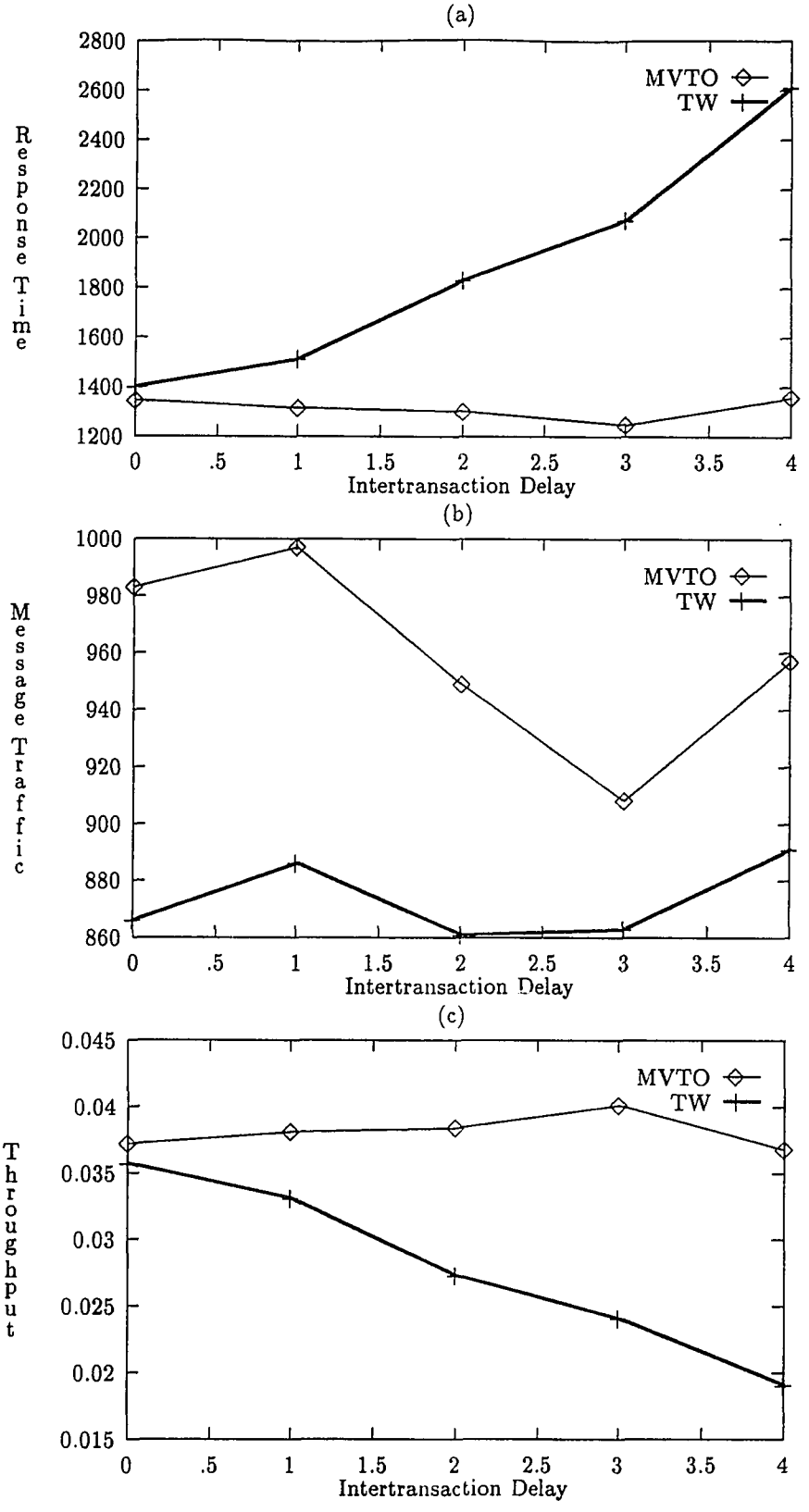


Figure 5.3: (a),(b),(c) Intertransaction Delay Test Results

to the one presented in Figure 5.3(a) since it is based on response time. In this test, MVTO throughput remained relatively constant while TW throughput dropped off drastically.

5.4 Nodes

The nodes test was not designed as one might think. Nodes were not added in an attempt to reduce response time. Instead, by adding a node, more transactions as well as databases are introduced into the distributed system. The nodes test holds all parameters constant except the number of nodes involved. This value varied from 4 nodes to 12 nodes. The transaction size for this test was 8 requests. The intertransaction delay was 0 and the database size was 10. All read requests were updated with a 25% probability.

The response time for this test is shown in Figure 5.4(a). From the graph, it is evident TW responded faster in all cases. It is interesting to note the shapes of both curves. The same basic shapes indicated that the parameters for each run had to the same relative effect on performance.

The message traffic for this test is presented in Figure 5.4(b). The traffic for TW is significantly less than that of MVTO in all cases. This test had the greatest traffic difference than the others when comparing the two systems.

The throughput for this test is displayed in Figure 5.4(c). This graph is inversely related to the one presented in Figure 5.4(a) since it is based on response time. This graph shows TW is superior in all cases. This is expected because of the response time results.

5.5 Read Only

A special one run test was conducted for read only transactions. In this test, 8 nodes were used. The database size was 10. The transaction size was 7.5% and the intertransaction delay was 0. The update probability was 0%.

MVTO was able to execute all the transaction with a response time of 600 seconds. The TW value was 704 seconds. MVTO performed much better in this test. MVTO required 600 messages while TW required 651. The throughput for MVTO was .083 and .071 for

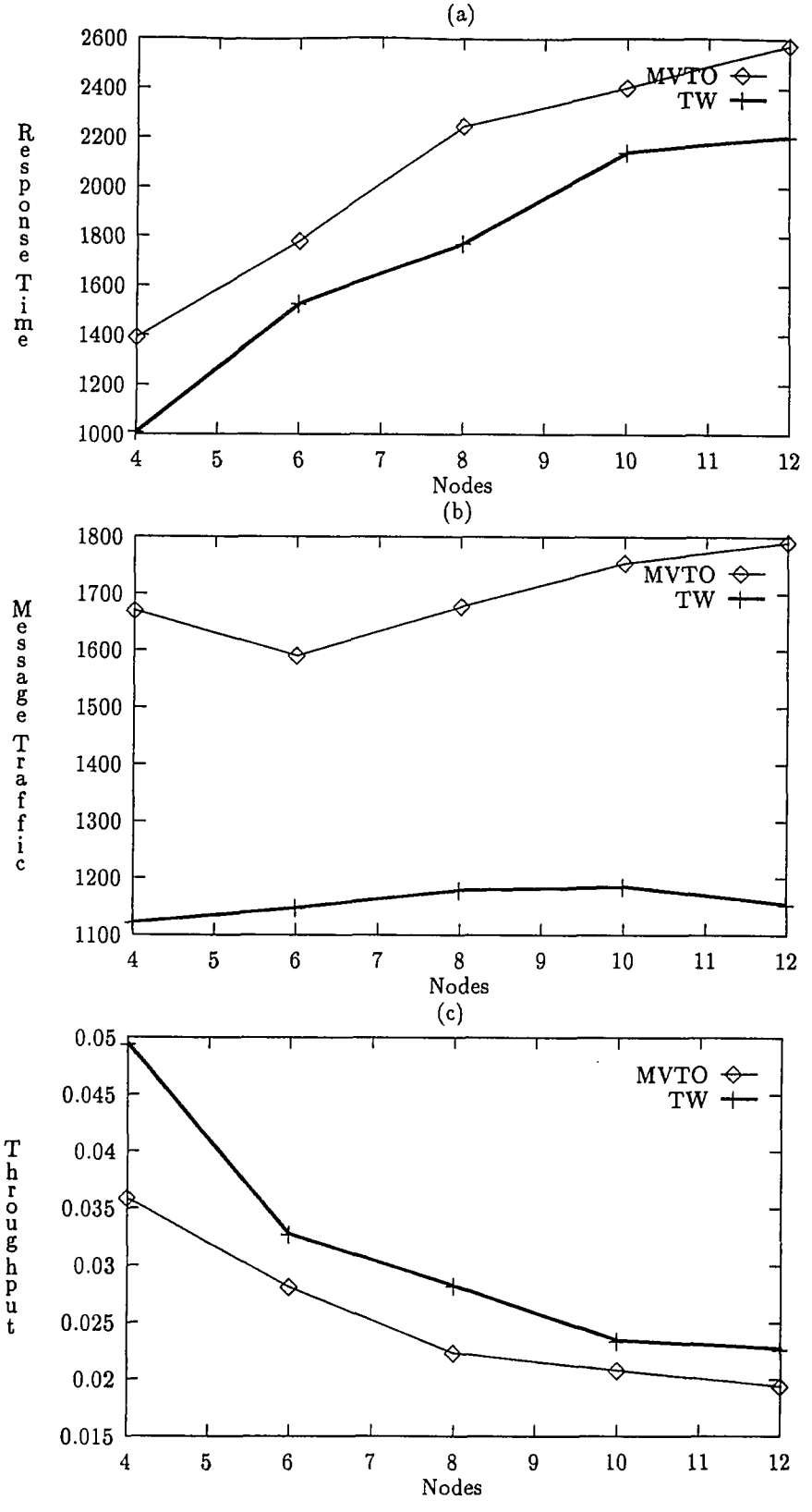


Figure 5.4: (a),(b),(c) Nodes Test Results

TW. From these results, it appeared that the overhead for TW is to much in a read-only transaction environment.

5.6 Other Measures

The CPU time required for all the tests favored MVTO significantly. On the average, The CPU for MVTO was busy 40% of the time while TW was busy 82%. This result was likely influenced by the implementation and environment. It is expected that TW would require more time than MVTO in any setting, but of a lower magnitude. This result is not that surprising since TW attempts to keep the CPU busy by processing (and reprocessing) messages while MVTO blocks frequently.

Chapter 6

CONCLUSION

This research reported on the first implementation and results of a DDBMS synchronized by virtual time. Additionally, a MVTO algorithm was implemented to compare these results. Message traffic, response time, and throughput were used for the relative comparison. Performance tests varied the database size, transaction size, intertransaction delay, number of nodes, and update probability. Design and implementation issues were discussed along with advantages and disadvantages.

Some broad conclusion can be drawn from the results presented earlier.

- Virtual time nominally outperformed MVTO as the transaction size increased.
- Virtual time performed similar to MVTO when the database size was varied.
- Adding artificial delay to a virtual time system drastically reduced performance and MVTO always outperformed it.
- Virtual time always outperformed MVTO as the number of nodes (as well as transaction and databases) increased.
- Virtual time requires substantially more memory for support of the rollback mechanism than the multiversion history information in MVTO.

For the read only test, it appeared that the virtual time overhead was too large as MVTO performed better. The message traffic results generally favored the virtual time approach

for a majority of the tests. CPU utilization was significantly higher with virtual time.

We argue that virtual time is a viable alternate concurrency control method for distributed database systems if the memory overhead can be absorbed. We believe that virtual time synchronization is at least capable of performance similar to MVTO. In general, our results showed that as conflict increased, virtual time outperformed MVTO. As in all performance studies, the results presented and conclusions drawn here are bound to the database model and transaction model assumptions. Additionally, the implementation and environment need to be taken into account.

There are several possibilities for future research in this area. The memory overhead for virtual time needs to be examined in detail in a distributed database environment. Clever compression algorithms for reducing memory overheads would be very beneficial. The performance of virtual time with different database and transaction models is desirable. Alternate tests for comparison could provide some interesting results. Other algorithms such as two-phase locking or an optimistic verifier would be valuable. Another environment more suited for distributed databases should be considered. An object oriented platform should be considered since virtual time can logically follow this and since object oriented databases are becoming more popular and accepted. A final topic not covered in this research in reliability. The ability to detect and recover from sites failure is an important issue in distributed database systems and needs to be explored.

Bibliography

[1] AGRAWAL, R., AND CAREY, M. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Transactions on Database Systems 12*, 4 (December 1987), 609–654.

[2] AGRAWAL, R., AND SENGUPTA, S. Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control. *ACM Special Interest Group Management of Data 18*, 2 (1989), 408–417.

[3] AGRE, J. Simulation of time warp distributed simulations. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 85–90.

[4] BAEZNER, D., CLEARY, J., LOMOW, G., AND UNGER, B. Algorithmic optimizations of simulations on time warp. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 73–78.

[5] BECKMAN, B., DILORETO, M., STURDEVANT, K., HONTALAS, P., WARREN, L., BLUME, L., JEFFERSON, D., AND BELLENOT, S. Distributed simulation and Time Warp Part 1: Design of Colliding Pucks. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 56–60.

[6] BELLENOT, S., AND DILORETO, M. Tools for measuring the performance and diagnosing the behavior of distributed simulations using time warp. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 145–150.

[7] BERNSTEIN, P., AND GOODMAN, N. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems. In *IEEE International Conference on Data Engineering* (1980), pp. 285–299.

[8] BERNSTEIN, P., AND GOODMAN, N. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys 13*, 2 (June 1981), 185–221.

[9] BERNSTEIN, P., AND GOODMAN, N. Multiversion Concurrency Control-Theory and Algorithms. *ACM Transactions on Database Systems 8*, 4 (December 1983), 465–483.

[10] BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[11] BERRY, O. *Performance Evaluation of the Time Warp Distributed Simulation Mechanism.* PhD thesis, University of Southern California, May 1986.

[12] BIRMAN, K., COOPER, R., JOSEPH, T., MARZULLO, K., MAKPANGOU, M., KANE, K., SCHMUCK, F., AND WOOD, M. *The ISIS System Manual, Version 2.1.* The ISIS Project, September 1990.

[13] CAREY, M., AND LIVNY, M. Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases* (1988), pp. 13–25.

[14] CAREY, M., AND MUHANNA, W. The Performance of Multiversion Concurrency Control Algorithms. *ACM Transactions on Computer Systems 4*, 4 (November 1986), 338–378.

[15] CAREY, M., AND STONEBRAKER, M. The Performance Concurrency Control Algorithms for Database Management Systems. In *Proceedings of the Tenth International Conference on Very Large Data Bases* (August 1984), pp. 107–118.

[16] CERI, S., AND PELAGATTI, G. *Distributed Databases Principles and Systems.* McGraw-Hill, 1984.

[17] CHANDY, K., AND LAMPORT, L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems 3*, 1 (February 1985), 63–75.

[18] CHANDY, K., AND MISRA, J. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering SE-5*, 5 (September 1979), 440–452.

[19] CHANDY, K., AND MISRA, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM 24*, 11 (April 1981), 198–206.

[20] CLEARY, J., UNGER, B., AND LI, X. A distributed and-parallel backtracking algorithm using virtual time. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1989), 177–182.

[21] DAN, A., TOWSLEY, D., AND KOHLER, W. Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control Protocols. In *IEEE International Conference on Data Engineering* (1988), pp. 418–425.

[22] EBLING, M., DILORETO, M., PRESLEY, M., WIELAND, F., AND JEFFERSON, D. An ant foraging model implemented on the time warp operating system. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 21–26.

[23] FUJIMOTO, R. Time Warp on a Shared Memory Multiprocessor. Tech. Rep. UUCS-88-021a, Computer Science Department, Univ. of Utah, January 1989.

[24] FUJIMOTO, R. The Virtual Time Machine. Tech. Rep. UUCS-88-019, Computer Science Department, Univ. of Utah, January 1989.

[25] FUJIMOTO, R. Parallel Discrete Event Simulation. *Communications of the ACM 33*, 10 (October 1990), 30–53.

[26] FUJIMOTO, R., TSAI, J., AND GOPALAKRISHNAN, G. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. Tech. Rep. UUCS-88-011, Computer Science Department, Univ. of Utah, July 1988.

[27] FUJIMOTO, R., TSAI, J., AND GOPALAKRISHNAN, G. The roll back chip: Hardware support for distributed simulation using Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 81–86.

[28] GAFNI, A. *Space Management and Cancellation Mechanisms for Time Warp.* PhD thesis, University of Southern California, December 1985. TR-85-341.

[29] GAFNI, A. Rollback mechanisms for optimistic distributed simulation systems. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 61–67.

[30] GATES, B., AND MARTI, J. An empirical study of time warp request mechanisms. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 73–80.

[31] HONTALAS, P., BECKMAN, B., DILORETO, M., BLUME, L., REIHER, P., STURDEVANT, K., WARREN, L., WEDEL, J., WIELAND, F., AND JEFFERSON, D. Performance of the Colliding Pucks simulation on the time warp operating system (Part 1: Asynchronous behavior and sectoring). *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 3–7.

[32] J.B. GILMER, J. An assessment of Time Warp parallel discrete event simulation algorithm performance. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 45–49.

[33] JEFFERSON, D. Virtual Time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.

[34] JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., DILORETO, M., HONTALAS, P., LAROCHE, P., STURDEVANT, K., TOPMAN, J., WARREN, L., WEDEL, J., YOUNGER, H., AND BELLENOT, S. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the Eleventh Annual ACM Symposium on Operating System Principles* (November 1987), pp. 77–93.

[35] JEFFERSON, D., AND MOTRO, A. The Time Warp Mechanism for Database Concurrency Control. In *International Conference on Data Engineering* (February 1986), pp. 474–481. (Cat. No. 86CH2261-6).

[36] JEFFERSON, D., AND SOWIZRAL, H. Fast Concurrent Simulation Using the Time Warp Mechanism, Part i: Local Control. Tech. Rep. Rand Note N-1906-AF, The Rand Corporation, Santa Monica, CA, December 1982.

[37] KOHLER, W. A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys 13*, 2 (June 1981), 149–183.

[38] KUNG, H., AND ROBINSON, J. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

[39] LEE, R., AND L.LILEN. Optimistic Algorithms in Distributed Systems. In *Second International Conference on Computers and Applications* (June 1987), pp. 699–705. (Cat. No. 87CH2433-1).

[40] LEMASTER, T., DATTA, A., AND GHOSH, S. Virtual Time-A New Paradigm for Synchronization in Distributed Systems. *Advances in Distributed and Parallel Computing 1* (1992). (in press at ABLEX publishing).

[41] LI, V. Performance Models of Timestamp-Ordering Concurrency Control Algorithms in Distributed Databases. *IEEE Transactions on Computers C-36*, 9 (September 1987), 1041–1051.

[42] LI, X., UNGER, B., CLEARY, J., LOMOW, G., AND WEST, D. Communicating sequential PROLOG. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 166–170.

[43] LIN, W., AND NOLTE, J. Basic Timestamp, Multiple Version Timestamp, and Two-Phase Locking. In *Proceedings of the Ninth International Conference on Very Large Data Bases* (1983), pp. 109–118.

[44] LIN, Y.-B., AND LAZOWSKA, E. Determining the Global Virtual Time in a Distributed Simulation. Tech. Rep. Tech. Rep. 90-01-02, Department of Computer Science, University of Washington, December 1989.

[45] LOMOW, G., CLEARY, J., UNGER, B., AND WEST, D. A performance study of Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 50–55.

[46] MISRA, J. Distributed discrete-event simulation. *ACM Computing Surveys 18*, 1 (March 1986), 39–65.

[47] NOE, J., AND WAGNER, D. Measured Performance of Time Interval Concurrency Control Techniques. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases* (1987), pp. 359–368.

[48] ORJI, C., LILIEN, L., AND HYSIAK, H. A Performance Analysis of an Optimistic and a Basic Timestamp-Ordering Concurrency Control Algorithms for Centralized Database Systems. In *IEEE International Conference on Data Engineering* (1988), pp. 64–71.

[49] OZSU, M., AND VALDURIEZ, P. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[50] PAPADIMITRIOU, C. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[51] PAPADIMITRIOU, C., AND KANELLAKIS, P. On Concurrency Control by Multiple Versions. *ACM Transactions on Database Systems 9*, 1 (March 1984), 89–99.

[52] PARK, S., AND MILLER, K. Random Number Generators: Good Ones are Hard to Find. *Communications of the ACM 31*, 10 (October 1988), 1192–1201.

[53] PEACOCK, J., WONG, J., AND MANNING, E. G. Distributed Simulation Using a Network of Processors. *Computer Networks 3* (1979), 44–56.

[54] PRESLEY, M., EBLING, M., WIELAND, F., AND JEFFERSON, D. Benchmarking the time warp operating system with a computer network simulation. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 8–13.

[55] PUCCIO, J. A casual discipline for value return under Time Warp. *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 171–175.

[56] REED, D. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, M.I.T., September 1978. Tech. Report TR-205.

[57] REED, D. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems 1*, 1 (February 1983), 3–23.

[58] SAMADI, B. *Distributed simulation: Performance and Analysis.* PhD thesis, Department of Computer Science, University of California at Los Angeles, January 1985.

[59] THOMAS, R. A Majority Consensus Approach to Concurrency Control for Mulitple Copy Databases. *ACM Transactions on Database Systems 4*, 2 (June 1979), 180–209.

[60] TINKER, P., AND AGRE, J. Object creation, messaging, and state manipulation in an object oriented time warp system. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 79–84.

[61] WEST, D., LOMOW, G., AND UNGER, B. Optimising time warp using the semantics of abstract data types. In *Proceedings of the Conference on Simulation and AI* (January 1987), pp. 3–8.

[62] WIELAND, F., HAWLEY, L., FEINBERG, A., DILORETO, M., BLUME, L., REIHER, P., BECKMAN, B., HONTALAS, P., BELLENOT, S., AND JEFFERSON, D. Distributed combat simulation and time warp: The model and its performance. *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 14–20.

[63] WITKOWSKI, A. *Performance Evaluation of Timestamp Driven Databases.* PhD thesis, Department of Computer Science, University of Southern California, September 1985.