Performance Evaluation of a Prudent Two-Phase Commit Protocol

Boutros S. Boutros

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

July 1994

ISBN 0-315-97583-0

Canada

Abstract

Performance Evaluation of a Prudent Two-Phase Commit Protocol

Boutros S. Boutros

A distributed database system is subject to data inconsistency in the presence of failures. Coordinating the execution of the subtransactions of a global transaction, and committing or aborting transactions, are used to maintain data consistency in case of failures. Two classes of atomic commitment protocols have been proposed in the literature. The first class, fault tolerant, aborts a transaction without checking for alternative solutions. The second, optimistic, ignores the occurrence of errors and assumes that a global transaction commits in most cases; hence, it leads to compensating transactions to preserve data consistency, in case the global transaction aborts.

In this thesis, we study the performance of a Prudent Two-Phase Commit protocol, which in the presence of failures does not abort a transaction carelessly. The prudent protocol uses a second chance for a transaction before it decides to abort it. This careful approach prevents a transaction from aborting in case of transient communication failures. In this way, it improves system performance and reliability.

For the purpose of this study, we simulate a distributed database system. The performance of the simulated distributed system is measured while using the Prudent, Basic, and Optimistic Two-Phase Commit protocols. The performances of these protocols are compared and evaluated according to a number of widely accepted performance metrics. The final performance results are presented with a discussion and interpretation of the graphs generated by the simulation. The result of these simulations confirms the improvement in system performance with the Prudent Two-Phase Commit Protocol.

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Dr. Bipin C. Desai, for the support and encouragement he has given me during the course of my studies. I am very grateful for the advice he has offered, and for the patience and understanding he has shown towards me.

I thank the Computer Science staff for the help they offered during the implementation of my programs. Special thanks to Mr. Stan Swiercz for the considerable time and effort he spent to install and maintain the Modula-3 compiler.

I thank all my friends in the Department of Computer Science. Special thanks to Shiri Nematollah for his advice and help.

Finally, many thanks to my parents for their caring emotional support.

Epigram

God hath constituted every other animal, one to be eaten, another to serve for tilling the land, another to yield cheese, another to some kindred use; for which things what need is there of the observing and studying of appearances, and the ability to make distinctions in them? But man He hath brought in to be a spectator of God and of His works, and not a spectator alone, but an interpreter of them.

Epictetus (50 AD)
Dissertations, i.,6.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

1.1. Overview

Currently, **database systems** are among the most important and widely used computer applications. The increased reliance on database systems by many applications and users requires databases to deal with a variety of demands, involving large amounts of data. Such applications could be banking, airplane reservation or many other on-line information systems. Consequently, a **database management system** (DBMS) should provide in addition to high **performance**, high **availability**, **reliability** and **maintainability**.

A database is a collection of objects satisfying certain **integrity** constraints, which may vary from one system to another. As an example, in a banking system no user account may have a negative value, or in a reservation system the number of confirmed reservations on a flight should not exceed the number of available seats on it. The integrity constraints must be respected by the user program and enforced by the DBMS. The user program is responsible for testing the values specific to its application before performing updates on objects. The DBMS is responsible for the correctness of updates on objects. Hence, updates running **concurrently** should not **interfere** with each other.

A **transaction** is a sequence of **read** and **write** actions issued by a user program on database objects. A transaction possesses certain properties to ensure the correctness of a DBMS: **atomicity**, **consistency**, **isolation** and **durability**. A transaction **commits** after it performs all of its actions. Hence, a **commit** will make the effects of the transaction permanent.

The data objects of a DBMS may be distributed among different locations which are geographically dispersed. This distribution is required in organizations with different operating units, each having a local database which is shared with other sites. Such DBMS are called **distributed database management systems**(DDBMS). DDBMS provide high availability and reliability by the distribution of data among many **sites**. Each site has an independent DBMS. These sites are connected by a communication network. A transaction in a DDBMS may require the use of data not available on its local site; such

a transaction is called a **global transaction**. A global transaction splits into a number of **subtransactions**, one for each site containing the objects of its read and write sets.

1.2. THE PROBLEM

The global transaction execution in a DDBMS creates a special problem: the **atomic commit** property of a global transaction, running a number of subtransactions, must be preserved; either all its subtransactions commit or none of them. Thus there is the need for a distributed **atomic commitment protocol** (ACP) that ensures the atomic commitment of a global transaction [BERN87]. Many ACP protocols have been proposed for DDBMS; the most simple and commonly used one is the **Two-Phase Commit** protocol (2PC) [BERN87], which we will refer to as **Basic Two-Phase Commit** (B2PC). B2PC ensures the atomicity of a global transaction. In the B2PC protocol there is a **coordinator** site and one or more **participant** sites. A coordinator site, usually the site where the global transaction originates, is responsible for orchestrating the sub-transactions. Its responsibility is to coordinate the atomic commitment, by receiving the **votes** of participants and sending the final **decision** after all votes are received. Each site participating in the execution of the global transaction is a **participant**. The responsibility of participants is to send their votes after they are ready to **commit** or **abort**. In the former case, the coordinator's decision is required before the actual commit. Participants communicate with their coordinator by **messages** sent through the communication network.

The **Optimistic Two-Phase Commit** protocol (O2PC) [LEVY91] is an ACP that works under the **optimistic** assumption of rare failures. In O2PC, participants which are ready to commit send their commit votes, and instead of waiting for the coordinator's decision go ahead and commit. This early commitment will violate the atomic commitment rule in case one of the other subtransactions aborts. For such a case, the O2PC uses the notion of **compensation**. A **compensating transaction** will undo the effects of a committed subtransaction semantically without resorting to cascading aborts.

A DDBMS is subject to **site** and **communication failures** which might violate the transaction atomicity rule. B2PC is designed to tolerate failures; the solution to a failure in B2PC is to **abort** the global transaction and all its subtransactions. Unfortunately, B2PC treats a **transient communication failure** as a site failure, leading to aborting the

transaction even in cases where the failure is transient and the abort could be avoided. Such aborts affect the system throughput and fault tolerance.

1.3. THE GOAL OF THIS THESIS

The cost of aborting a transaction is high, especially in systems where a delay in transaction execution may affect system throughput and response time. Hence, it is desirable to avoid aborting transactions whenever possible. A **Prudent-Two Phase Commit** protocol (P2PC) that is careful about aborting a transaction was first proposed in [DESA]. P2PC will give a coordinator or participant another chance before deciding to abort a transaction in case of a failure. This second chance could save the transaction from an abort when a transient communication failure occurs.

In this study we implement P2PC, simulate it and compare its performance with B2PC and O2PC. The results of this simulation study demonstrate the performance of these protocols according to certain performance metrics used widely in the database literature.

1.4. THE THESIS OUTLINE

In chapter 2 we describe the different components of a distributed DBMS. Chapter 3 reviews the different commit protocols in the literature. In chapter 4 we present the algorithms for P2PC, O2PC and B2PC. Chapter 5 presents the simulation model and the distributed database system which is used to compare the performance of these protocols. Chapter 6 presents the results of our study and compares their relative performance. Finally, in chapter 7 we give our conclusions and suggestions for future work.

CHAPTER 2

BACKGROUND

"Distributed database system (DDBMS) technology is the union of what appear to be two diametrically opposed approaches to data processing: *database system* and *computer network* technologies." [ÖZSU91]. Özsu thus defines the components of a distributed database system to be the union of two disparate technologies, and points out that these two contrastive approaches can be synthesized to produce a technology that is more powerful and more promising than either one alone. In this chapter we briefly describe the database system, the database transactions, and the computer network which are the main elements of a DDBMS.

2.1. DATABASE SYSTEM COMPONENTS

A database system consists mainly of four modules [BERN87]:

I. A **Transaction Manager** (TM), which performs the necessary pre-processing of a transaction before it starts.
II. A **Scheduler**, which controls the relative order of transaction executions. A Scheduler can reject, delay or forward a transaction.
III. A **Recovery Manager**, which is responsible for ensuring that the database contains all of the effects of committed transactions and none of the effects of aborted ones.
IV. A **Data Manager** (DM), which is responsible for operating directly on database objects.

2.2. THE DATABASE TRANSACTION

A transaction in a database system is an execution of a program that accesses a shared database [BERN87]. The goal of concurrency control is to ensure the atomicity of transactions that are executing in parallel. A transaction may be viewed as a sequence of read or write actions. Two transactions executing in parallel may conflict if they want to access the same object at the same time, and at least one of the actions upon that object is a write operation.

A database is a set of objects each having a value. The set of values of the objects at any time constitute the **state** of the database. A transaction changes the state of the database from a **consistent** state to another **consistent** state. The effects of a transaction are made **permanent** once the transaction commits. A transaction that has issued its start but not committed is called an **active transaction.** A transaction that aborts is a transaction that did not complete its execution successfully. An aborting transaction does not change the state of the database, since none of its effects are made permanent. Hence, it is invisible to the users of a database system.

When a transaction starts it is forwarded first to the Transaction Manager, which does the necessary pre-processing before it is forwarded to the Scheduler. For example, in a distributed database system the preprocessing of a global transaction would be to determine which sites are involved in its execution and how to divide it into a number of subtransactions.

Before executing an operation, the transaction passes that operation to the Scheduler. When the Scheduler receives an operation, it can perform one of the following actions [BERN87]:

I. **Execute**: The Scheduler can pass the operation to the DM. Once the operation is finished the DM will inform the Scheduler.
II. **Delay**: The Scheduler can delay the operation by putting it on its internal queue. This delay can happen in case the transaction is blocked due to a lock conflict (*i.e.* another active transaction has reserved the right to write to some objects involved in this transaction).
III. **Reject**: The Scheduler can reject the operation. Such a decision will force the transaction to abort.

Using these actions, the Scheduler can control the order in which the operations are executed. Hence, the Scheduler controls the concurrent executions of transactions, and can play a role in deadlock detection and prevention.

2.3. Concurrency Control Correctness

The execution of a number of transactions concurrently leads to the **interleaving** of the operations of these transactions. Interleaving could lead to an inconsistent database; this

could be prevented by **Serial execution**, in which one transaction performs all of its actions before the second starts. Serial executions are correct, because a transaction is internally consistent. The execution of a number of transactions is said to be **serializable** if the result of their execution is the same as if the transactions were executed in some serial order. **Serializability** is one of the main tools for proving the correctness of Concurrency Control algorithms. All serializable executions are equally correct; therefore the DBMS might execute the DB operations in any order, as long as they are serializable [BERN87].

Locking is a mechanism used to ensure serializability in a Database System; it is commonly used to solve the problem of synchronizing access to shared resources. Each data item has a **lock** associated with it. A transaction requests a lock for the item it wants to access. If the item is already locked by some other transaction in an incompatible mode, the requesting transaction will be **blocked** until the lock is released. Otherwise, the lock is granted and the transaction can perform its operations on the item. There are at least two kinds of locks, a **read lock** and a **write lock**. A read lock is a **shared** lock, *i.e.*, many transactions can read the same item at the same time. On the other hand, a write lock is **exclusive**, *i.e.*, only one transaction can write to an item at a time. Therefore, conflicts exist on the same object between read and write, write and write.

Two-Phase Locking (2PL) is a protocol widely used to synchronize locking in a DBMS, due to its simplicity and ease of implementation; a brief description of 2PL is given by the following three rules [BERN87]:

I. When the 2PL Scheduler receives an operation to lock a database object, first it checks whether it is already locked by another active transaction. If so, the 2PL Scheduler delays the transaction requesting the lock until all objects involved are released. Otherwise, the 2PL Scheduler grants the lock to the requesting transaction, and locks the objects.
II. Once the 2PL Scheduler sets a lock for a database object, it may not release that lock until it receives a signal stating that the operations on the object have been accomplished.
III. Once the Scheduler has released a lock for a transaction it may not set any more locks for that transaction.

The third rule is the main source of the name "Two-Phase". The first phase is the **growing** phase, where the transaction requests locks. The second phase is the **shrinking** phase, where the transaction releases its locks. Once a transaction releases a lock it cannot obtain any more locks.

2.4. DISTRIBUTED DATABASE MANAGEMENT SYSTEMS (DDBMS)

A centralized database system has one shared database under its control. A number of terminals can access this database at the same time. If the system fails in a centralized DBMS no access can be done to the database until the system recovers, and the system throughput is limited to the maximum number of users that the local CPU's can handle. Therefore, the availability reliability and performance of a centralized DBMS are relatively lower than a DDBMS. The notion of distributed database systems comes from **distributivity**, *i.e.*, distributing the database over a number of computer systems.

Distributed databases constitute a single relatively large database, distributed over many sites connected by means of a communication network. A distributed database offers many advantages; the first is the **increased availability** of data. If one site fails, data can still be obtainable from other sites. The second advantage is **load distribution;** the data is distributed on the network. If the number of sites is n, and if the level of concurrency at each site is equal and is p, then the total throughput of the system is $n*p$. Compare this to a throughput of p on a centralized DBMS. These advantages are the goals of an ideal DDBMS, but unfortunately there is a price to pay for these additional features. The price includes the high cost of DDBMS, the additional difficulty of distributed concurrency control, the management of replicated data and the communication costs between sites.

A DDBMS can be **strictly partitioned, partially redundant** or **replicated**, or **fully redundant** [DESA91]. A **strictly partitioned** database contains no replicated data among different sites. In a **partially redundant** DDBMS only certain fragments of the database are redundant. In **fully redundant** DDBMS, complete database copies are distributed among all sites. The main reason for redundant databases is to increase the database availability by storing the critical data at multiple sites; in this way, the DBMS can continue to operate even though one of its sites has failed. Furthermore, the performance is improved since there are many copies of a data item. The data may be read from the closest site, thereby reducing communication costs.

The advantages of a DDBMS can be summarized as being: **high data availability** and **reliability, increased performance** and **throughput**. On the other hand, a DDBMS involves increased overhead for controlling global transactions, and the associated problems that are to be solved in order to achieve the above advantages.

2.4.1. ATOMIC COMMITMENT

In a distributed system a transaction's execution may involve many sites, *i.e.*, a transaction T may split into n subtransactions at n sites. To satisfy the **atomicity** property of a transaction, either all these subtransactions **commit** or none of them. Consider a distributed transaction T whose execution involves n sites. Suppose that site s_1 supervises the execution of the transaction. Before s_1 sends commit decisions to the other sites it must make sure that they are able and ready to commit. Assuming that each site has its own process which controls the subtransaction at the site, these processes at each site are responsible for ensuring the atomic commitment protocol for T. The process at T's home site, s_1, is called T's **coordinator**. The remaining sites are T's **participants**. The **coordinator** site is aware of the participant sites, so that it can communicate with them. The **participants** know who their coordinator is but may not necessarily know the other participants. An **Atomic Commitment Protocol** (ACP) [BERN87] is an algorithm meant to synchronize the commitment of the distributed transactions such that either **all** the sites of that transaction **commit** or they **all abort**. Each process may vote a **Yes** or a **No**, and can reach exactly **one** of **two decisions**: either **commit** or **abort**.

An ACP protocol is an algorithm for a number of co-operating processes to reach a decision such that [BERN87]:

1. All processes that reach a final decision reach the same one.
2. A process cannot reverse its interim decision after it has reached one.
3. The commit decision can only be reached if all processes have reached an interim decision to commit.
4. If there are no failures and all processes voted to commit, then the decision will be to commit.
5. Consider any execution containing only failures that the algorithm is designed to tolerate. At any point in its execution, if all existing failures are repaired rapidly and no new failures occur for a sufficiently long period of time, then all processes will eventually reach a decision.

Rule 1 says that all processes should reach the same final decision, either a commit or an abort. **Rule 2** forbids a process to change its decision after it has reached one. Hence once a process sends a Yes or No vote, it cannot reverse this vote later. If a process is allowed to change its vote, then the coordinator's decision according to the first vote would no longer be consistent. This might end up with processes not reaching the same decision, hence violating the atomic commitment rule.

Rule 3 implies that even if a single process votes No, the commit decision will not be reached. A consequence of rule 3 is that a process can decide to abort at any time before it has voted to commit. On the other, hand after voting Yes, a process cannot change its decision to an abort. The period between the processes voting Yes and receiving the coordinator's final decision is called the **uncertainty period**. During this period, a process is not certain regarding the final decision, and cannot decide to **unilaterally commit** or **abort** before the coordinator's decision.

Rule 4 states that in the absence of failures, with all processes reaching a commit decision, then the decision will be to commit. This rule excludes the existence of trivial and useless protocols which most of the time abort, *i.e.*, a protocol that in the absence of failures will often reach an abort decision with no valid reason. **Rule 5** says that if a failure occurs at any point of the protocol execution, and this failure does not last for a long period of time, then the processes should reach a decision. This rule excludes the existence of protocols which, in the presence of a failure that can be tolerated, will not reach a decision after the failure is repaired.

Unfortunately, these rules might lead to **blocking**. A process will **block** if it is unable to communicate with other processes during its uncertainty period. This process will be blocked from the time of a failure until it is able to communicate with the other processes. This possibility leads to **blocking** in some ACP protocols, such as B2PC. The case of B2PC will be studied in detail in chapter 4, where we will introduce P2PC, the subject of this thesis.

2.4.2. THE COMMUNICATION NETWORK

A computer **network** can be defined as an *interconnected collection of autonomous computers that are capable of exchanging information among themselves* [ÖZSU91].

The **network** allows these autonomous computers to exchange information among themselves. Computers on a network may be referred to as **sites**, **nodes**, or **hosts**. Computer networks may be classified according to many criteria; one criterion is **geographic distribution**, another is the structure of **interconnection**. A network consisting of a number of sites interconnected over a large geographical area, where the **distance** between any two sites is greater than 20 km, is called a **wide area network**(WAN). A computer network that exists on small geographical area, is called a **local area network**(LAN); the **distance** between any two sites of a LAN is less than 20 km.

The network connection, which is also called the network topology, may be either a: **star**, **ring**, **mesh**, **bus** or other configuration [DESA91].

1. **Star** topology: all sites or nodes are connected to a central node.
2. **Mesh** topology: Each node is directly connected to all other nodes.
3. **Bus** topology: All nodes are connected by taps to a single linear cable.
4. **Ring** topology: Nodes are connected in a circular manner.

2.5. FAILURES

Hardware failures in a computer system can be attributed to deficiencies in its components, in its design, in its construction or assembly. Two types of failures can occur in a system, **permanent** and **non-permanent**. A permanent or hard fault causes an irreversible change in the behavior of the system. Recovery from these failures requires an external intervention to repair the fault. The other type of failure, a **transient failure**, manifests occasionally due to an unusual state of the system, heavy load, sudden change in the room environment or other cause. It may be possible to recover from some of these transient failures, since the fault can be traced to a component of the system [ÖZSU91].

Two approaches are followed to structure a reliable system, **fault tolerance** and **fault prevention**. **Fault tolerance** refers to building a system that recognizes the occurrence of faults. This system includes mechanisms to detect faults when they occur, and try to repair them. **Fault prevention** involves building a system that prevents the occurrence of faults; this system will be error-free, with no possibility of certain errors occurring. Building a totally fault-preventing system is difficult. A reliable system will make use of

these two notions, by trying to eliminate error occurrence, and in case errors occur it is able to detect them and recover internally.

2.5.1. FAILURES IN A DDBMS

In a distributed DBMS many types of errors and failures can occur, including **transaction failures, site failures, media failures** and **communication failures.**

Transaction failures: A transaction can fail for a number of reasons. It can fail due to an incorrect input, due to a deadlock encountered during its lifetime or due to failure in performing a certain test, etc. The recovery from a transaction failure is simply to abort it.

Site failures: A site failure may be due to a **hardware failure,** including failures in the processor, main memory or power supply. It may also be due to a **software failure,** a bug in the operating system or in the DBMS. It is assumed that a site failure will be **fail-stop** [BERN87], hence all transactions taking place on the site where it fails will stop. A site failure will lead to main memory loss; on the other hand, the database is stored on secondary storage. Upon recovery the state of the database may be restricted to the last state before it failed.

Media failures: They occur when the secondary storage at a local site fails. Such failures may be due to an **operating system** or a **hardware device failure.** The solution to such failures is to use multiple devices; when a device fails the system uses the backup device which keeps a copy of the data. Such failures do not affect the performance of other sites in a distributed database system, since they are resolved within the failed site.

Communication failures: The three types of failures discussed above are common to both central and distributed DBMS. However, communication failures occur only in a DDBMS. The most common ones are: corrupted messages, messages delivered in the wrong order, lost messages and line failures. The first two types of failures are resolved by the communication network, which has its own techniques to detect corrupt and misordered messages.

The last two types, the line faiiures and lost messages, are the consequence of communication link failures. A communication link failure may divide the network into two or more partitions which are unable to communicate with each other. Such a failure

is called a **network partitioning**. When a network partitioning occurs, the sites are still operating, but they can only communicate within their partition.

If messages cannot be delivered, we assume that the network does nothing about it. Hence, it is the responsibility of the database software to detect lost messages. One way to do this is by the use of **timeouts**. Timeouts are usually set to a value greater than the maximum round-trip propagation delay of a message in the network, for the destination to respond to the message and for the response to reach the source. Timeout values may vary from system to system, depending on the network topology used and the load of the system.

This aspect of failure in a DDBMS does not exist in the central DBMS, where failures are complete; either the site operates or it doesn't. However, a part of a distributed DBMS may be functioning while the other part has failed. This aspect makes a DDBMS more reliable, but also it imposes more overhead in detecting errors and in minimizing their effects on data consistency.

2.6. Concluding Remarks

In this chapter, we have reviewed the different components of a distributed DBMS. It is clear that a distributed DBMS offers more advantages than a central DBMS. However, more care must be taken in resolving the possible system errors that might occur, leading to data inconsistency. In the next chapter, we review the work done in concurrency control to improve the reliability and performance of DDBMS.

CHAPTER 3

REVIEW OF DISTRIBUTED CONCURRENCY CONTROL ALGORITHMS

In this chapter we review some of the concurrency control algorithms that are implemented and discussed in recent database literature. We consider two classes of algorithms; the first class deals with the algorithms that are based on **Two-Phase Commit** and **Two-Phase Locking** protocols and enhancements of those algorithms. The second class deals with a set of algorithms that implement different strategies by using similar mechanisms to 2PL and 2PC, but violate their strict rules. Then, we show the strategies and models used in performance studies of commit protocols.

3.1. 2PC AND 2PL ENHANCEMENT ALGORITHMS

In [TRAV92] **optimization strategies** and **performance** issues of **2PC** were studied. A 2PC protocol was designed that insured atomicity of updates even in the case of failures. "We have implemented the Two-Phase commit for five different topologies: tree, star, chain, ring and complete graph" [TRAV92]. In the case of a **tree,** the algorithm was called **Hierarchical Two-Phase Commit.** The communication of messages in the Hierarchical case as done in the first phase from the root down to the leaves, and in the second phase from the leaves up to the root node. Performance was evaluated in terms of message complexity; the comparison was drawn between **Hierarchical, Circular** and **Distributed Algorithms.** The message complexity for **Circular** broadcast algorithms was $2n$ messages, for **Distributed** algorithms $2n+1$ messages, and for **Hierarchical** models $2(d+n-1)$ messages. Here, d is the number of subtrees and n is the number of sites.

In [WOLF90] four variants of the 2PC were compared, the **Tree Commit, Decentralized, Linear** and **Central-site** algorithms. The results in [WOLF90] indicated that communication costs of Tree Commit are equal to these of the Linear and Central-site algorithms; its communication time cannot be worse, but it can be twice as fast as they. The communication time of the Decentralized algorithm is better than that of Tree Commit, whereas Tree Commit is better as far as communication cost is concerned. When comparing the communication complexity of the latter two algorithms, the product of communication cost and communication time, Tree Commit is better.

In [STAM90] a **Coordinator Log Transaction Execution** protocol is proposed. This protocol eliminates two rounds of messages, and reduces the number of **log forces** needed for distributed atomic commit (A log force is when the DM moves the log of a transaction from volatile storage to non-volatile stable storage). In the absence of failures, the **Presumed Commit** protocol (PC) eliminates the last round of 2PC and halves the number of log forces compared to 2PC. Another way used to reduce failures was the **Early Prepare** (EP) protocol, which lets each Data Manager enter the prepared state after it performs its work and before it replies to the coordinator. Finally, a **Coordinator Log** (CL) is proposed, wherein all the log records for the transaction are placed in a single log. This results in minimizing the number of log forces required to execute a given transaction.

3.2. ALGORITHMS THAT VIOLATE THE RULES OF 2PC AND 2PL

In [LEVY91], a revised 2PC protocol is described to overcome the difficulty of the potential unbounded delay that a transaction may have to endure if certain failures occur. In the revised protocol, the transaction at a participant site is committed or aborted locally and locks are released as soon as the site votes to commit or to abort. This scheme solves the indefinite blocking problem of the B2PC. This protocol is called the **Optimistic Two-Phase Commit** Protocol (O2PC). Later, if it is found that the transaction is to be aborted, then its effects are undone semantically using a **compensating transaction.** The **Optimistic** 2PC protocol is a modification of the B2PC protocol. The same message exchange is carried out as in the B2PC protocol. If a site votes to abort T_i, then as in the standard protocol, an abort is sent back to the coordinator. In this case, the locks held by the transaction are released at once but without waiting for the coordinator's final commit or abort message.

The key to an adequate solution when a global Transaction aborts is the notion of **compensating transactions.** They are intended to handle situations where it is required to undo a subtransaction which had committed and whose updates have been read by other transactions, without using cascading aborts. Unfortunately, the idea of compensating transactions cannot work in environments where a missile has been fired or funds delivered; these effects cannot be undone by a compensating transaction.

In [DASG90] a **Five-Color Concurrency Control** protocol, that applies to general databases, is presented. The protocol uses five kinds of locks: **read locks, intent locks,**

write locks and two types of **marker** locks. This protocol requires each transaction to pre-declare its read and write sets. This can be achieved by static data analysis by the query compiler; the pre-declared read and write sets need not be the exact read or write sets, but may be a superset of them. The performance depends, however, on the closeness of the pre-declared sets to the actual read and write sets. The Five-Color protocol allows Non-Two-Phase locking by keeping track of transaction ordering using the pre-declared read and write sets.

In [AGRA92], a new relationship between locks called **Ordered Sharing** is proposed, to eliminate blocking that arises in the traditional locking protocols in real-time database systems. **Ordered Sharing** eliminates blocking of read and write operations, but may result in delayed commitment. This delay is exploited to allow transactions to execute in the slack of delayed transactions. For example, in order to eliminate read/write blocking, a transaction T_j can be granted a write lock on an object even if a transaction T_i holds a read lock on the same object. The rule of the ordered sharing is as follows :

"If T_j acquires a lock with an ordered shared relationship with respect to a lock held by another transaction T_i, the corresponding operation of T_j must be executed after that of T_i; furthermore, T_j cannot commit until T_i terminates." [AGRA92]. This scheme places on T_j the burden of ensuring the consistency of the shared objects.

In [MOHA92] a **transient versioning** method is proposed. This scheme permits read-only transactions, that do not mind reading a possibly slightly old but still consistent version of the database, to execute without acquiring any locks.

3.3. Performance Evaluation of Concurrency Control Algorithms

In this section, we review the different implementations, models and parameters used in database performance studies. Many studies have been done to evaluate the performance of concurrency control algorithms. Some of their results are contradictory, as noted in [AGRA87]: "These performance studies are informative, but the results that have emerged, instead of being definitive, have been very contradictory." The main reason behind this is the different implementation models and parameters chosen for the simulation models; the choice of models and parameters are very important to any simulation study. One of the greatest problems in performance study is to choose

appropriate parameter values, which are able to demonstrate the performance characteristics of the protocols to be studied.

In [AGRA87] three main parts of a concurrency control model were examined: A **database system** model, a **user** model, and a **transaction** model. The **database system** model defines the relevant characteristics of the system's hardware and software including CPU's and disks. The **user** model describes the temporal pattern of arrival of user transaction requests. The **transaction** model captures the reference behavior and processing requirements of the transactions in the workload. Three main algorithms were compared: **blocking, immediate restart** and **optimistic**.

In [AGRA87] there are a fixed number of terminals from which transactions originate; there is also a limit on the number of transactions allowed to be active at any time in the system. If the system has already a full set of active transactions, any new transaction enters the ready queue, where it waits for an active transaction to commit or to abort. The transaction then enters the concurrency control queue, and makes the first of its concurrency control requests; if this is granted, the transaction proceeds to the object queue and accesses its first objects. If the result of a concurrency control request is such that a transaction must block, it enters the blocked queue until it is once again able to proceed. If a request leads to a decision where a transaction must restart, it goes back to the ready queue and restarts after a randomly chosen delay. When a transaction decides to commit, if it is read-only it is finished; if it involves writing, then it enters the update queue and writes the deferred updates into the database.

The database is assumed to be **main memory resident** in [HUNG92]. **Real-time databases** are main memory resident, to support fast input output response. In the model, it is assumed that for each transaction, the **deadline** and **execution** times are known upon its arrival. Before the execution of a transaction, its deadline will be compared with its remaining expected execution time. If it is found that it is not feasible to complete before its deadline, the transaction will be immediately decided to **abort** so as to minimize the amount of wasted work. In this model, transactions will be continuously generated. The inter-arrival time is assumed to be exponentially distributed. Each arriving transaction will be put in the ready queue, which uses the discipline **priority on deadline;** a transaction has a priority over another transaction if the deadline of the first is before the deadline of the second. The performance of the concurrency control protocols in this paper was evaluated by calculating the **missing ratio** and the **relative missing ratio**. A *relative*

missing ratio is calculated as the *missing ratio* divided by the *missing ratio* for the **base model** which has no data contention. A *missing ratio* is the mean proportion of transactions missing their deadlines in each set of simulation runs, each set consisted of 6 runs of 1000 terminated transactions.

An analysis of **concurrency coherency** protocols for distributed transaction processing with **regional locality** was performed in [CICI92]. The simulation results were presented for a distributed system with 10 local sites, each connected to the central site through a link with 200 ms communication delay. The central CPU has 10 MIPS, while each local site has 1 MIPS. Transaction arrivals are Poisson-distributed processes, with the same arrival rate at each distributed site. A global lock space of 32K elements was used; for local transactions each site makes lock requests uniformly over one tenth of the entire lock space, while central transactions make lock requests over the entire lock space. The simulation maintains a lock table, explicitly simulates lock contention, and waits for locked entities.

A study on **empirical performance evaluation** of Concurrency Control protocols for database sharing systems was done in [RAHM93]. A major effort was invested in implementing a detailed simulation system for message-based Database Sharing complexes. The system is structured in a modular way such that different algorithms and realization strategies for the main component may easily be incorporated. In this paper, a **trace-driven** approach was chosen instead of **synthetic** workloads. The simulation system uses a representation called reference string, containing only the relevant record types for the trace. Four different types were essential for the purpose of the study: (1) A **begin of transaction** (BOT); (2) An **end of transaction** (EOT) record for every transaction; (3) a **FIX** and an (4) **UNFIX** record for every page record. A page reference is actually represented by the FIX record, while the UNFIX record is used to indicate to the buffer manager that the page may be considered for replacement.

Simulation runs were conducted for six different transaction loads originating from real applications with a non-relational DBMS. The largest reference string contained over 1 million page references and 17,500 transactions. But for this size of load, simulation time turned out to be too long, so most runs had to be conducted for smaller loads. In the simulation, the cost for transaction processing is modeled by requesting a certain number of instructions for every **unit of processing** (UP), which is either a page reference, a BOT, or an EOT. The values of "Number of instructions per UP" are based on the path

length measurements, and differ from load to load. The simulation system was built in PL/1, and models a Database sharing system with a random number of nodes. Communication costs are represented by CPU overhead for sending, receiving and processing messages, as well as communication delays for their transmission in the network. For each of the nodes a fixed multiprogramming level p is applied, so the maximum level of programming is $n*p$ where n is the number of sites. The simulation system determines throughput and response time as the main performance measures. Throughput is not expressed as transactions per second, since transactions differ significantly in size; instead the number of UP's per second is used as the throughput measure.

3.4. Concluding Remarks

After reviewing the different concurrency control algorithms and the various simulation models that exist for these algorithms, one could remark that all these studies were made to improve the performance of a database system and to maintain the maximum level of reliability and confidence in the system. The common factor in the evaluation of the database systems is how fast the system can be. This is indicated by the level of throughput; the higher the rate of throughput, the better the system is. All these studies were based on **synthetic workloads** except the last one [RAHM93], where a **trace-driven workload** taken from a real-life application was used. The reason mentioned in [RAHM93] that others used a synthetic workload was that obtaining a real-life application was difficult. However, one may argue that a real-life application is a special case because it applies to a specific implementation, and cannot be generalized.

The common goals of various studies are summarized by the following points:

- To improve the system **performance**.
- To improve the system **reliability**.
- To reduce the **cost** of the system.
- To maintain the **simplicity** of the system.

In the next chapter we propose a P2PC that satisfies all the above characteristics, and which gives a second chance to a global transaction before deciding to abort it. Also we study, in detail, the following three protocols: B2PC, O2PC, and P2PC. These algorithms are presented along with a case study of their behavior in the presence of failures.

CHAPTER 4

DESCRIPTION OF B2PC, O2PC AND P2PC PROTOCOLS

In this chapter we describe and compare the following commit protocols for a distributed DBMS: **Basic Two-Phase Commit, Optimistic Two-Phase Commit** and **Prudent Two-Phase Commit.**

4.1. BASIC TWO-PHASE COMMIT (B2PC)

Basic Two-Phase Commit is a simple and elegant protocol; it ensures the atomicity of a global transaction in a distributed database system. It extends the local atomic commitment of a transaction to distributed transactions, by insisting that all subtransactions reach the same decision; either they all **commit** or they all **abort.**

A description of **B2PC** that does not consider failures is as follows[ÖZSU91]:

I. Initially, the coordinator writes a **begin_commit** protocol record in its log; then it sends the **vote-request** to the involved sites, and enters the **wait** phase.
II. When a participant receives the **vote-request**, it checks whether it can commit the transaction. If so, the participant writes a **ready** record in its log, and sends a **vote-commit;** otherwise it writes an **abort** record, and sends an **abort** message to the coordinator.
III. After the coordinator has received a reply from every participant, it decides whether to **commit** or to **abort** the transaction. If even one participant has voted No, then the decision of the coordinator will be to write an *abort* in its log, to send an *abort* message to all the participants, and to enter the *abort* state. If all the participants voted Yes, it writes a *commit-record*, sends a global *commit*, and enters the *commit* state.
IV. The participants either **commit** or **abort** the transaction according to the coordinator's *decision* and then send back an *acknowledgment*, at which point the coordinator terminates the transaction by writing an *end-of-transaction* log in the log-record.

Two important points can be observed in the B2PC protocol:

- First, B2PC permits a participant to *unilaterally* abort a transaction until it has decided to register an affirmative vote.
- Second, once a participant decides to vote Yes, it cannot change its decision.

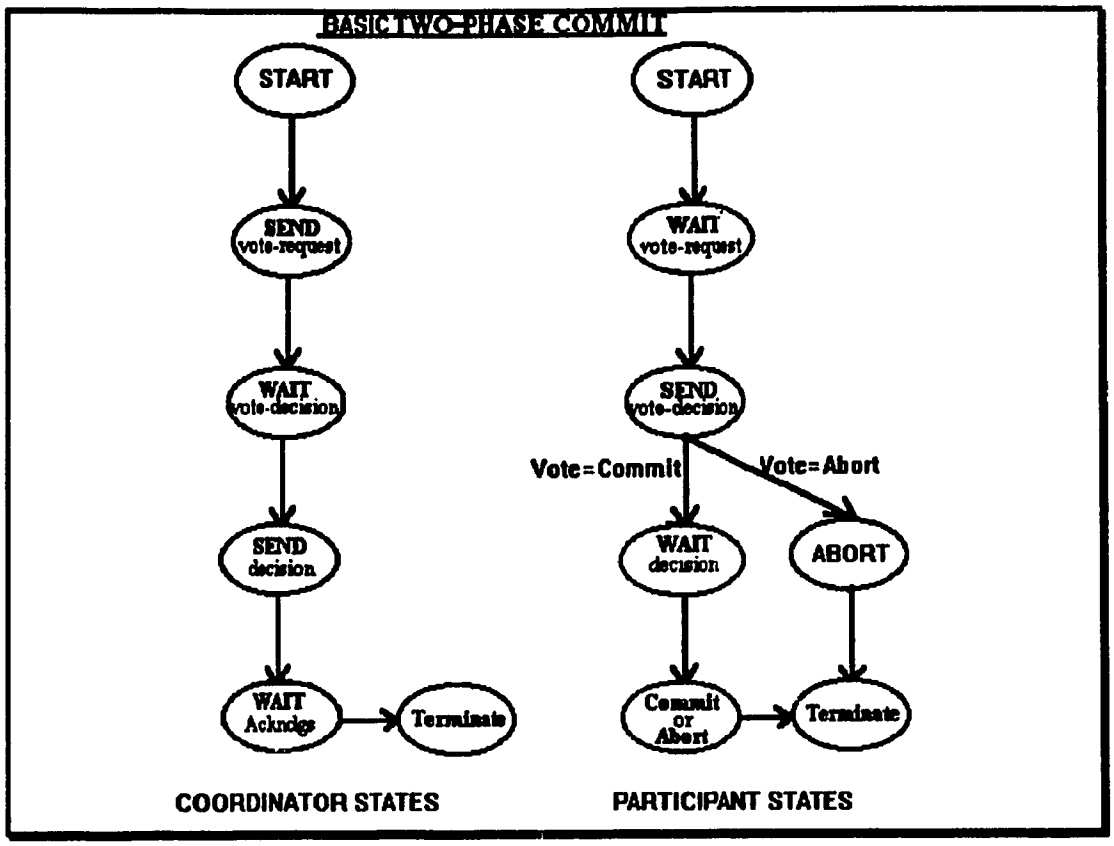

Figure 4.1. Basic Two-Phase Commit transition states in the absence of failures

Note that the coordinator and participants enter **states** where they must **wait** for messages from one another (Figure 4.1). To guarantee that they can exit from these **states** and terminate, **timeouts** are set usually according to the *maximum* delay that a message can encounter. If the expected message does not arrive for one reason or another before the **timeout** expires, then the process times out and invokes its **timeout protocol.**

4.1.1. Dealing with Site Failures

A site failure that occurs at a participant's or coordinator's site can delay the global transaction commitment until the site recovers. The period of time a site needs to recover is **non-deterministic**, and can be very large compared to the lifetime of a transaction. Consequently, the other transactions waiting for a reply from that particular site will

block, holding their locks, and prevent other transactions from using the objects they have locked. Since the B2PC protocol is subject to blocking, participants should be able to terminate after their timeouts have expired, either normally or abnormally. **Termination** protocols are called when a timeout is detected, to terminate a transaction, so that locks are not held unnecessarily for a long time. Termination protocols serve the timeouts for both the coordinator and the participant processes. The method for handling timeouts depends on the timing, as well as on the types, of failures. We need therefore to consider failures at various points during the execution of a Basic Two-Phase Commit protocol.

There are two places where the coordinator can timeout: during the *wait vote-decision* state, and the *wait acknowledgments* state (Figure 4.2).

I. **Timeout** in the *wait vote-decision* state. In this state the coordinator cannot decide unilaterally to commit, since this will violate the global commit rule. On the other hand, it can unilaterally decide to abort the transaction, and send a global abort message to all participants.
II. **Timeout** in the *wait acknowledgments* state. In this case the coordinator is not certain that the *commit* or *abort* procedures have completed at all participating sites. Thus the coordinator will resend the global commit to those sites which have not yet responded, and wait for their acknowledgments.

A participant can timeout in two states: the *wait vote-request* state and the *wait decision* state (Figure 4.2).

I. **Timeout** in the *wait vote-request* state. In this state the participant can unilaterally decide to abort, and write an abort decision in its log.
II. **Timeout** in the *wait decision* state. In this state the participant has voted to commit the transaction, but does not know the global decision of the coordinator. The participant is unable to make a unilateral decision, since it cannot change its decision from a Yes to a No. In this case, the participant will remain blocked until it receives a commit or abort decision.

Let us consider a **centralized** communication structure, where participants can only communicate with their coordinator. In this case the participant, after timing out in the *wait-decision* state, must ask the coordinator for its decision and wait for its response. If the coordinator has failed, then this participant will be blocked. If the participants are

allowed to *communicate with each other* at the time of *commit*, then a blocked transaction might reach a decision from other participants without waiting for the coordinator's decision. We will use the latter case in our study.

A *termination protocol* in the case where all participants can communicate with each other at the time of commit can be as follows [ÖZSU91]: If P_i is the participant which timeouts waiting for the coordinator's *decision*, and P_j are the rest of the participants, then all P_j would respond in the following manner:

I. If P_j is in the *wait vote-request* state, this means that P_j has not voted yet. It can therefore unilaterally abort the transaction and reply by a *vote-abort* message to P_i.
II. If P_j is in the *wait-decision* state, then P_j cannot help P_i since it is in the same situation.
III. If P_j is has received a global *commit* or *abort* decision, then it can send P_i a global *commit* or *abort* decision.

P_i will interpret P_j responses according to the following possible cases:

I. P_i receives at least one *vote-abort* message, then P_i will *abort*.
II. P_i receives a note that all the other processes are in the *wait decision state*, then P_i can't reach a decision; in this case **blocking** is unavoidable.
III. P_i receives at least one global *commit* or *abort* decision, then P_i will decide accordingly and terminate.

4.1.2. RECOVERY PROTOCOLS

These protocols are used by a coordinator or a participant when a site fails and then restarts:

In the case of a coordinator's Site failure:

I. The coordinator fails before sending the *vote-request*, therefore it will continue by sending the *vote-request*.
II. The coordinator fails while waiting for *vote-decisions*, upon recovery it will restart the commit process by sending the *vote-requests* one more time.

III. The coordinator fails while in the *wait acknowledgments* state. In this case the coordinator would have informed the participants of its decision; it does not need to do anything upon recovery, if all the acknowledgments were received; otherwise the termination protocol is invoked.

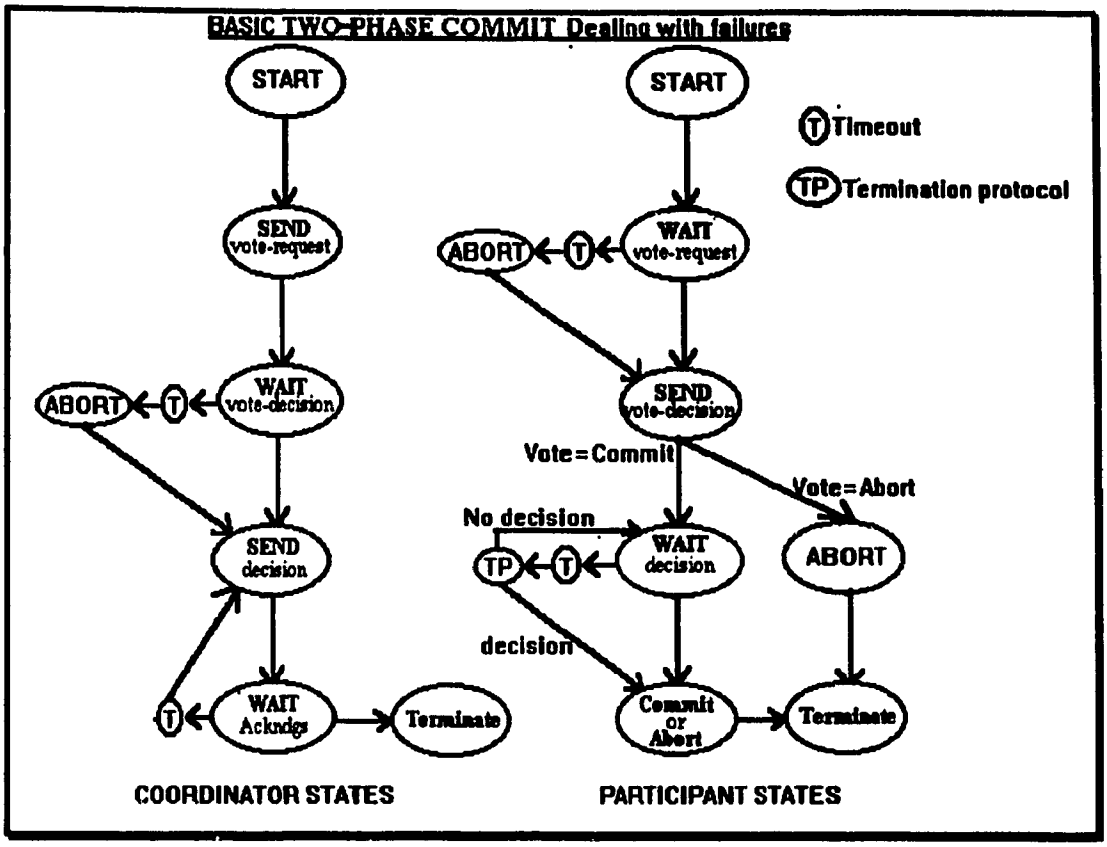

Figure 4.2. Basic Two-Phase Commit transition states in the presence of failures

In the case of a participant Site failure:

I. A participant fails in the wait *vote-request* state. Upon recovery the participant should abort the transaction, because the coordinator will be waiting for *vote-decisions* and may have sent *abort* decisions during the failure of that site.

II. A participant fails while waiting for the *decision.* Upon recovery the participant at the failed site can treat this as a *timeout* in the *decision* state and calls the termination protocol.

III. A participant fails while in the *Abort* or *Commit* state. These states represent the termination condition, so after recovery it does not need to do anything.

In all of these cases we have considered that writing a record in the log and sending a message is atomic, and no failures can occur between those two actions. This assumption is made for simplicity.

```
Coordinator's algorithm:
begin;
 start;
 send vote-request to all participants;
 initialize Timeout; (* reset the value of the timeout *)
 WHILE (NOT Timeout AND NOT (all vote-decisions received)) DO
     Receive vote-decisions;
 END;
 IF (NOT (all vote-decisions received)) THEN
    Send ABORT message;
    ABORT;
  ELSE
    IF (all vote-decisions = Yes) THEN
       Send COMMIT messages;
       COMMIT;
       initialize Timeout;
       WHILE (NOT Timeout AND NOT (all acknowledgments received)) DO
            receive acknowledgments;
       END;
       IF NOT (all acknowledgments received) THEN
          resend COMMIT messages to sites that did not send acknowledgments;
       END;
      ELSE
       Send ABORT messages;
       ABORT;
     END;
  END;
  Terminate;
end;
```

Figure 4.3. Coordinator's algorithm for Basic Two-Phase Commit protocol

4.2. Optimistic Two-Phase Commit (O2PC)

The **O2PC** protocol as described in [LEVY91] is based on the **optimistic** assumption that the probability of B2PC terminating unsuccessfully is very low. Therefore, locks can be **released** earlier than in the B2PC protocol. This will result in reducing the **waiting** time due to data contention, thereby improving the overall performance of the system. The **O2PC** is a slightly modified version of the B2PC protocol; the same message exchange is carried out as in B2PC. In B2PC releasing of locks is done after the transaction receives

the coordinator's decision, in case the transaction vote is Yes. In case the transaction vote is No, it sends its decision and releases its locks (Figure 4.5). However in **O2PC** the transaction voting Yes does not have to wait for the coordinator's decision; it sends its decision, commits on the local site, and then releases its locks.

```
Participant's algorithm:
begin
  START
  Initialize Timeout; (* reset the timeout *)
  WHILE (NOT (received vote-request) AND NOT Timeout) DO
     wait vote-request;
  END;
  IF NOT (received vote-request) THEN
      send No decision; (* Unilateral abort *)
      ABORT;
      Terminate;
  END;
  Send vote-decision;
  IF vote-decision = Yes THEN
         initialize Timeout;
         WHILE (NOT Timeout AND NOT (decision received)) DO
                wait-decision;
          END;
          IF decision received THEN
             IF decision = COMMIT THEN
                COMMIT;
               ELSE
                ABORT;
             END;
           ELSE
             decision := Termination-protocol();
              CASE decision OF
                      COMMIT: COMMIT;
                      ABORT :  ABORT ;
                      BLOCK : WHILE ( NOT (decision received))DO
                                wait-decision; (* blocking *)
                              END;
                              IF decision = Yes THEN
                                 COMMIT
                              ELSE
                                 ABORT
                              END;
               END;
          END;
  END;
  Terminate;
end;
```

Figure 4.4. Participant's algorithm for Basic Two-Phase Commit protocol

4.2.1. COMPENSATING TRANSACTIONS

The uncoordinated local commitment may result in the violation of the transaction's **atomicity** rules. If at least one of the subtransactions of a transaction *T* votes to *abort*, this will result in *T committing* at some sites and *aborting* at other sites, thus violating the *atomicity* rule of a transaction. To rectify this situation, the effects of the committing subtransaction must be undone locally at their sites. The solution for this problem is the idea of **compensating transactions** [LEVY91], intended to handle situations where it is required to undo the effects of a committed subtransaction the results of which have been used by other transactions. Compensating transactions, along with the O2PC, are a way of ensuring the *atomicity* of a transaction. A compensating transaction undoes the effects of a committed transaction *semantically*, without resorting to cascading aborts which affects the system's throughput. The side effect of compensation is that the system may not be in the same state as if the original transaction had not taken place, while on the other hand, compensating transactions guarantee the consistency of the system. The success of a compensating transaction is guaranteed by the property of **persistence** [LEVY91], which ensures that a compensating transaction completes successfully.

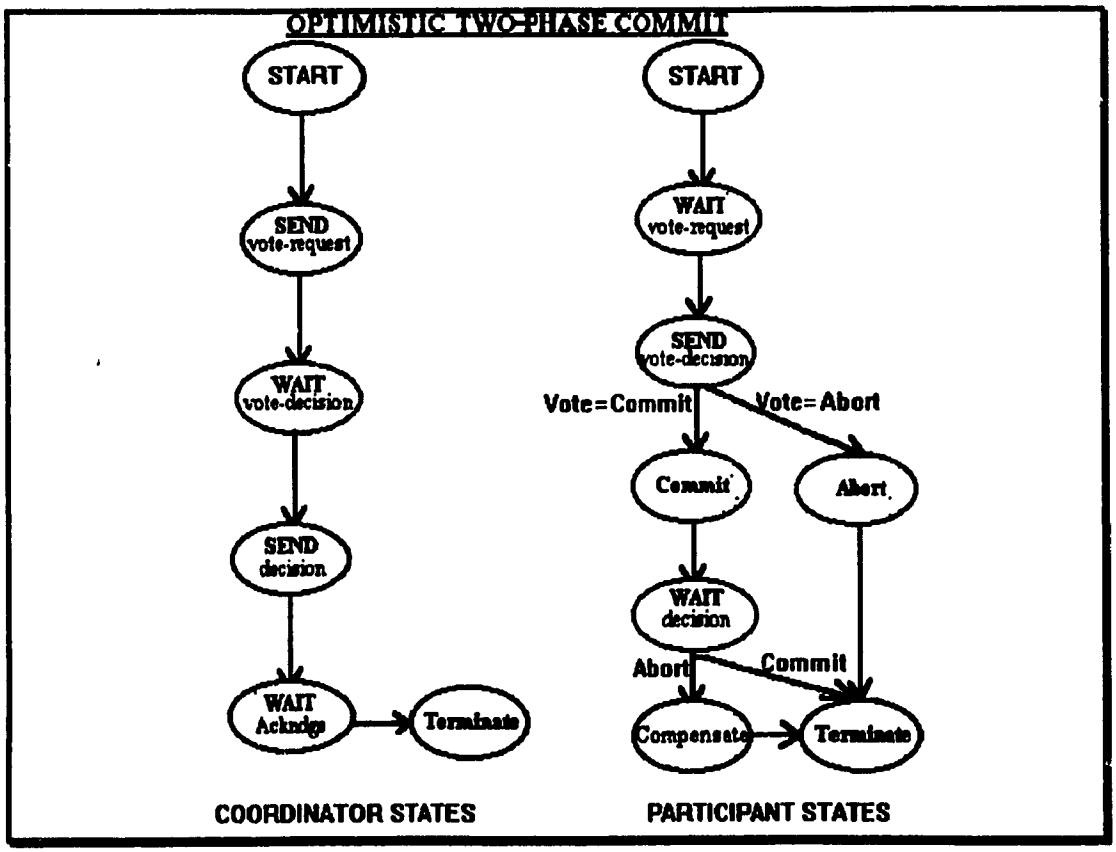

Figure 4.5. Optimistic Two-Phase Commit transition states in the absence of failures

4.2.2. O2PC AND FAILURES

It is obvious from the description in section 4.2.1 that O2PC is identical to B2PC in dealing with failures, except for the case where the failure results in aborting a transaction that will lead to compensating transactions at all sites where subtransactions have committed. The originators of this algorithm [LEVY91] claim that the overhead of a compensating transaction is compensated for by the gain of releasing the locks earlier than the B2PC, thereby increasing the system's throughput.

We have seen in the description of a site failure in section 4.1.1 that the transaction will block if and only if all participants are in their uncertainty period at the time of the failure, and the failure occurs before the coordinator sends the *decision* messages. Therefore the probability of holding the locks for a long time is limited to the probability of having a site failure while all participants are in their *uncertainty* period. In all other cases, a failure will result in a *Commit* or *Abort* without blocking; the *termination protocol* ensures that. We will evaluate the performance of O2PC in Chapter 6, where the simulation and its results are presented.

4.3. PRUDENT TWO-PHASE COMMIT (P2PC)

Unfortunately, neither the B2PC nor O2PC protocols distinguish between communication failures and site failures; they treat both failures in the same manner. This can lead to aborting a transaction even in the case of a transient communication failure. We believe that in this case, the situation can be rectified without the need to *abort* the transaction. We consider that a communication failure may cause a message to be lost, and that the network cannot detect this lost message. The P2PC algorithm, given in Figures 4.8 and 4.9, gives *another chance* to a participant or coordinator before aborting the transaction; this compensates for most lost messages.

In the B2PC protocol, when a participant which has completed its transaction and is waiting for a message timeouts, it is simply aborted. This leads to aborting the global transaction. In the same situation P2PC will allow the participant to resend the *vote-decision* again and wait an additional *timeout* before it decides to abort (Figure 4.9), thereby avoiding unnecessary aborts. In B2PC, if the coordinator did not receive a *vote-decision* from a participant, it sends a No decision and aborts the global transaction, while in P2PC the coordinator is allowed to communicate another time with the participant that

its message was not received, by resending the *vote-request* and waiting one additional timeout before it decides to abort (Figure 4.8). We have excluded the *ping* action in the case of a site failure, as mentioned in [DESA], for the reason of simplicity. However, it can be added in a future study to improve P2PC. In brief, before giving another chance the coordinator or participant *pings* the site it is sending a message to; if the site does not respond, P2PC does not give a second chance. Instead it will act as the B2PC, hence saving the additional messages.

4.3.1. Dealing with Communication Failures

In B2PC there are three places where the participant and coordinator must communicate by sending messages; we will consider communication failures in these three cases:

I. If a *vote-request* message to a participant is lost, then the coordinator will timeout waiting for a *vote-decision* from that participant. Also, the participant that did not receive the *vote-request* will timeout after waiting for the coordinator's *vote-request*, which was already sent. The other participants will time out in their uncertainty period (*wait-decision state*). In such a situation, B2PC results in an *abort* of the current transaction; in O2PC this would result in an *abort* plus *Compensating Transactions* on the sites that have committed.

II. If a *vote-decision* message from a participant to a coordinator is lost, then the coordinator will timeout in the *vote-decision* phase. In such a situation B2PC leads to an *abort*; O2PC leads to an *abort* plus *Compensating Transactions* on the sites that have committed.

III. If a *decision* message to a participant is lost, then this participant will time out waiting for the coordinator's *decision*, and according to B2PC the coordinator not receiving the *acknowledgment* will resend the decision, or a termination protocol will be called. Thus, this situation can be rectified with the B2PC protocol.

We conclude that the B2PC protocol takes care of only the last situation, and solves the other two situations by aborting the global transaction. The P2PC protocol attempts to take care of the other two situations, without the need of aborting the global transaction.

4.3.2. P2PC ALGORITHM

The following illustrates the P2PC algorithm and how it handles the situations that B2PC and O2PC do not handle:

There are two places where the coordinator can time out: during the *wait vote-decision* state, and in the *wait acknowledgments* state (Figure 4.6).

I. **Timeout** in the *wait vote-decision* state. In this state the coordinator cannot decide unilaterally to commit, since this will violate the global commit rule. On the other hand, it can unilaterally decide to abort the transaction and send a global abort message to all participants. Before the coordinator decides to unilaterally abort the transaction, it gives another chance to the participants whose votes were not received yet, by sending them another *vote-request* message. Subsequently, the coordinator waits for an additional *timeout*. If, after the second timeout, the coordinator doesn't receive all the vote messages, then it decides to abort the transaction (Figure 4.8).
II. **Timeout** in the *wait acknowledgments* state. No modifications are done in this case; it is handled in the same manner as in B2PC.

A participant can time out in two states, the *wait vote-request state*, and the *wait decision* state (Figure 4.7).

I. **Timeout** in the *wait vote-request* state. In this state, in the B2PC protocol, the participant unilaterally decides to abort, and writes an abort decision in its log. However, in the P2PC protocol, before deciding to unilaterally abort, the participant gives the coordinator another timeout chance (Figure 4.9).
II. **Timeout** in the *wait decision* state. In this state the participant has voted to commit the transaction, but does not know the global decision of the coordinator. The participant is unable to make a unilateral decision, since it cannot change its decision from a Yes to a No. Before calling a *termination protocol*, the participant resends its decision and gives the coordinator another *timeout* chance (Figure 4.9).

To show logically that P2PC handles the two cases that B2PC does not, we will consider its actions for all possible cases:

Case 1. A *vote-request* message from a coordinator to a participant is lost: In P2PC the coordinator must send another *vote-request* to the sites that did not vote and wait for another *timeout*, and the participant that did not receive the *vote-request* will give the coordinator another chance by resetting its *timeout*. In the meantime, if any other participant timeouts in its uncertainty period, before calling the *termination protocol* it will give the coordinator another chance by resending its *vote-decision* and by resetting its Timeout. Therefore the participants that did not receive the *vote-request* the first time, because of a communication failure or a delay in receiving the coordinator's message, have another chance to send their votes, without the need to abort the transaction.

Case 2. A *vote-decision* message from a participant to the coordinator is lost: This will result in the participant's timeout in its uncertainty period, and the coordinator will timeout waiting for a *vote-decision*. In P2PC, all participants that timeout in their uncertainty period send their *vote-decisions* again, and reset their timeouts. The coordinator gives the participants that did not vote another chance by resending them a *vote-request*, and by resetting its *timeout*. Therefore, the participants that did not vote yet or whose *vote-decision* was lost have another chance to vote before aborting the transaction.

Case 3. A *decision* message from the coordinator to a participant is lost: Although this case is solved by the B2PC protocol, we trace it for the sake of completeness. First, the participant will timeout in its uncertainty period, send its *vote-decision* again and wait for another *timeout* before calling the *termination protocol*. Second, since the coordinator did not receive an acknowledgment from this participant, it will send it another decision message.

This demonstrates that the P2PC protocol prevents a global transaction abort in the case of a transient communication failure or delay.

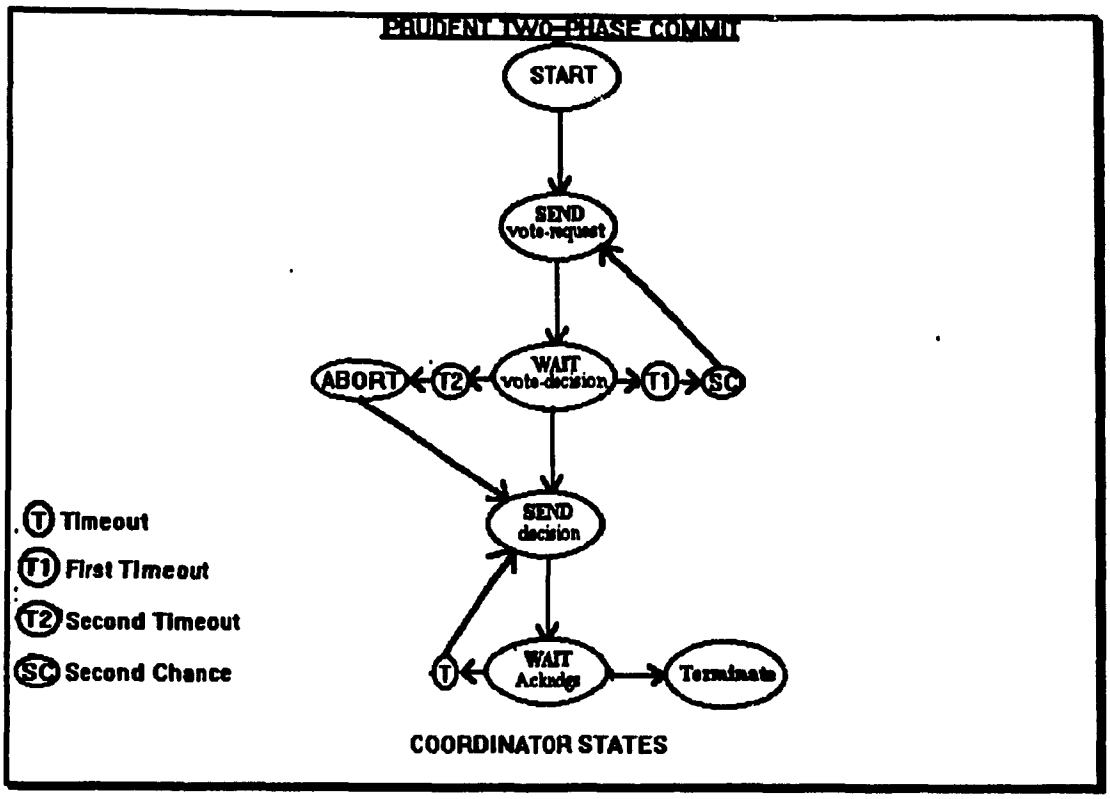

Figure 4.6. Prudent Two-Phase Commit coordinator transition states

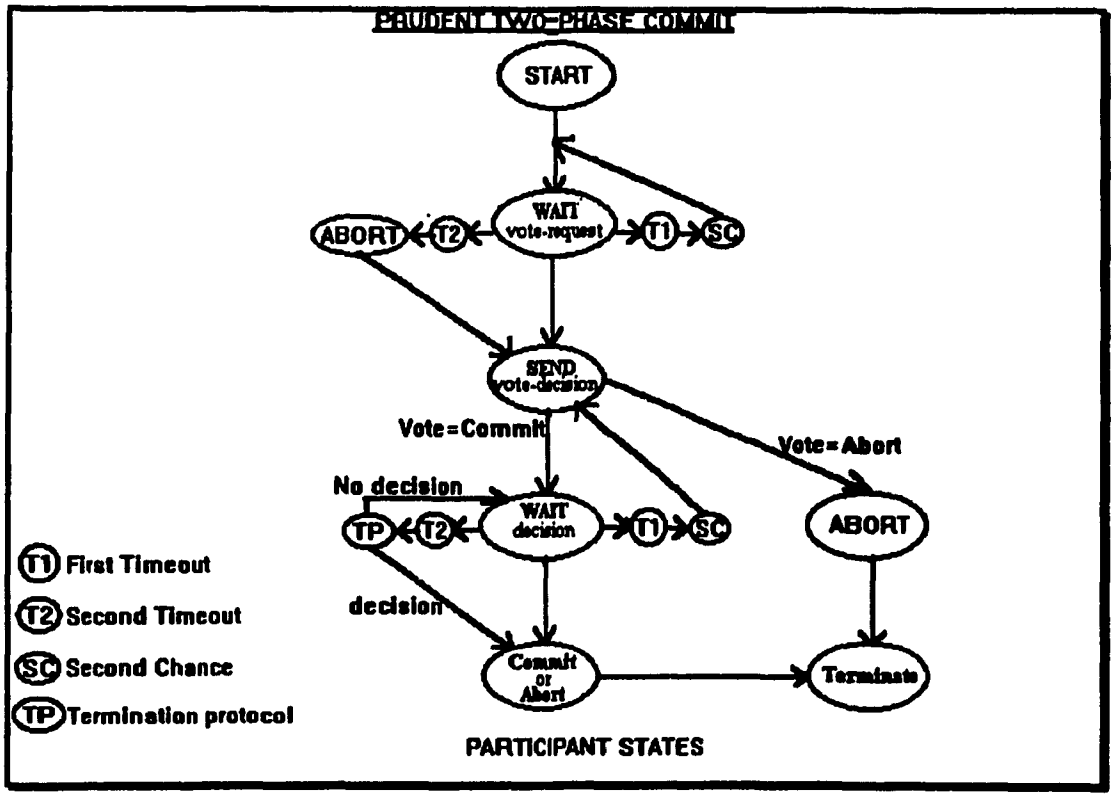

Figure 4.7. Prudent Two-Phase Commit participant transition states

```
Coordinator's Algorithm.
begin;
 start;
 send vote-request to all participants;
 initialize Timeout;  (* reset the timeout *)
 WHILE (NOT Timeout AND NOT (all vote-decisions received) ) DO
     Receive vote-decisions;
 END;
 IF (NOT(all vote-decision received)) THEN
     send vote-requests to participants which did not vote; (* Give another chance *)
     initialize Timeout;
     WHILE (NOT Timeout AND NOT (all vote-decisions received)) DO
        Receive vote-decisions;
      END;
 END;
 IF (NOT (all vote-decisions received)) THEN
    Send ABORT message;  (* Unilateral abort *)
    ABORT;
  ELSE
     IF (all vote-decisions = Yes) THEN
        Send COMMIT messages;
        COMMIT;
        WHILE (NOT Timeout AND NOT (all acknowledgments received)) DO
            receive acknowledgments;
        END;
        IF NOT (all acknowledgments received) THEN
           resend COMMIT messages to sites that did not send acknowledgments;
        END;
      ELSE
        Send ABORT messages;
        ABORT;
      END;
  END;
  Terminate;
end;
```

Figure 4.8. Coordinator's algorithm for Prudent Two-Phase Commit protocol

```
Participant's Algorithm;
begin
  START
  Initialize Timeout; (* reset timeout *)
  WHILE (NOT (received vote-request) AND NOT Timeout) DO
     wait vote-request;
  END;
  IF NOT (received vote-request) THEN
     Initialize Timeout;
     WHILE (NOT (received vote-request) AND NOT Timeout) DO
        wait vote-request; (* give another chance *)
     END;
  END;
  IF NOT (received vote-request) THEN
     send No decision;
     ABORT; (* Unilateral abort *)
     Terminate;
  END;
  Send vote-decision;
  IF vote-decision = Yes THEN
        initialize Timeout;
        WHILE (NOT Timeout AND NOT (decision received)) DO
              wait-decision;
        END;
        IF NOT (decision received) THEN
           resend vote-decision; (* give another chance *)
           initialize Timeout;
           WHILE (NOT Timeout AND NOT (decision received)) DO
                 wait-decision;
           END;
        END;
        IF decision received THEN
           IF decision = COMMIT THEN
              COMMIT;
           ELSE
              ABORT;
           END;
        ELSE
           decision := Termination-protocol();
           CASE decision OF
                 COMMIT : COMMIT ;
                 ABORT  : ABORT  ;
                 BLOCK  : WHILE ( NOT(decision received)) DO
                            wait-decision; (* blocking *)
                          END;
                          IF decision = Yes THEN
                             COMMIT
                          ELSE
                             ABORT
                          END;
           END;
        END;
  END;
  Terminate;
end;
```

Figure 4.9. Participant's algorithm for Prudent Two-Phase Commit protocol

4.4. CORRECTNESS AND RELIABILITY OF P2PC

A transaction in a database system is an execution of a program that accesses a shared database. The goal of concurrency control is to ensure the **atomicity** of transactions executing concurrently. A transaction must satisfy certain properties in order to ensure the correctness of the database system.

These properties are:

I. A transaction is **atomic**, either all of its effects are made permanent to the database or none of them.
II. A transaction must not introduce any **inconsistency** in the data which are modified.
III. Transactions executing in parallel must not **interfere** with each other.
IV. All the effects of a transaction which has terminated successfully are made **permanent.**

The correctness of any algorithm depends on the preservation of these properties. Thus, to prove the correctness of P2PC protocol, we must demonstrate that P2PC preserves the properties of a transaction.

4.4.1. ATOMIC COMMITMENT OF P2PC

A distributed system is composed of a number of local **database systems**, plus a **network** that connects them. A **global** transaction is a transaction that involves more than one site. P2PC orchestrates the execution of **global** transactions, and controls their commitment.

The correctness of a local transaction is ensured by the protocols applied at each local site; in our case it is ensured by the correctness of the **Two-Phase Locking** protocol, which is assumed to be correct. A proof of the correctness of the 2PL protocol can be found in [BERN87]. Thus properties II, III, and IV are ensured locally; the remaining property to be proven is the **atomic** commitment of a global transaction.

An **Atomic Commitment Protocol** (ACP) for a global transaction is an algorithm meant to synchronize the commitment of the distributed subtransactions, such that either all the sites involved in the global transaction *commit*, or they all *abort*. The rules of ACP are described in detail in section 2.4.1.

We can see clearly that the P2PC algorithm, in Figures 4.8 and 4.9, satisfies rules 1-4 (section 2.4.1) of an atomic commitment protocol:

- The coordinator either sends an **abort** or a **commit** decision to all participating sites, and a participating site cannot decide to **commit** unless it has received the **commit** decision from the coordinator.
- A participant is not allowed to change its interim decision once it has voted.
- The Coordinator sends a Commit decision if and only if all sites have reached an interim decision to commit.

To satisfy rule 5 (section 2.4.1) and guarantee that all processes will eventually reach a decision, *Timeouts* are used as in the case of the B2PC algorithm. The difference from B2PC is that P2PC gives an additional *Timeout* in the case of a failure; if the failure persists after this additional *Timeout*, then processes will reach a decision either by a unilateral abort or by calling the *Termination protocol*, thus satisfying rule 5.

4.4.2. EVALUATION OF P2PC

An ACP can be evaluated according to many criteria [BERN87]:

Resiliency: What are the failures that the protocol can tolerate?
Blocking: Can blocking of processes occur, and if so in what situations?
Time complexity: How long does it take to reach a decision?
Message complexity: What is the maximum number of messages to be exchanged in order to reach a decision?
The two first criteria refer to the ACP's **reliability**, and the last two to the **efficiency** of the ACP.

4.4.2.1. RELIABILITY OF P2PC

In the case of a site failure P2PC behaves the same way as B2PC, except for the additional message exchange and the extra Timeout. Note that in the case of a site failure, depending on the timing of the failure, a Timeout will call the appropriate termination protocol and terminate the processes. In the case of a transient communication failure, B2PC will act exactly as if there were a site failure, and will terminate the transaction in the same manner. However, P2PC assumes that the Timeout is not necessarily due to a

site failure, and it gives another chance to the transaction before it terminates it. Therefore P2PC is more resilient to transient communication failures than B2PC, by giving the transaction another chance to recover from the communication failure.

The P2PC protocol does not eliminate the possibility of blocking that existed in the B2PC protocol. The blocking can happen when a process timeouts in its uncertainty period, and the only processes it can communicate with are in their uncertainty period also.

To illustrate blocking, we consider the case where all processes are in their uncertainty period, and the coordinator's site fails before sending the decisions. In this case, the participants timing out in their uncertainty period send their vote-decisions another time, and reset their timeout according to P2PC. Timing out the second time, the participants call the termination protocol, which will return an undecidable answer since all participants are in their uncertainty period. Hence, blocking of participants still occurs in P2PC. Such blocking could be resolved by the SOS protocol [DESA90].

4.4.2.2. EFFICIENCY OF P2PC

The efficiency of P2PC is calculated in terms of time and message complexities.

Time Complexity: In the absence of failures, P2PC protocol requires three **rounds** of messages to reach a decision. This is the same as in the case of B2PC [BERN87]. A round is the maximum time for a message to reach its destination. The use of Timeouts to detect failures is based on the assumption that such a maximum message delay is known [BERN87]. Two messages belong to different rounds if one cannot be sent until the other is received. So, in the absence of failures in P2PC, the three rounds of messages are: **(1)** the coordinator broadcasts the vote-request, **(2)** the participants reply with their vote-decision, and **(3)** the coordinator broadcasts the decision message.

In the case of a transient communication failure, the coordinator must send the vote-requests again to the sites whose votes were not received. Simultaneously, the participants in their uncertainty period resend their votes. Those participants who resend their vote-decisions have already received a vote-request. Hence these two rounds of messages may overlap, and are considered a single round of messages. The other round of messages is the case where a participant did not receive the vote-request the first time. Such a participant must wait for the coordinator's second vote-request before it sends its

vote-decision message. Therefore, this will add two additional rounds of messages to the time complexity; that is **(4)** the coordinator rebroadcasts the vote-requests and the participants resend their vote-decisions, and **(5)** the participants that did not receive the first vote-request send their vote-decisions.

In case of site and communication failures that persist, the termination protocol may need two additional rounds of messages [BERN87]: **(6)** one for the participant that timed out to send a decision-request to other participants, and **(7)** one for the participant that received the decision-request to send the decision.

Therefore, in the presence of failures that persist, P2PC needs two more rounds of messages than B2PC, which has 5 rounds. These two additional rounds of messages are the cost to make the system more reliable. The results and effects of additional messages, and the possibility that they will be compensated for by the fact that the transaction is not aborted, will be shown in detail in the simulation results in chapter 6.

Message Complexity: The message complexity is measured by the **number** and **lengths** of messages issued by the protocol. Since the messages are uniformly short, then the **number** of messages exchanged will be sufficient to calculate the message complexity [BERN87]. In the B2PC protocol the number of messages without failures is $3n$; with failures it becomes $n(3n+7)/2$ [BERN87] in the worst case, where n is the number of participants. These figures do not consider the acknowledgments sent from the participants to the coordinator; they add n messages for a total of $4n$ in the absence of failures, and $n(3n+9)/2$ in the worst case.

In the absence of failures P2PC has the same number of messages, namely $4n$, including the acknowledgments. In the presence of failures we have three cases:

Case 1. The *vote-message* from the coordinator did not reach one or more participants. Suppose m is the number of participants that are in the *wait-vote* state, and k is the number of participants that are in their *wait-decision* state. Here $m + k \leq n$, since a participant cannot be in both *wait-vote* and *wait-decision* states. Furthermore, the sum may be less than n because a failed participant may not be in any state. Therefore the coordinator will send m *vote-request* messages to m participants, and the remaining participants that could timeout in their uncertainty period will send k messages to the coordinator. These will add up to a maximum of n messages in this case.

Case 2. One or more *vote-decisions* did not reach the coordinator. Assume that all participants are in the *wait-decision* state. The coordinator will resend *vote-requests* to the participants whose vote did not reach the coordinator. If the number of participants, m, whose vote was not registered is $0 \leq m \leq n$, the coordinator will send m *vote-requests.* At the same time, participants which timeout in their uncertainty period will send n *vote-decisions.* This will add up to a maximum of $n + m$ messages; in the worst case it will be $2n$ messages.

Case 3. A *decision* did not reach one or more participants; these participants resend their *vote-decision.* At the same time the coordinator, not receiving an acknowledgment from these participants, will resend them the *decision.* In this case the maximum number of messages would be $2n$ if we consider that all *decisions* did not reach all participants.

Scanning the three possible cases, we realize that in the worst case the additional number of messages is $2n$, where n is the number of participants. Hence, in the worst case, P2PC requires a total of $n(3n+13)/2$ messages.

CHAPTER 5

THE SIMULATION MODEL

The objective of our simulation experiments is to validate the P2PC and compare its performance with B2PC and O2PC protocols. To meet our objective, we model a distributed database system which uses these three commit protocols. Each simulation run produces results in terms of the performance metrics discussed later in this paper. A distributed database system is composed of a number of sites, each site representing a centralized database system; the sites are interconnected by a communication network. We have used the incremental concept in implementing the different components of the system; we first build a simple centralized database system, then we move to a distributed database system. This method was used for the following reasons:

- To examine the behavior of our centralized database system independent of the distributed database system.
- To guarantee the correctness of our system, since an error in the centralized database would propagate to the distributed one and be harder to detect.
- To make the implementation of the DDBMS easier, since it is built on top of our DBMS.

5.1. THE PERFORMANCE MODEL

We start by the study of the performance model of a centralized database system; subsequently we move to the distributed one. Agrawal defined the main parts of a concurrency control performance model to be [AGRA87]: a **database system model**, a **user model**, and a **transaction model**. The database system model defines the characteristics of the system's hardware and software, the characteristics of the database, *i.e.* its size and granularity, and the concurrency control algorithm itself. The user model defines the pattern of arrival of requests from users. Finally, the transaction model captures the transaction behavior in the workload.

5.2. THE CENTRALIZED DATABASE SYSTEM MODEL

The database contains 4096 distinct items; each item can have a distinct lock. The concurrency control for the centralized database system is based on locking; we use the Two-Phase Locking protocol (2PL) [BERN87].

5.2.1. SOFTWARE COMPONENTS

The centralized database system is mainly composed of two processes: a **Transaction Manager** (TM), and a **Scheduler** plus a **Data Manager** (DM). The Scheduler and the DM are embedded into a single process, which we refer to as the Scheduler process.

The **Transaction Manager process** is responsible for receiving the transactions and forwarding them to the Scheduler. Once the maximum multiprogramming level is reached, the TM will block all new incoming transactions, and will forward a blocked transaction as soon as one of the running transactions completes its execution.

The **Scheduler process** is responsible for the execution and coordination of transactions. It involves obtaining locks and handling deadlocks and conflicts. First, when the Scheduler receives a transaction, it may delay it for a period of time before the transaction actually starts. After the delay the Scheduler will start servicing read and write operations for that transaction. For each read or write operation issued by a transaction on a data item, the Scheduler will obtain the necessary locks before the transaction can proceed. In case there is a lock conflict, the transaction will be blocked until the lock is obtained. In case the lock is obtained and the resources are all currently being used, the transaction must wait on the appropriate queue until it can be serviced. When a deadlock is detected by the Scheduler, one of the transactions involved in the deadlock will be aborted and restarted by creating a new transaction. The Scheduler will serve the transactions according to FCFS basis, with no scheme for deadlock prevention or avoidance; deadlocks are detected as they occur.

5.2.2. CONCURRENCY CONTROL ALGORITHM

The concurrency control algorithm used in our simulation for the centralized DBMS is the **Two-Phase Locking protocol** (2PL). We use probability theory to simulate deadlock and conflicts. Given the database parameters, the probability formula computes the probability

of a conflict or a deadlock in 2PL. In case a transaction encounters a conflict, it will be delayed for a period of time, which is predicted by a probability formula. In case a transaction encounters a deadlock, it will be aborted and restarted. [THOM91] presents 2PL results from probability theory, and formulas for deadlock and delays; we use these results in the simulation of 2PL concurrency control in the centralized database system. Thomas considered two types of transactions: a **fixed size** transaction and a **variable size** transaction with identical step duration. A fixed size transaction exists in a system where all transactions are of the same size; a variable size transaction exists in a system where the size of transactions may be different. We use these two types of transactions in our system. Below, we present the different formulas that were adopted from this paper.

M	The number of transactions activated in the system.
N	The number of data items in the database.
K	The number of distinct locks requested by a transaction.
s	The mean processing time of a transaction step.
s'	The mean residual processing time of a transaction step, s'=s/2.
K_i	The ith moment of the number of requested locks ($K_i = \sum_{k>0} k^i P_i(k)$). $P_i(k)$ represents the probability that k locks are requested by a transaction.

Table 5.1. Probability Notations and their Meanings

Probability of Result for **Fixed Size Transactions**:
The probability that a transaction requesting K distinct locks experiences a lock conflict during its execution, per lock request assuming the independence of lock conflicts, is:

$$p_{fc} = \frac{(M-1)K^2}{2N} \qquad (1)$$

The probability that a transaction encounters a deadlock of cycle 2 (deadlock that involves two transactions) during its execution is:

$$P_{fD2} \cong \frac{(M-1)K^4}{12N^2} \qquad (2)$$

The delay when a transaction encounters a lock conflict is:

$$W_f = \frac{(K-1)s}{3} + s' \tag{3}$$

Probability of Result for **Variable Size Transactions:**
The probability of lock conflict per lock request is:

$$p_{vc} \cong \frac{(M-1)(K_2+K_1)}{2(K+1)N} \tag{4}$$

The probability of a deadlock of cycle 2 is:

$$P_{vD2} = \frac{(M-1)(K_2-K_1)(K_3-1)}{12(K_1+1)N^2} \tag{5}$$

The delay in the case of a lock conflict is:

$$W_v = \frac{K_3-K_1}{3(K_2+K_1)}s + s' \tag{6}$$

5.2.3. HARDWARE COMPONENTS

Object access requires the use of two resources: the **CPU** and **disks**. The access time and the number of resources are used as parameters in the model. CPUs and disks are considered as serially shareable resources, *i.e.*, a CPU can only service only one transaction at a time. When a transaction needs CPU service, it is assigned a CPU server; otherwise the transaction will be blocked until one is free (this applies also to I/O requests). Thus, the CPU servers can be considered as a pool of identical servers, serving one main global CPU queue. Requests in the CPU and disk queues are served according to the first-come first-served (FCFS) discipline.

5.2.4. TRANSACTION MODEL

In our system the transaction is modeled by the set of read and write requests performed during its lifetime. There is a limit on the number of transactions allowed to be active at any time, specified by the current **multiprogramming level** (MPL). Whenever a

transaction terminates it is automatically replaced by a new one. The transaction execution time starts from the time it is forwarded to the Scheduler, up to the time where it commits or aborts. Once the transaction is in the Scheduler, it will execute all of its read and write operations. When the transaction encounters a lock conflict computed by the probability of conflict equation (1) or (4), according to its type, it will be delayed by a period of time determined by delay equation (3) or (6).

The delay is implemented using the global logical clock of the system. For example, if we consider delaying transactions T_i and T_j respectively by 100 and 110 ms, taking the logical clock as 500 ms at the delay request, then T_i can only resume its execution when the clock value becomes 500+100 = 600 ms, and T_j when the clock value becomes 610. The clock is an infinite process that is incremented by 10 ms each time the control transfers. A transaction can be in one of the following states: running, blocked, or delayed. If a transaction is in the delayed state, then it will wait until its delay value expires. A delayed state results from the transaction waiting for its I/O or CPU service to be completed. A blocked state is the result of the transaction being blocked, waiting for its turn on the I/O or CPU service queue.

A transaction involved in a deadlock will be aborted and restarted. After a transaction performs all of its read and write requests successfully, it will commit and release all the locks. The release of locks in our case is done by subtracting 1 from the number of active transactions in the system. This changes the probability of conflict, since the number of active transactions is a parameter in the equation.

5.3. Results of the Centralized System Simulation

Although our focus is not on a centralized database system, we discuss its performance behavior, because it is a main component of a distributed database system. The remainder of this section presents and analyzes the centralized database system experiments.

5.3.1. Performance Metrics

The primary performance metric we used throughout our study is **transaction throughput**, the number of transactions that finish per second. The throughput is calculated for a number of different MPL, each run for a period of 200 simulated seconds to calculate its throughput. This duration is large enough to give more than 95 percent

confidence in the throughput values as discussed at Figure 6.10. The confidence interval will be determined for the results of the distributed system.

Response time, given in seconds, is also used as a performance metric. It is the time interval between the submission of the transaction to the Scheduler and the time the transaction commits. This metric is useful to measure the system's convenience to the user.

Another performance metric used is the **blocking ratio**. It is the measure of the number of times a transaction blocks on a lock conflict in its lifetime per commit. Average Blocking ratio is computed as the ratio of the number of transaction blocking events to the number of transaction commits. This metric indicates data contention and its effects on lock acquisition.

5.3.2. PARAMETER SETTINGS

The parameters of the model reflect the state of the database, and give it a particular flavor. They are important to the simulation model, since they can make it close to or far from reality depending on the selected values. If, for example, a study were based on a very small or infinitely large database, then the study would be unrealistic and its results not representative of real systems. Another point to be considered is the logical relation between different parameters; they must be considered as a unit, and not in isolation. For example, the CPU access time might be set to 30 ms, and the disk access time to 15 ms; however, these values are not realistic because the disk cannot be 2 times faster than the CPU on most current systems. Therefore, choosing the model parameters and their values is of great importance to any performance study; the results of the simulation depend heavily on them. Table 5.2 shows the model parameters as well as their meanings: these parameters are used to simulate a centralized database system.

The values of the parameters used in our simulation are given in Table 5.3. These values are similar to those used in [AGRA87, THOM91, RAHM93], and are not specific to a particular database and do not represent extremes. As an example, a database of size 4096 represents an average database size. This applies also to the other parameters used in the simulation. Therefore, the parameters values we use do not represent a specific application. Also these values were tuned according to extensive tests on the program

behavior for different values for each parameter. As an example we varied the transaction size for different runs and chose the size that has the most representative performance.

Some of the parameter values are modified in different simulation runs; these new values are indicated whenever they are changed. The MPL varies between MinMpl and the MaxMpl; the former is 10 and the latter is 300. The MPL indicates the number of users or terminals requiring transactions on the database.

The MaxMpl is the maximum MPL that our system is going to reach. Throughput and other performance metrics are calculated for each MPL between MaxMpl and MinMpl, which is incremented in steps of 10. ThinkTime is the mean time of delay before a new transaction starts executing, due to processing or to a delay by the Scheduler. TrMin is the minimum size of a transaction; TrMax is the maximum size of a transaction.

Parameter	Meaning
Items	Number of items in the database
TrMin	The minimum size of a transaction's read-write set.
TrMax	The maximum size of a transaction's read-write set.
MaxMpl	The maximum MPL used in the simulation.
MinMpl	The minimum MPL used.
Disk	The number of disks
CPU	The number of CPUs
ThinkTime	The think time before a transaction starts.
DiskAccess	The disk access time in milliseconds.
CpuAccess	The CPU access time in milliseconds.

Table 5.2. Centralized DBMS Simulation Parameter Meanings

5.3.3. Simulation General Information

We use the SRC Modula-3 [© 1990 Digital Equipment Corporation], which provides concurrency programming facilities by the Thread library interface. We did not have any major problem using Modula-3 to implement our simulation. The only minor problem was

when we moved to the distributed database system. Here, the larger number of threads used a lot of main memory, obliging us to reduce the MaxMpl from 300 to 200 per site. The simulation was performed on a DEC-ALPHA under OSF/1. Each run took an average of 6 hours of real CPU time, simulating 200 seconds per MPL and a MaxMpl of 300 in the centralized system. The same CPU time was needed for the distributed system for MaxMpl of 200.

Parameter	**Default Value**
Items	4096 items
TrMin	7 (minimum)
TrMax	12 (maximum)
MaxMpl	300
MinMpl	10
MaxDisk	15 units
MaxCpu	7 units
ThinkTime	100 ms (milliseconds)
DiskAccess	35 ms
CpuAccess	15 ms

Table 5.3. Centralized DBMS Simulation Parameter Values

5.3.4. Simulation Results

Our experiments concerning the centralized database system cover three types of transactions: (1) a fixed-size transaction, of size equal to TrMax; (2) a variable size transaction, of size chosen randomly between 1 and TrMax; (3) and a variable size transaction of size chosen randomly between TrMin and TrMax. We refer to these types as fixed-size-transaction, variable-size1-transaction and variable-size2-transaction respectively. We have done three experiments where we varied the number of resources: in the first we had infinite resources; in the second 15 disks and 7 CPUs; finally in the third one we restricted the number of resources to 5 disks and 2 CPUs. These experiments were meant to show the behavior of a central system.

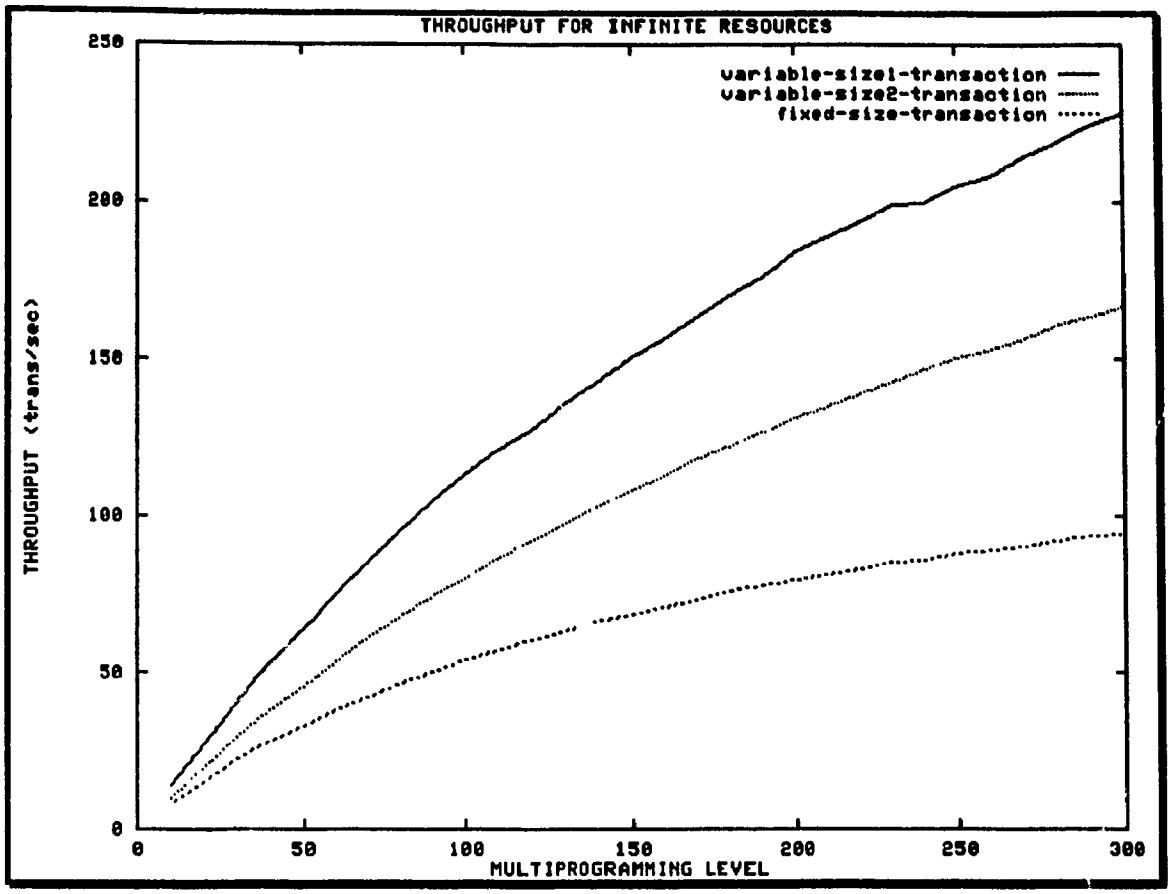

Figure 5.1. Throughput with infinite resources for the centralized system

Experiment 1: in this experiment we had infinite resources. This means that the transactions do not encounter any resource conflicts. The only conflicts are due to lock conflicts and deadlocks. We will show graphs based on the chosen performance metrics: throughput, lock conflict and response time.

As we can see in Figure 5.1, the throughput for the variable-size1-transaction reaches 225 transactions per second at the highest MPL level, and is increasing. This shows that the data contention is low for a variable-size1-transaction, while the variable-size2-transaction, whose size varies between 7 and 12, has less throughput, around 170 transactions per second. The third type of transaction, the fixed-size-transaction, has the lowest throughput results, of nearly 90 transactions per second. It is clear that the size of the transaction, requiring more locks and more resources, reflects its time to complete.

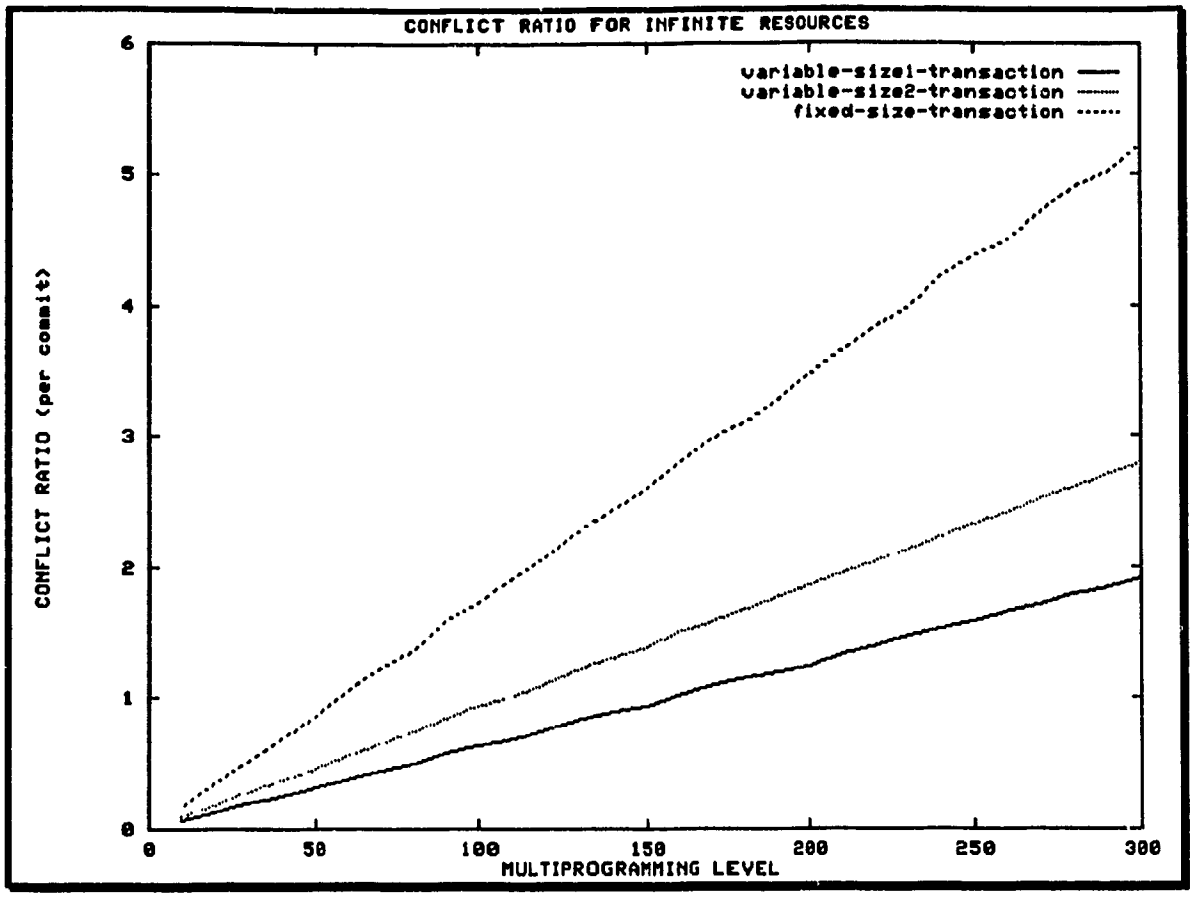

Figure 5.2. Conflict ratio with infinite resources for the centralized system

The more lock conflicts the transaction has, the longer time it takes to execute and finish. We can see in figure 5.2 that the maximum number of lock conflicts per transaction commit for the variable-size1-transaction is approximately equal to 2 conflicts per transaction commit at the highest MPL. For the variable-size2-transaction it reaches 2.8 conflicts. The worst case remains the fixed-size-transaction, where it reaches 5.2 conflicts per committed transaction.

Examining Figure 5.3, we see that for infinite resources the response time is relatively low. This is because a transaction does not need to wait in a queue for its I/O or CPU service. The only delay is the lock conflicts, which will make a transaction wait if its lock cannot be granted. Again, the variable-size1-transaction outperforms the other two types of transactions. The results of the response time in seconds are shown in Figure 5.3.

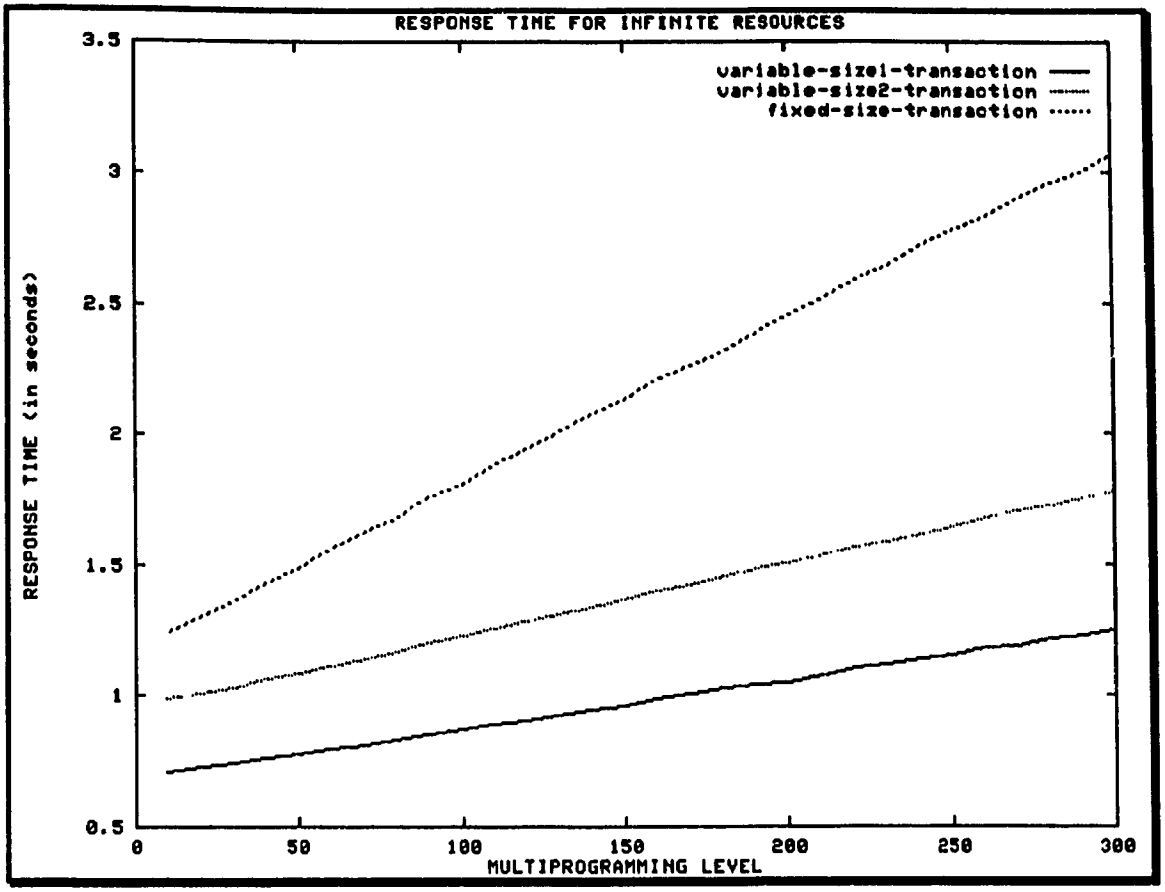

Figure 5.3. Response Time with infinite resources for the centralized system

Experiment 2: Here the number of resources was limited to 15 disks and 7 CPU's in the centralized site. We chose these figures because if we give more resources, the system performance will be close to the infinite resource assumption. These figures reflect a real system more closely than an infinite resources assumption.

We can see in Figure 5.4 that the throughput keeps on increasing up to MPL 50 for the three types of transactions, and then a plateau is reached which indicates that by increasing the MPL the throughput will not increase; however, the response time continues to degrade in Figure 5.6. We notice also that the difference in terms of throughput between the fixed-size-transaction and the variable-size2-transaction has become less than that in the infinite resource case. This reduction is due to the increase in CPU and Disk resource contention.

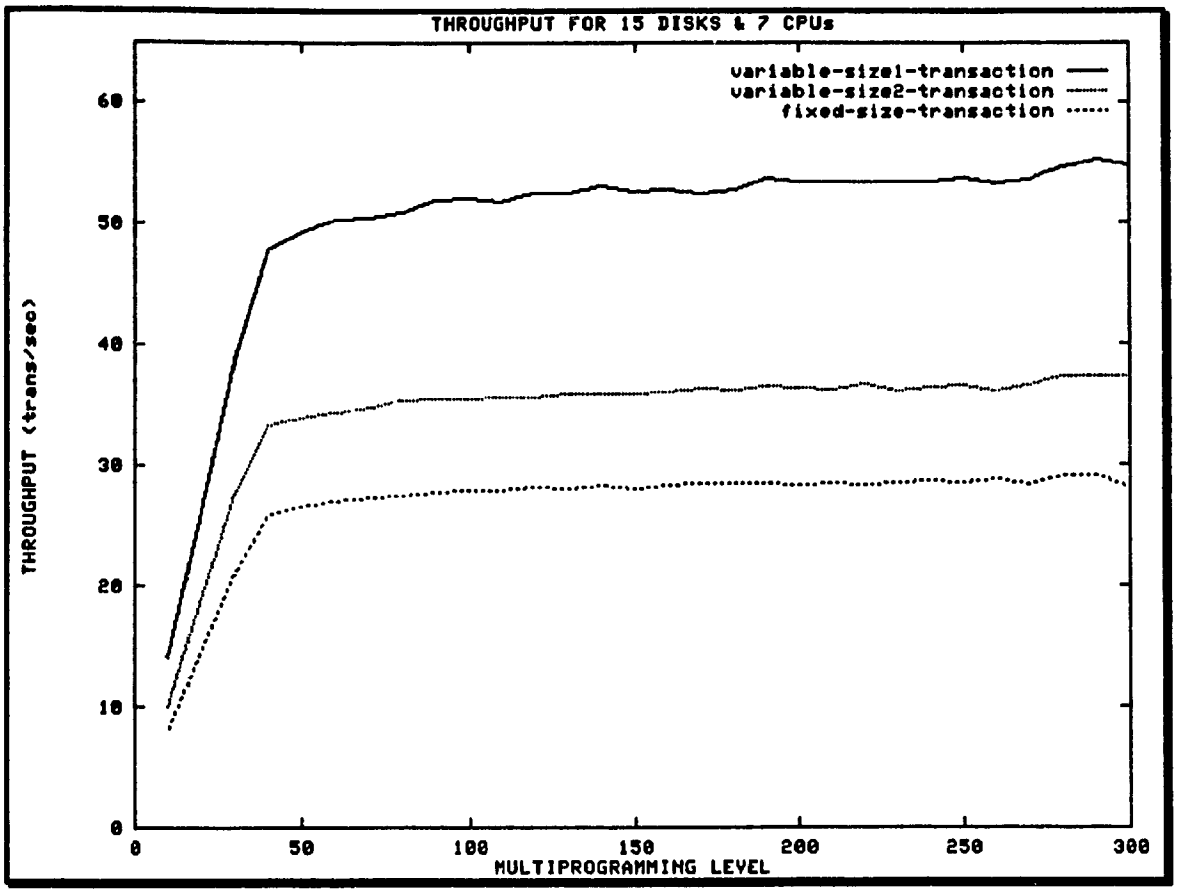

Figure 5.4. Throughput with 15 Disks and 7 CPUs for the centralized system

However, the conflict ratio results are the same as with the infinite resources! To explain this, we recall that the conflict ratio represents only lock conflicts and not resource conflicts. Hence the conflict ratios are independent of resource conflicts. The throughput decrease is caused by the resource conflict due to decreasing the number of resources. The graphs in Figure 5.5 are very similar to those in Figure 5.2. We deduce that an increase in both lock and resource conflicts results in decreasing the throughput of the system. It is obvious that resources are a big bottleneck to performance, since they are limited and resource conflicts occur more frequently than lock conflicts.

The response time plays a big role in determining the behavior of the system where a plateau is reached; increasing the MPL does not increase the throughput of the system. In this experiment, the throughput results do not show any difference of performance between one MPL and another. This difference can be seen in the response time results. In the worst case, an infinitely high response time is similar to thrashing; both ratios of active transactions to completed or committing transactions are high.

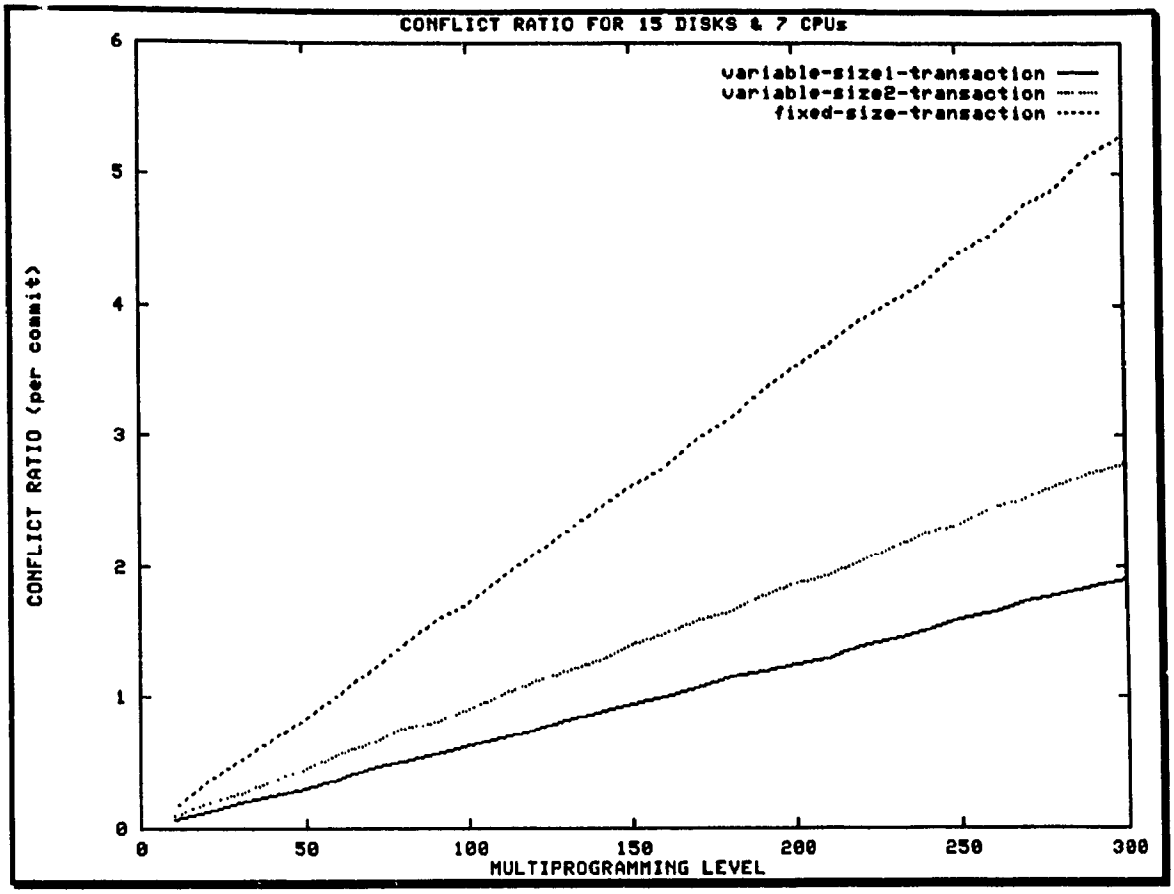

Figure 5.5. Conflict ratio with 15 Disks and 7 CPUs for the centralized system

In Figure 5.6 we see clearly that the response time increases constantly as we increase the MPL beyond 50. This is the effect of reaching a plateau, as shown in Figure 5.4. The response time in the case of fixed-size-transaction reaches 10.2 sec; with variable-size2-transaction it reaches 8 sec; while with variable-size1 it is around 5.5 sec. These results are much higher than corresponding values in Figure 5.3 (3, 1.6 and 1.3 sec) for the three transaction types respectively.

Experiment 3: This experiment uses 5 Disks and 2 CPUs. The results show further degradation of performance. We omit the results of the conflict ratio, since they are nearly the same as in the previous two sections, for the reasons given in Experiment 2. The results of throughput and response time are given in Figures 5.7 and 5.8 respectively.

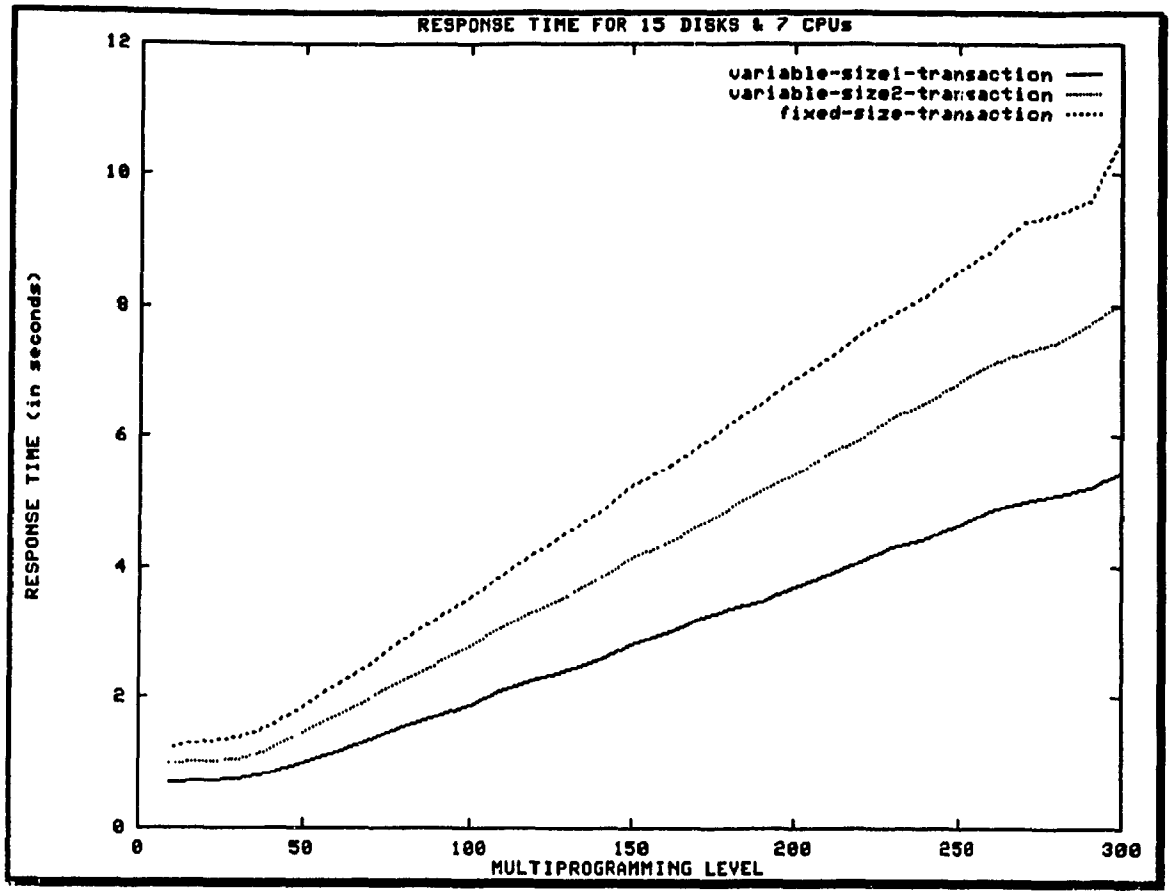

Figure 5.6. Response Time with 15 Disks and 5 CPUs for the centralized system

We can predict that the performance with this number of resources will degrade dramatically. It is easy to notice the influence on throughput due of the lack of resources in Figure 5.7. The plateau is reached at a level lower than in Experiment 2, and the level of throughput remains constant up till MPL 300. For the fixed-size-transaction it does not exceed 8.2 transactions per second, for variable-size2-transaction it is 11 transactions, and for variable-size1-transaction it is 16 transactions per second. These results indicate that for 5 Disks and 2 CPU's, extending the MPL above 30 does not improve throughput. On the other hand, performance will degrade with respect to response time, which will become higher as shown in Figure 5.8.

The response time is affected seriously in this experiment by low throughput and high resource conflict. We can see in Figure 5.8 that it jumps above 30, 25 and 15 sec for the fixed-size-transaction, variable-size1-transaction and variable-2-transaction respectively.

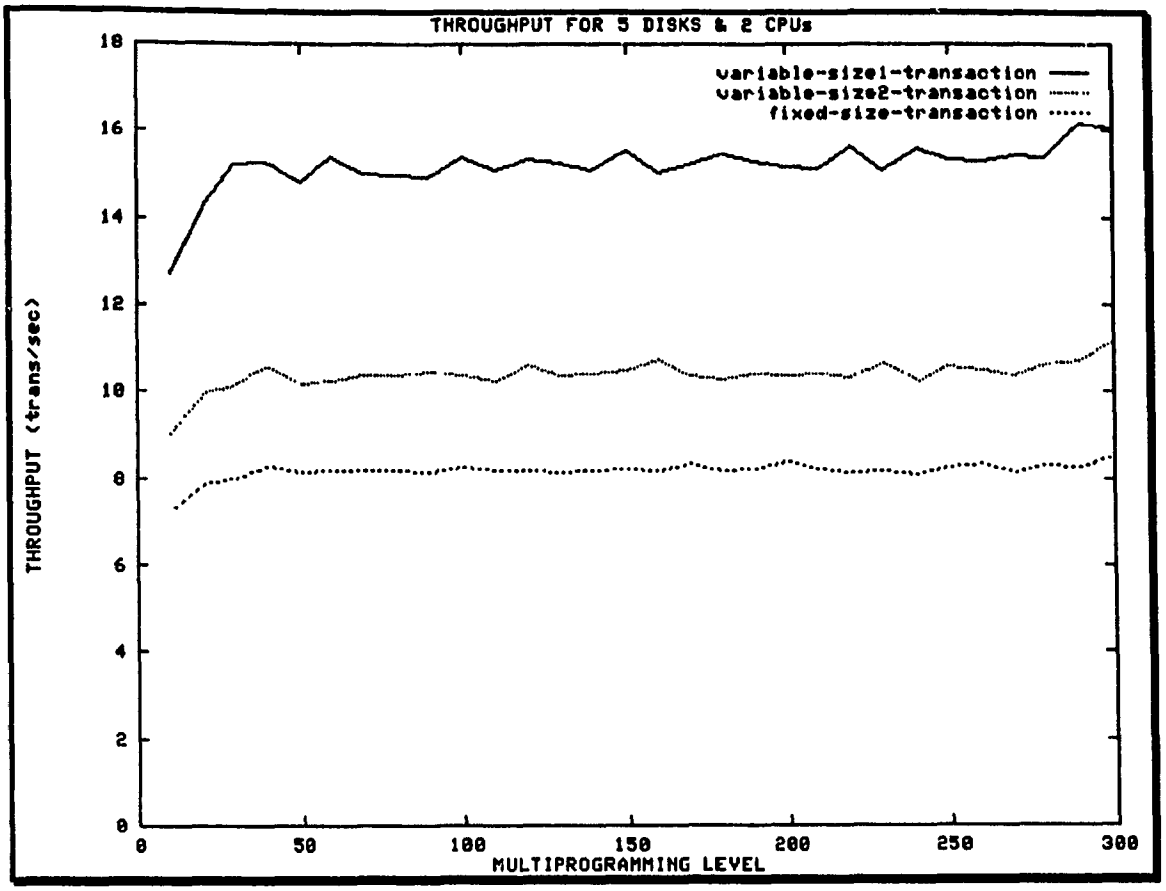

Figure 5.7. Throughput with 5 Disks and 2 CPU's for the centralized system

Conclusions from Experiments: We present the results of these experiments for the sake of showing the behavior of the centralized system, and for studying its performance in detail. We proceed in developing our distributed system, knowing the response of the underlying centralized system. Comparing our results to those in [AGRA87], we see that our results are pretty similar to the blocking concurrency control results. Finally, we conclude for the centralized study that resources are the bottleneck of a database system, in cases where requests are more that the resources can handle. In such a case, concurrency control enhancements do not improve the system performance much, while in the case where adequate resources are available, concurrency control enhancements can play a bigger role in improving system performance.

5.4. THE DISTRIBUTED DATABASE MODEL

Having examined the centralized database system, we discuss the components of the distributed database model, keeping in mind that all the components of the centralized system are preserved at each site, except where stated explicitly.

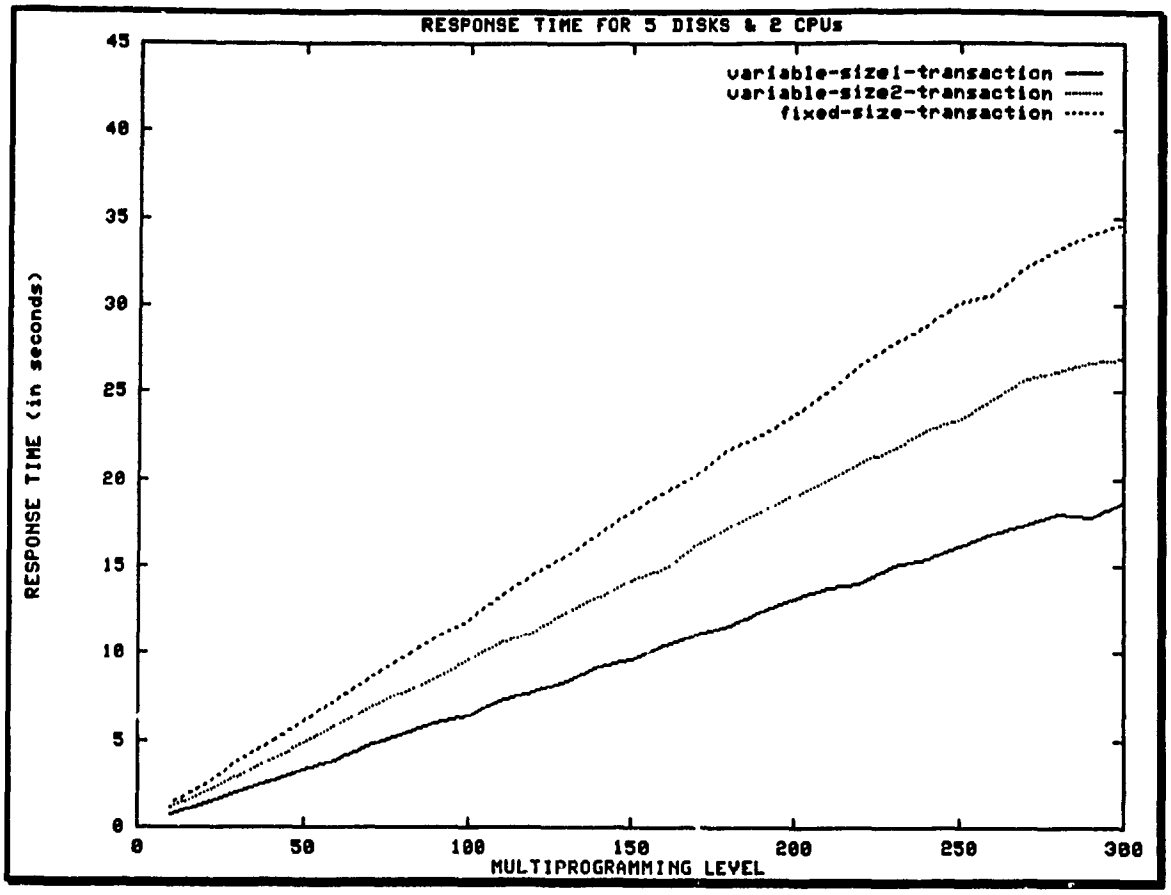

Figure 5.8. Response Time with 5 Disks and 2 CPU's for the centralized system

5.4.1. HARDWARE COMPONENTS

Our distributed database system is composed of 6 sites, each running a centralized database model, and a network that connects them. The sites communicate by sending messages through the network. A site may issue a global transaction, which requires more than one site to participate. Whenever a message is sent from site *a* to site *b* or *vice versa*, the load of the logical line (*a*,*b*) is incremented by the message size. Messages encounter a delay according to the load of the line they travel on. Timeouts are defined according to the maximum delay that a message may encounter before reaching its destination. Therefore, in the absence of failures, a participant or a coordinator will not Timeout.

Timeouts have minimum and maximum values, the minimum value at the lowest MPL, and the maximum value at the maximum MPL.

Site failures occur according to a fixed number of failures during the simulation run. For example, we simulate 3, 6 and 9 site failures; these are normally distributed over the period of the run. When a failure occurs at a site, the processing will stop. Hence all transactions running on that site will be suspended; they resume when the site recovers. Once a site fails, a process responsible for the recovery of that site is created. This process delays the failed site for a random period. Once the delay has expired, the site recovers and the transactions are resumed. The effect of site failures on local transactions is limited to the delay encountered by the site failure, since the recovery of these transactions can be accomplished locally. On the other hand, for a global transaction a site failure may cause the abort of the transaction; if the coordinator site fails before sending the decision message, and participants are in their uncertainty period, blocking will occur.

Communication failures are simulated according to the probability of a message being lost. Each time a message is sent, we check whether it will be lost according to the given probability; if the message is lost, then it will not reach its destination. The effects of communication failures are limited to global transactions. The worst effect is aborting the transaction; blocking is not possible due to the fact that participants can communicate with each other at the time of commit in all protocols.

5.4.2. SOFTWARE COMPONENTS

The distributed database system simulation is composed of a coordinator process, a participant process, and a global clock process. The synchronization between participants and their coordinator is done by one of the following Atomic Commitment protocols: B2PC, P2PC, and O2PC, which are discussed in detail in Chapter 4.

Once the **Coordinator Process** receives a global transaction from the TM of a site, it is responsible for splitting that transaction into subtransactions, and forwarding them to the appropriate sites. After delaying itself for a period of time, while the participants have some time to do the processing at their sites, it will start one of the three commit protocols used in this paper. Once the commit protocol is accomplished, the coordinator will terminate itself.

The **Participant Process** is responsible for receiving and following the instructions of its coordinator, according to the protocol used. The participant process at each site starts when the coordinator splits the global transactions and forwards them to the appropriate sites.

The **Global Clock Process** represents the progress of real time; hence its value is shared among all sites. It synchronizes all the actions involved in the ACP used. It is assumed, for simplicity, that the processing time is uniform on all sites, hence all CPU's have the same speed.

5.4.3. TRANSACTION MODEL

Two kinds of transactions are considered in our distributed system: transactions that are running locally, and global transactions that split into subtransactions. A subtransaction performs the same steps as a local transaction on its site; it acquires locks and performs its read and write operations. On the other hand, a subtransaction commitment or abort is controlled by its participant, which operates according to an ACP. Hence, the subtransaction will not commit until it receives a commit message through the ACP.

The size of a subtransaction is between 1/3 and 2/3 of the maximum transaction size. The percentage of local transactions in our system at each site is 40%; the rest are subtransactions of global transactions. The MPL is thus composed of 60% subtransactions of global transactions and of 40% local transactions. For example, for an MPL of 10 on each site, the number of local transactions is equal to 4, the number of subtransactions is less than or equal to 6. The reason for the upper limit of 6 global subtransactions is that it is not necessary to generate a subtransaction at all sites. Global transactions are issued from each site with uniform probability, and in turn issue between 2 and 6 subtransactions at random sites. The global MPL is controlled by the generation of global transactions; when a global transaction terminates, a new global transaction is issued. For local transactions we adopted the variable-size2-transaction type, used in the centralized database system.

5.4.4. PERFORMANCE METRICS

The performance metrics we used for the distributed system are:

- **Average local throughput** per second is the average number of transactions that have committed on each site, local transactions and subtransactions included. This throughput is calculated for the total MPL (local and global transactions). It is calculated by dividing the sum of all committed transactions and subtransactions on all sites by the number of sites.
- **Global transaction throughput** per second represents the number of global transactions committed per second for each global MPL.
- **Global transaction aborts** per second is the number of global transactions that have aborted per second for each global MPL.
- **Local transaction response time** represents the local transaction's average response time, excluding the subtransactions.
- **Global transaction response time** is measured from the time that the global transaction is generated on the sites, to the time that the coordinator terminates it.
- Number of **messages exchanged** per commit is the total number of messages exchanged divided by the number of global transactions that committed. It includes the messages of both global transactions that have aborted and those that have committed.
- Number of **compensating transactions** is the average number of compensating transactions issued at all sites per second. It is applied only to the O2PC protocol.

5.4.5. PARAMETER SETTINGS

Table 5.4 shows the parameters we used as default values for the simulation of a distributed database system. Any change in the values will be indicated. The site failure rates are indicated with the results, since they vary from experiment to experiment. The time that a site fails is 8-10 sec, including recovery time. We choose this time since simulating a longer time of failure requires extending our time of simulation, which in turn would require longer execution times. The Communication Failure gives the probability of failure for each message sent; this failure might be due to noise in the link, network partitioning, or a long delay.

Parameter	Default Value
Items	4096 items (each site)
TrMin	7 (minimum for local transactions)
TrMax	12 (maximum for local transactions)
MinMpl	10 (each site)
MaxMpl	200 (each site for both local and global)
MaxDisk	15 units (each site)
MaxCpu	7 units (each site)
Nsites	6 sites
Subt-Size	1/3 - 2/3 of TrMax
GlobalMpl	60% of Mpl
Timeout	1-4 seconds
Message-Delay	16 ms per 512 bytes
Message-Length	512 bytes
Site-Failure-Time	8-10 seconds
Communication-Failure	0.2 %

Table 5.4. DDBMS Simulation Parameter Values

5.5. Concluding Remarks

Finally, in this chapter we have described the characteristics and details of our implementation model, taking into consideration the whole system from the centralized database system up to the distributed database. We have shown the behavior and performance of the centralized database, independent of the distributed database. The next chapter presents the final simulation results of this study.

CHAPTER 6

SIMULATION RESULTS

In this chapter we present the results of the simulation experiments for a DDBMS. The experiments were conducted for three different Atomic Commitment Protocols, P2PC, B2PC and O2PC. The performance of these protocols in these experiments is determined in terms of their tolerance of failures in a Distributed Database Management System. The strategy we used was to measure the performance of these protocols for various failure rates. In the first set of experiments the performance was measured in the absence of failures; the second set of experiments dealt with communication failures; finally the last set of experiments were conducted in the presence of both communication and site failures.

6.1. CONFIDENCE INTERVALS

Confidence intervals [AGRA87] or error bars are the range of y abscissa values between different runs for the same values of x. Before starting our experiments we measured the confidence intervals between different runs with the same parameters, to make sure that results are independent of a particular run. Confidence intervals were measured in the absence of failures, since they affect confidence intervals by their unstable behavior. We did four runs with the absence of failures using B2PC, and two graphs were produced from the results showing the error bars. Figure 6.1 shows the global transaction throughput, and Figure 6.2 shows the average local transaction throughput.

The "GLOBAL MULTIPROGRAMMING LEVEL" in Figure 6.1 stands for the number of global transactions running in the system. The "LOCAL MULTIPROGRAMMING LEVEL" in Figure 6.2 indicates the total number of local transactions plus subtransactions running at each site. The local MPL may be less than the actual transactions running on each site, because a global transaction may involve between 2 and 6 subtransactions.

The error bars for the global transaction throughput vary between 0 and 4% (Figure 6.1), which makes the confidence intervals between 95 and 100%. Therefore, the possibility of error between two different runs is less than 4%, which is an acceptable range in this context [AGRA87]. The difference between two runs occurs because different runs of the

simulation do not have exactly the same sequence of transactions, due to the use of a random number generator.

For the average local transactions the error bars show a very tight range, which is hardly noticeable (Figure 6.2). This is due to the fact that this range involves the average of all the sites, which makes the error more tight than the global transaction one.

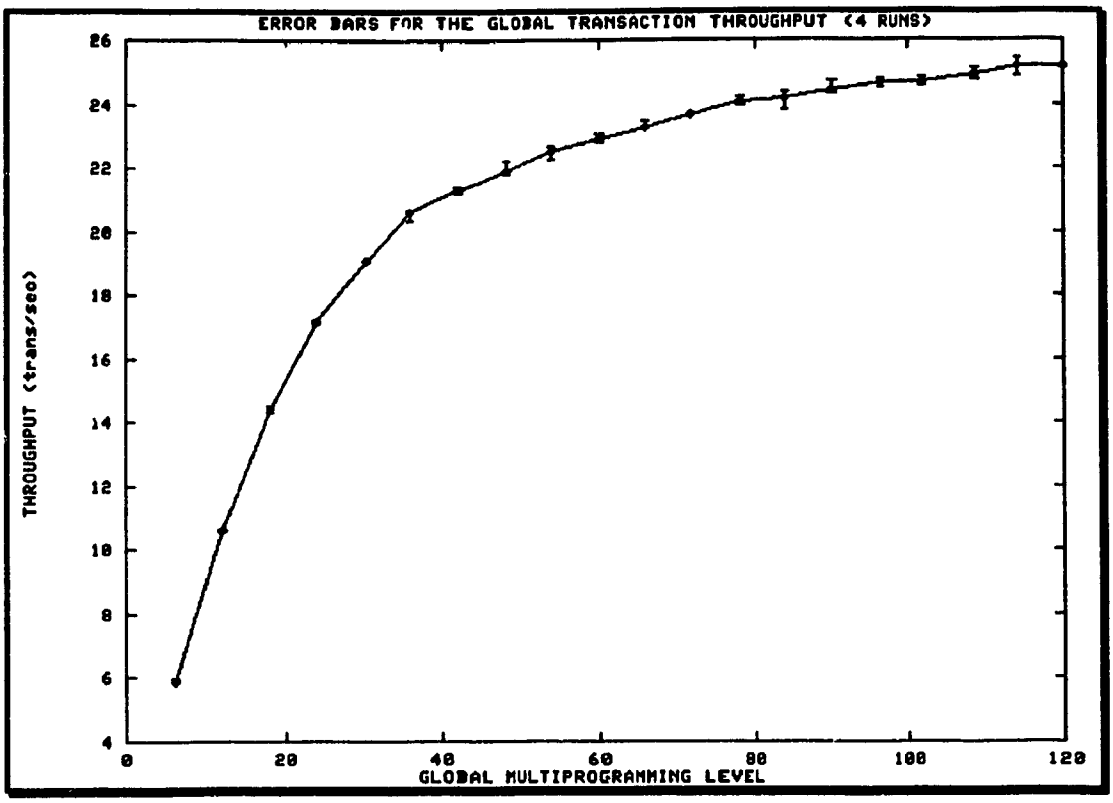

Figure 6.1. Error Bars for Global Transaction Throughput

6.2. Simulation General Information

We implemented a program that simulates the three different protocols used in our study. Each protocol is used as an input parameter inside the program, and requires an independent run. This entailed running the simulator once for each protocol using the same experimental parameters. Each run of a protocol produces a separate result file for each performance metric used in this study. Graphs were plotted from these result files using the GNUPLOT [© Thomas Williams and Colin Kelley, 1986-1993] software under UNIX.

6.2.1. SIMULATION OF COMPENSATING TRANSACTIONS

As described in Chapter 4, a participant in O2PC releases the locks before the coordinator's decision. This may result in the need to run compensating transactions on all committed sites, if the coordinator decides to abort the global transaction. A compensating transaction is one that undoes the effects of a committed transaction semantically. Unfortunately, we can't incorporate the cost of semantic compensations in the simulation; this is difficult to determine, because it involves semantic knowledge of the transactions. Hence, we limit the cost of a compensating transaction to the cost of issuing an additional transaction on its local site; this will decrease the local throughput. Thus the cost of a compensating transaction will mostly affect the average local throughput and not the global throughput; we ignore any economic or real cost of an optimistic commit.

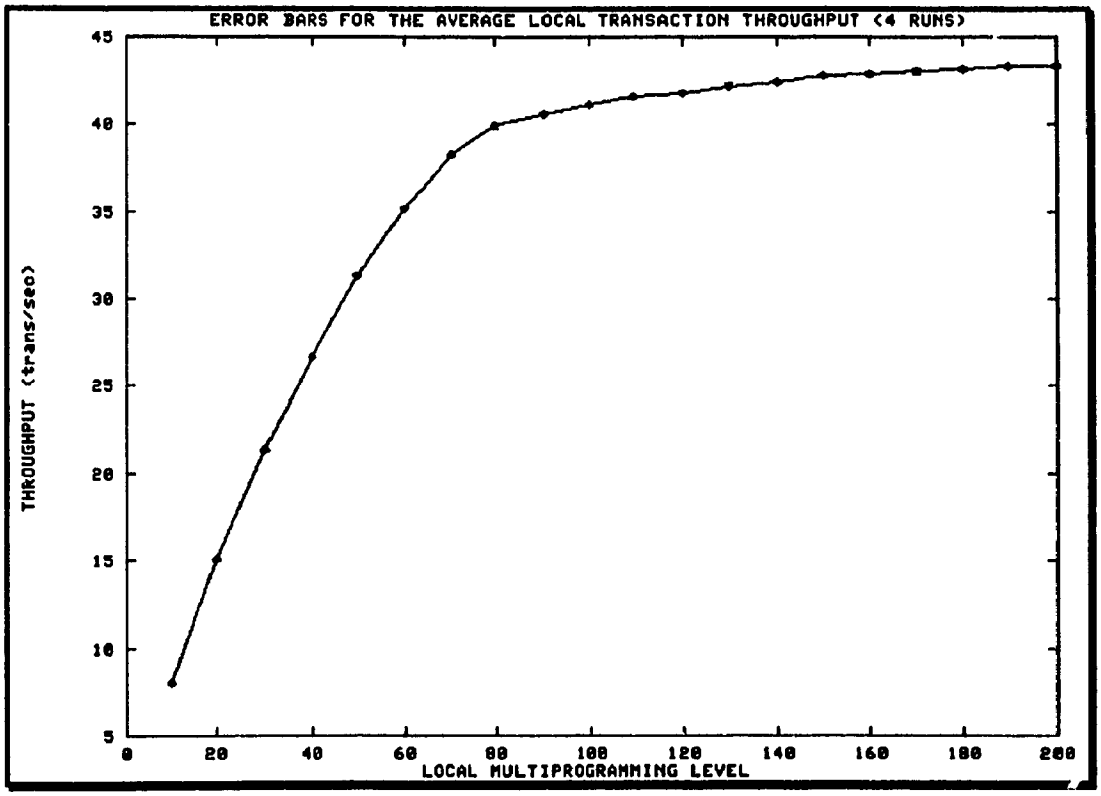

Figure 6.2. Error Bars for Average Local Transaction Throughput

6.3. SIMULATION RESULTS: NO FAILURES

The first set of experiments was conducted with no failures, except those due to deadlocks. The purpose of these experiments was to compare the performance of O2PC with that of B2PC in the absence of failures. In this way we have an idea about their relative performance due to deadlocks only, before we move to experiments involving failures. Three graphs were produced by these experiments: global throughput (Figure 6.3), average local throughput (Figure 6.4) and compensating transactions graphs (Figure 6.5).

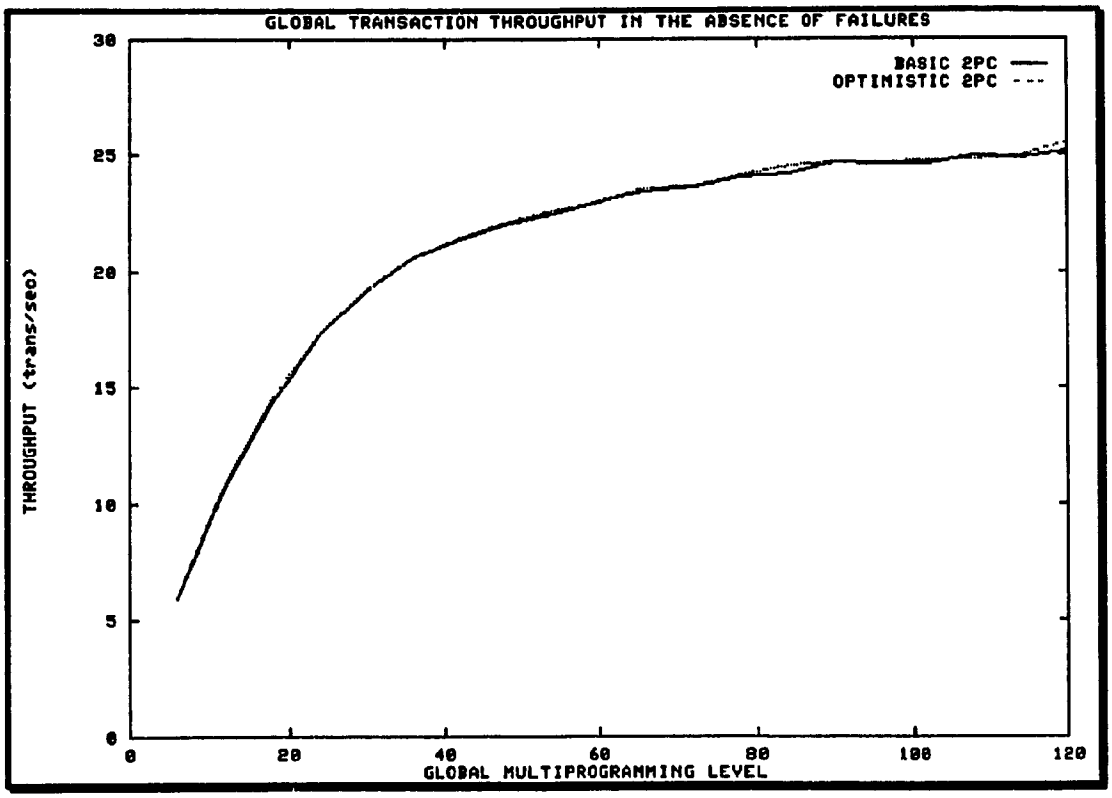

Figure 6.3. Global Transaction Throughput with no failures

For our database, the global throughput results show that the performance of O2PC is at best slightly better than the B2PC protocol, due to releasing the locks earlier. The results do not show a big difference, since in the absence of failures locks are not held long; therefore an early release of locks does not make much difference on the global throughput level.

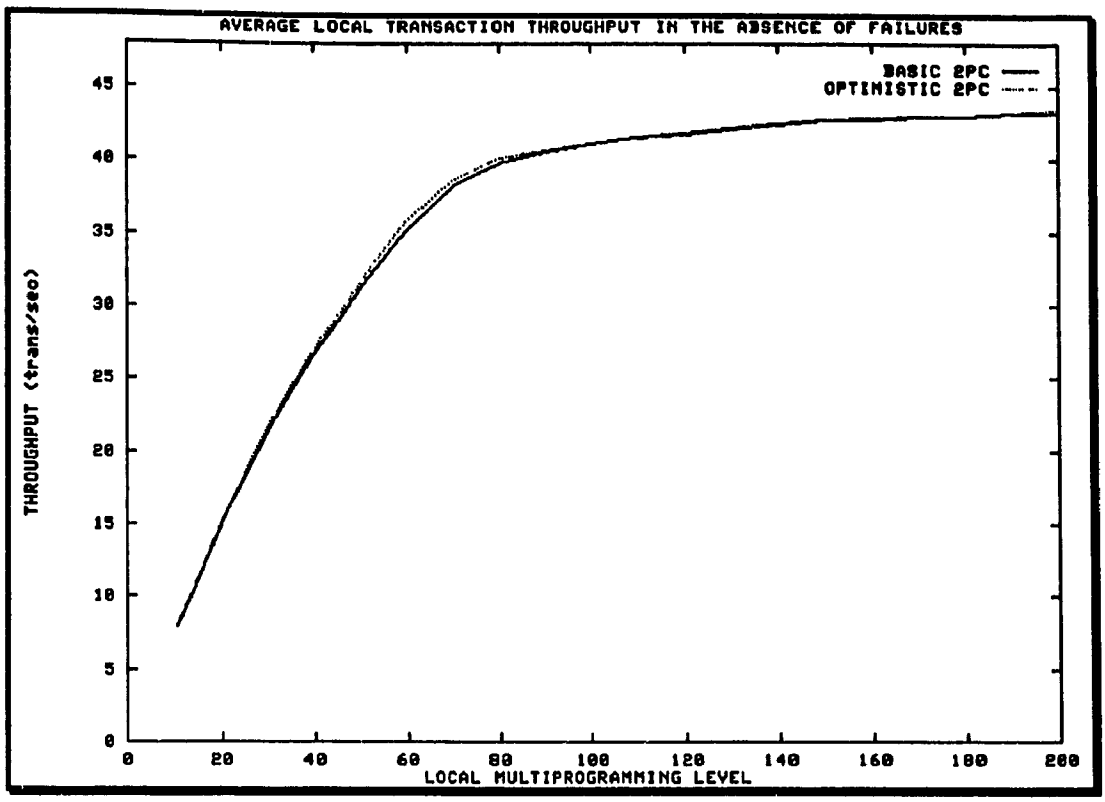

Figure 6.4. Average Local Transaction Throughput with no failures

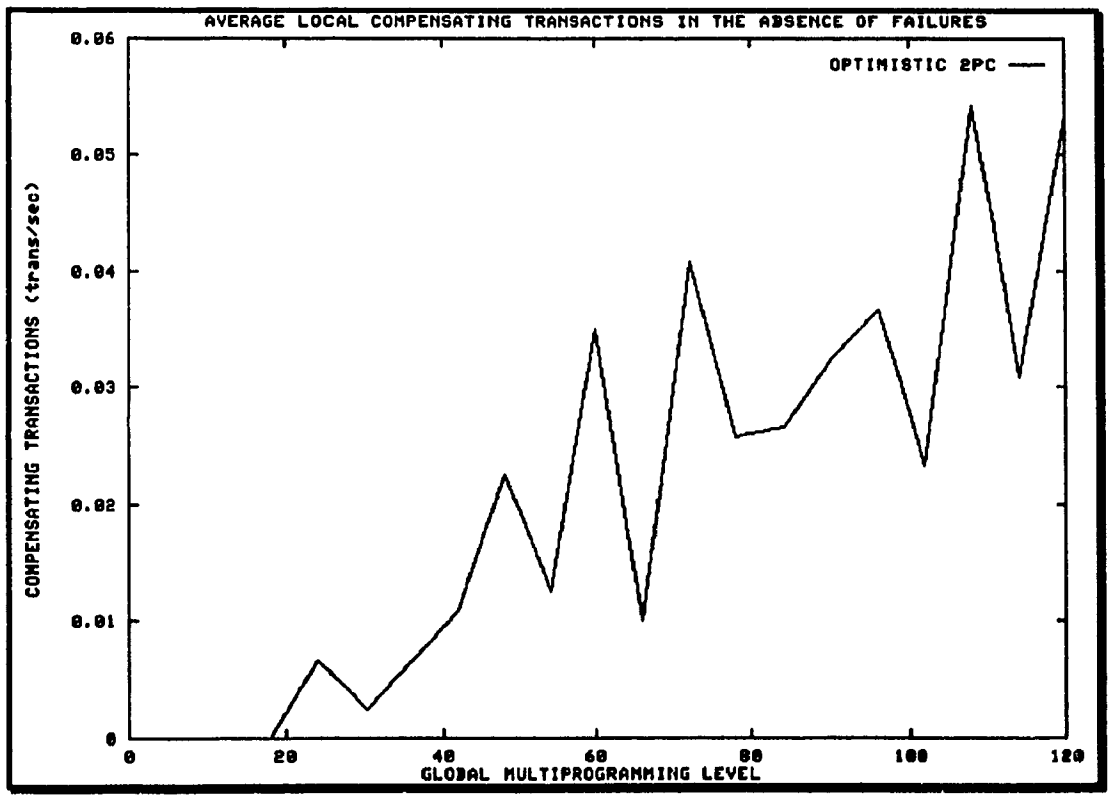

Figure 6.5. Average Compensating Transactions with no failures

The average local throughput results confirm the global results, and show that O2PC cannot overcome the overhead of its compensating transactions, even in the absence of failures. We can see in Figure 6.4 that O2PC performs slightly better between MPL 40 and 90. As data contention increases, performance drops to overlap that of B2PC, due to the compensating transactions issued because of failures due to deadlocks.

Figure 6.5 shows the number of compensating transactions per second; the maximum number of compensating transactions reached is 0.06 transactions on each site. These compensating transactions are issued due to a deadlock that a subtransaction encounters during its lifetime. Compensating transactions are required when at least one subtransaction of a global transaction is aborted due to a deadlock, while subtransactions of the same global transaction go ahead and commit and release their locks. Since a participant sends the abort message to the coordinator and not directly to other participants, the latter rely on a message (to abort) from the coordinator. In the meantime, some participants that have reached the wait-decision before others will release their locks. Therefore the abort message resulting from a deadlock at one or more participants will reach them after they have unilaterally committed. Hence, a deadlock at even one participant will lead to compensating transactions for the other sites.

6.4. Simulation Results: Communication Failures

For communication failures we developed two experiments; the first with medium failure rates, the second with higher failure rates. The rates of failures were 0.2 % and 1% per message sent respectively. These experiments provide performance results for all three protocols; the performance metrics measured are: global throughput, local throughput, number of messages exchanged, global response time, local response time, number of global aborts, and number of compensating transactions.

Experiment 1: The results of this experiment show clearly the difference between the three protocols in the presence of communication failures. A communication failure will result in an abort in the case of B2PC, but in the case of O2PC, in an abort and a number of compensating transactions. For P2PC, communication failures are mitigated by a second chance, a possibility of not aborting the transaction: this in turn results in improving the system performance.

The global throughput presented in Figure 6.6 shows that P2PC protocol is the best among the three protocols, due to not always having to abort a global transaction in the case of a transient communication failure. We see also that the graphs of B2PC and O2PC interleave, and exhibit better or worse throughput at different MPL. We conclude that the performance of the latter protocols is, on the average, comparable, and that P2PC outperforms them. The difference is around 12% improvement at the highest MPL in favor of P2PC.

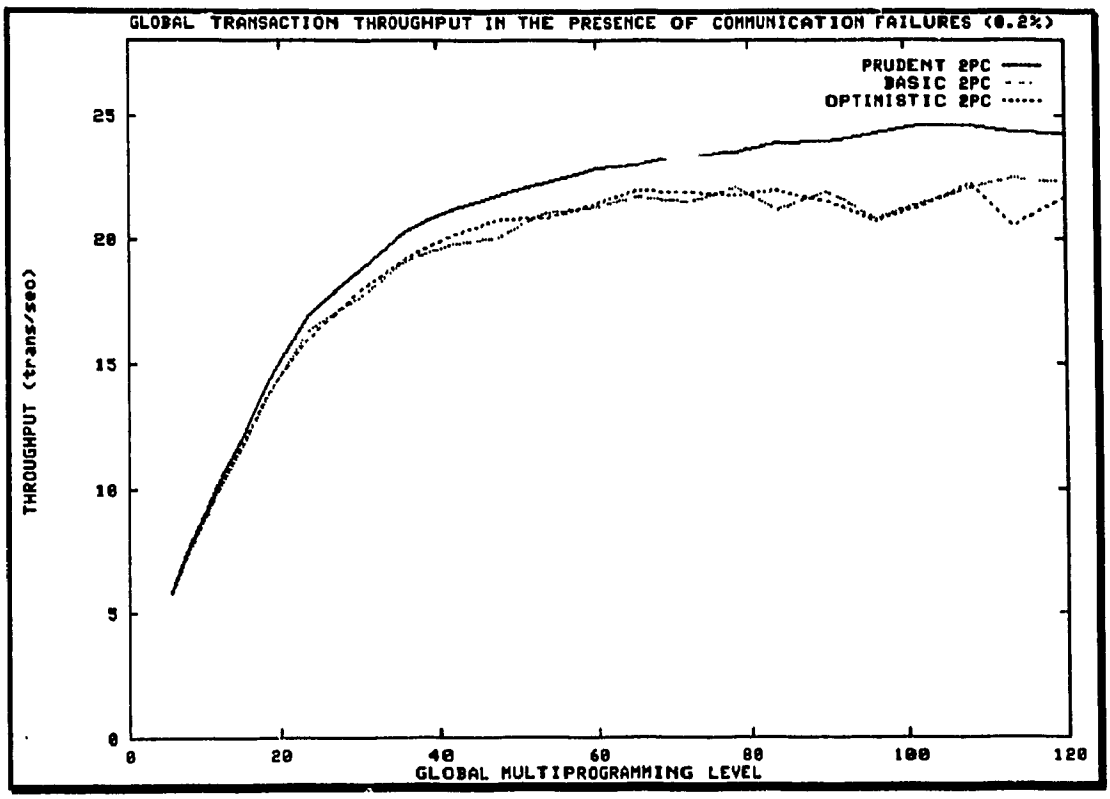

Figure 6.6. Global Transaction Throughput with Communication Failures (0.2%)

The local throughput results show also a difference of performance, since they embody a combination of both local and global transactions. An improvement in global transaction throughput will lead to improving the average local throughput, with no change in local transaction throughput. The performance of O2PC (Figure 6.7) and P2PC show slightly better throughput than B2PC for global MPL less than 80. Beyond an MPL of 80 P2PC reaches a plateau, but it outperforms the other ACPs. The performance for O2PC degrades and becomes lower than B2PC beyond MPL 140.

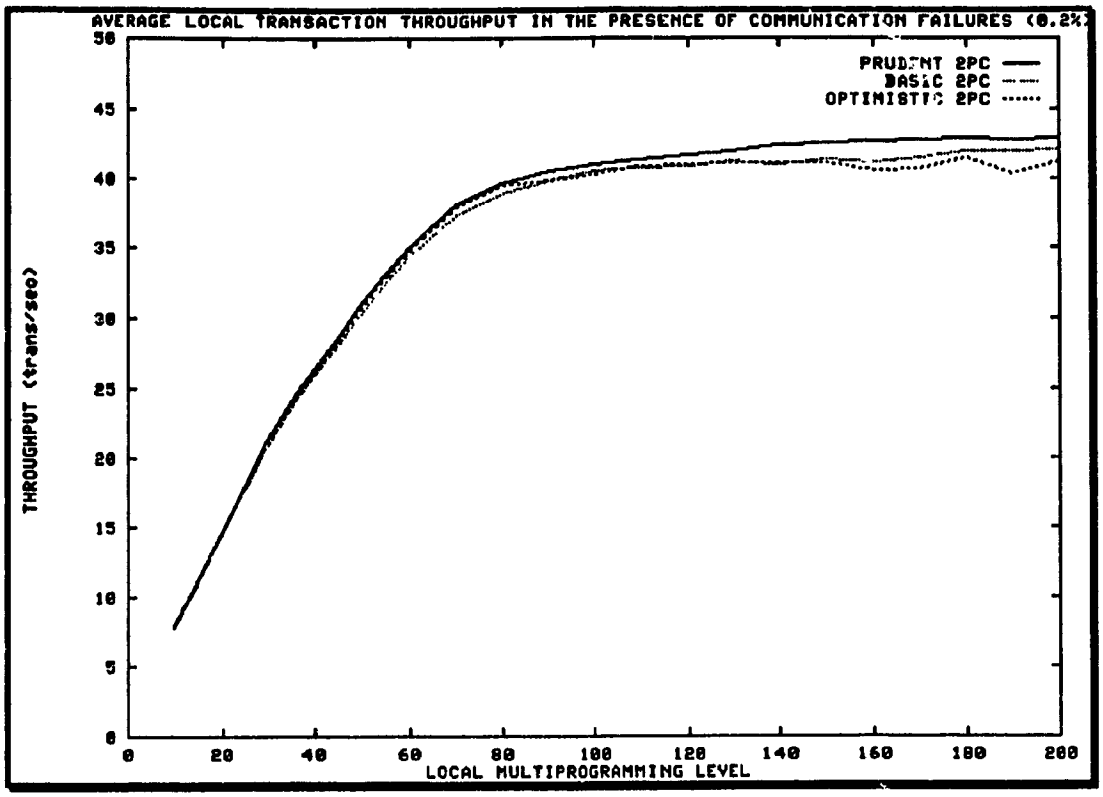

Figure 6.7. Average Local Transaction Throughput with Communication Failures (0.2%)

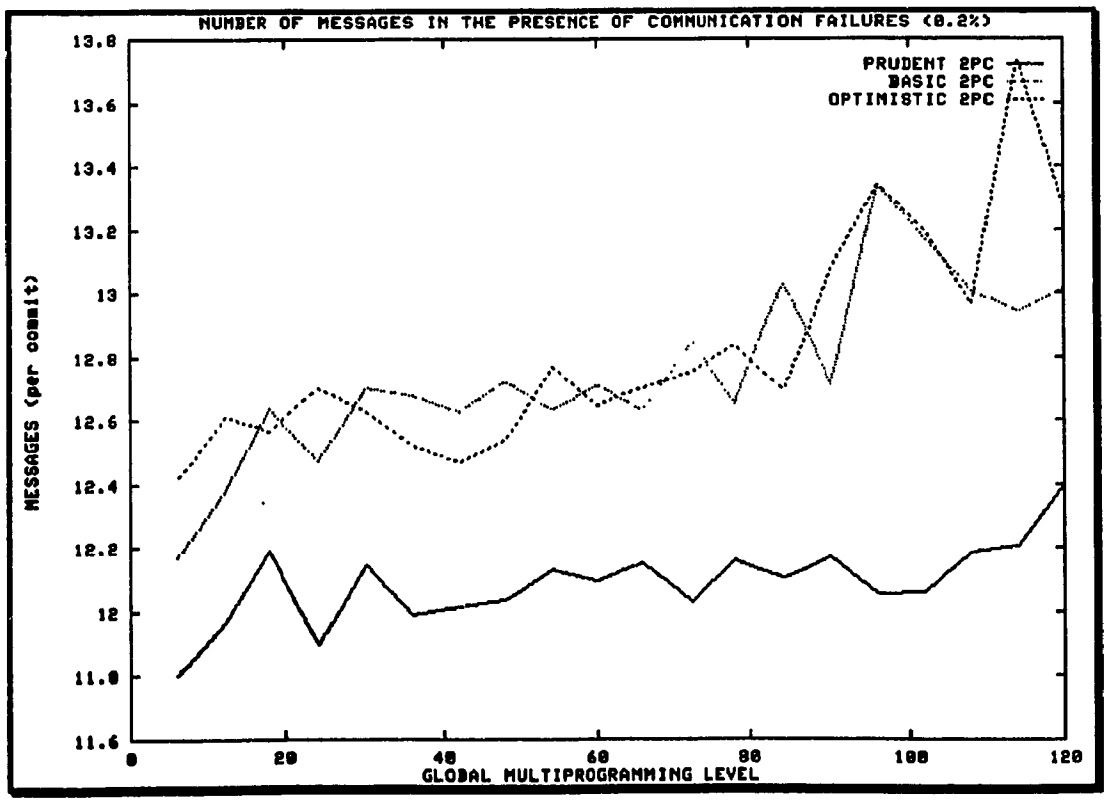

Figure 6.8. The Number of Messages with Communication Failure (0.2%)

The results of this experiment confirms the results of the first experiment, which was done with no failures; the performance of O2PC is no better than B2PC, and considerably poorer than P2PC.

Let us consider the number of messages exchanged. Figure 6.8 shows that P2PC has the lowest number of messages, while the results show that O2PC and B2PC have comparable numbers of messages. The maximum number of messages required is with O2PC (13.7 messages), and the lowest with P2PC (11.8 messages).

As discussed in chapter 4, P2PC is characterized by the most complex message exchange among the three protocols. However, the graph in Figure 6.8 show a decreased number of messages for it. The reason for this is as follows: in P2PC a global transaction, instead of aborting, uses additional messages to run to completion. In the other two protocols, a global transaction will abort after the maximum number of messages have been exchanged. Such action requires that the global transaction restart again; this restart adds to the number of messages. Hence, P2PC has the lowest number of total messages, and the lowest messages per transaction. Note, that the number of messages exchanged includes messages for transactions that abort and must restart.

The number of compensating transactions for O2PC protocol reaches 0.55 per second; this is shown in Figure 6.9. Comparing the compensating transaction graph with the O2PC abort graph in Figure 6.10, we notice that the two graphs are similar in shape, indicating that an abort leads to a compensating transaction in most cases. The numbers are not identical, since an abort of a global transaction does not necessarily require compensating transactions at all participating sites. We can see that aborts are lower in the case of P2PC, the maximum abort being 0.18 transactions per second, while in the case of O2PC, it reaches 1 transaction per second. The intersecting graphs of O2PC and B2PC show that they have the same performance and rate of failures; they do not overlap exactly, because failures are generated randomly and do not give the exact same numbers for both runs.

The local response time (Figure 6.11) indicates very similar results for three protocols. The difference is less than 0.1 second at the highest MPL. Hence, the only discriminating measure of performance is throughput, is best for P2PC, which outperforms the other two protocols. The average local response time for the three protocols is around 1.9 sec in this simulation at MPL 120, as shown in figure 6.11.

The global response time in Figure 6.12 also shows very close and comparable results. The maximum response time reached, at the highest MPL, is 5 seconds. This indicates that a global transaction needs around 2.5 times as much time as a local transaction. This is due to the overhead of sending messages and waiting for the coordinator's decision message. Again, the only measure of performance for global transaction is the global throughput graph. As we see from Figure 6.6, P2PC has better throughput than B2PC or O2PC.

Finally this experiment shows that P2PC outperforms B2PC and O2PC in terms of throughput and number of messages. This improvement, as shown in Figures 6.8, 6.11, and 6.12, is not at the expense of the response time, or the number of messages exchanged.

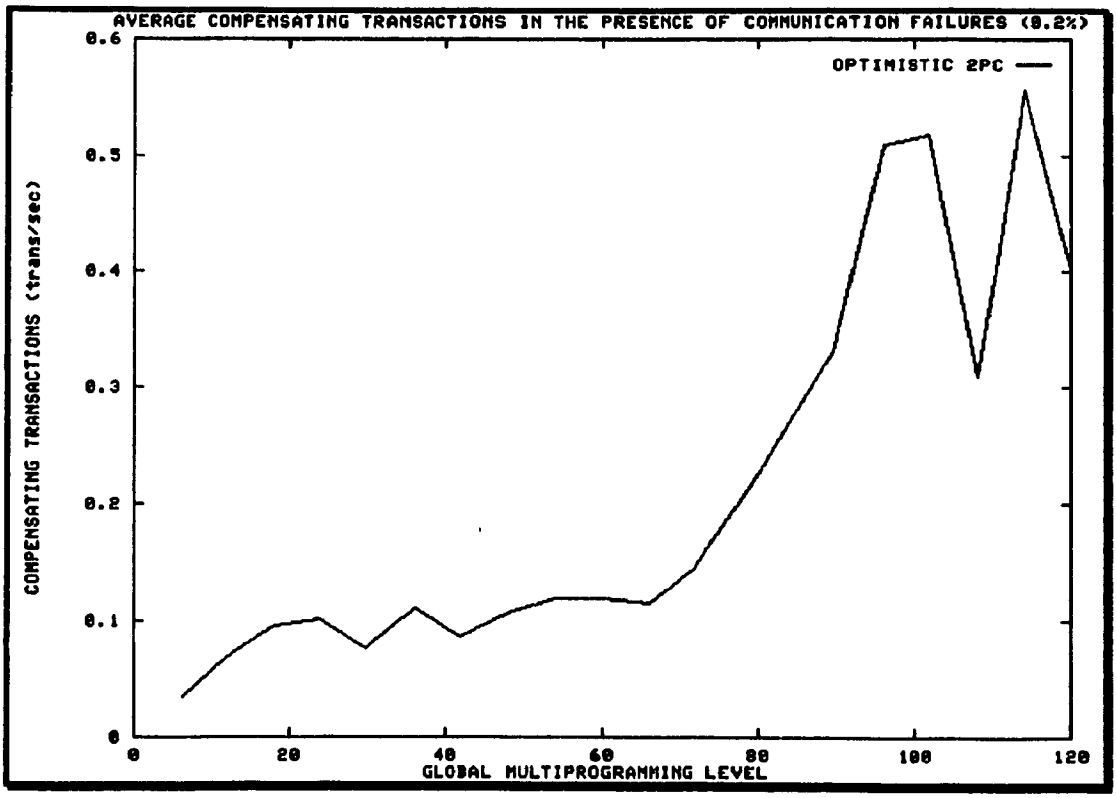

Figure 6.9. Average Compensating Transactions with Communication Failures (0.2%)

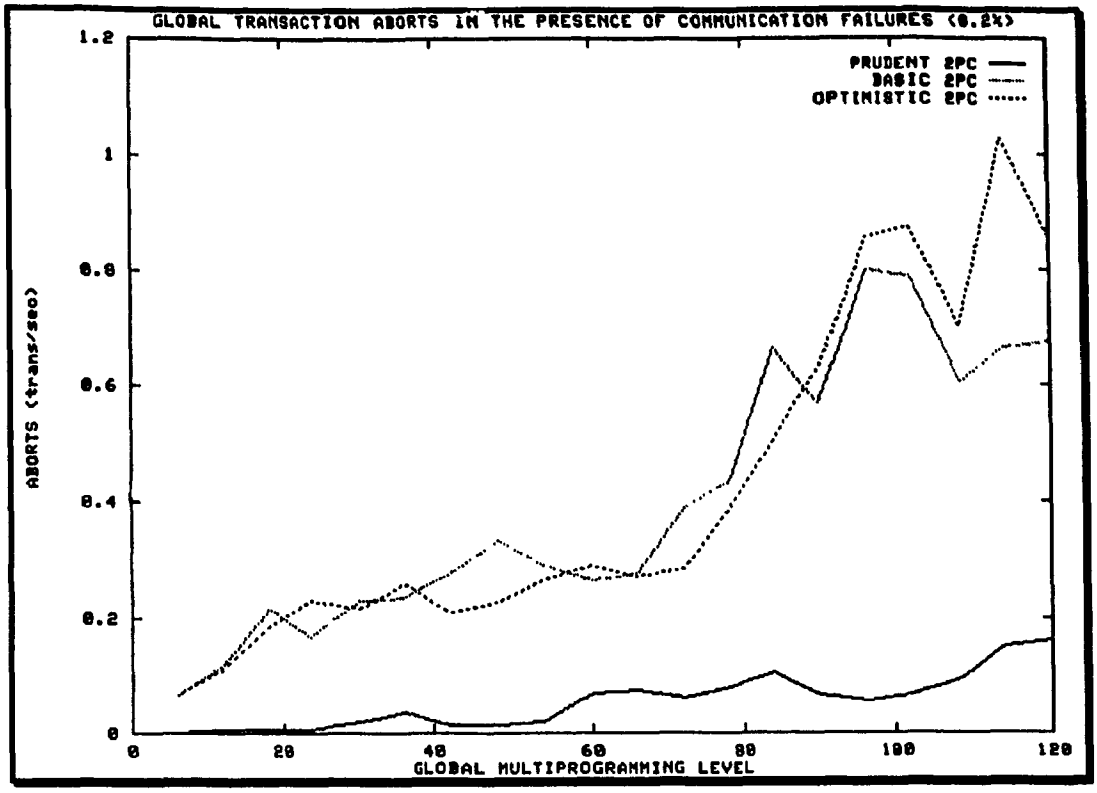

Figure 6.10. Global Transaction Aborts with Communication Failures (0.2%)

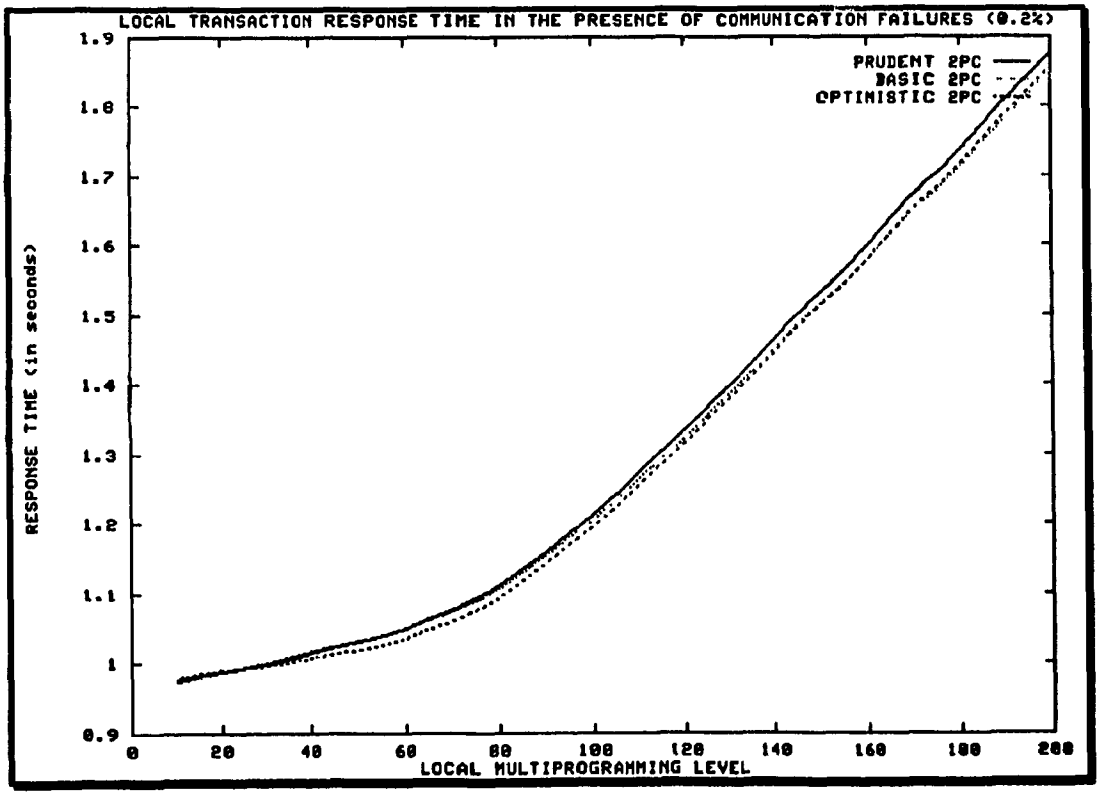

Figure 6.11. Local Transaction Response Time with Communication Failures (0.2%)

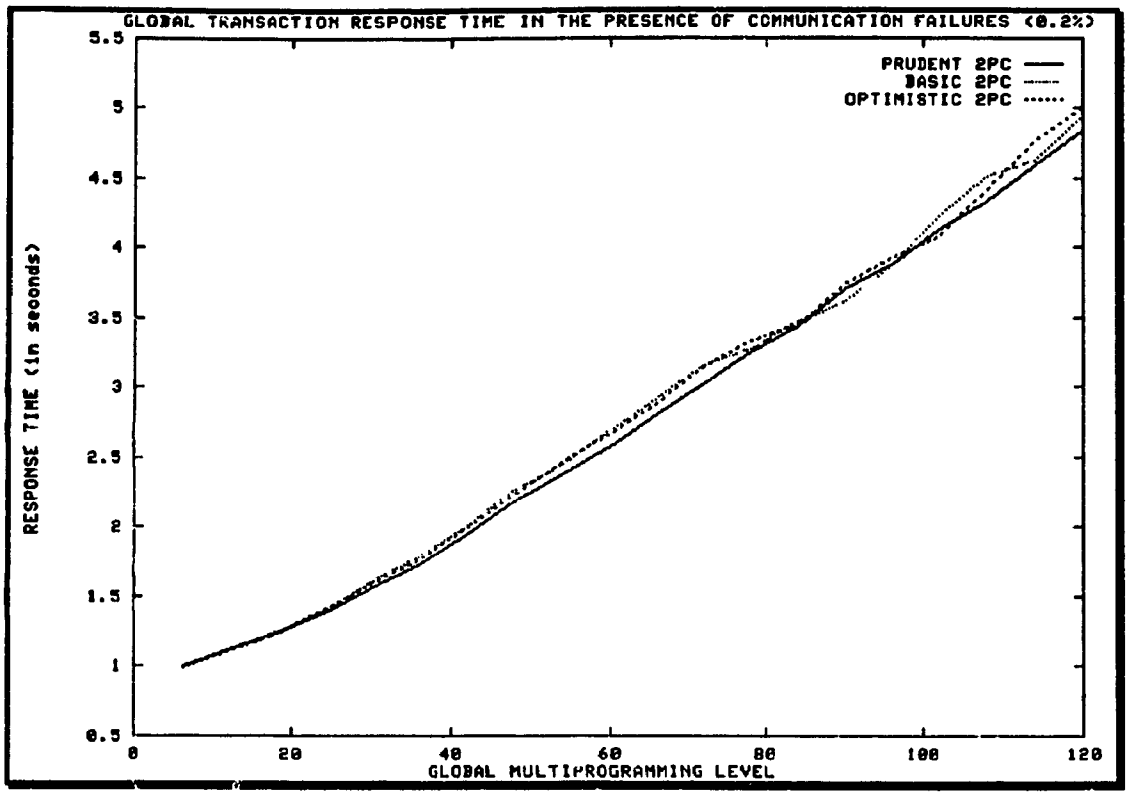

Figure 6.12. Global Transaction Response Time with Communication Failures (0.2%)

Experiment 2: In this experiment we increased the communication failures to 1% of messages sent. This compares the behavior of the three protocols under a higher communication failure rate, and verifies whether O2PC performance persists compared to B2PC, and whether the advantage of P2PC holds.

A difference in performance among the three protocols is illustrated by the global throughput result in Figure 6.13. P2PC reached 23 transactions per second at the highest MPL, while O2PC and B2PC reached 16 transactions per second and then dropped to 15 at MPL 200. The performance of O2PC was similar to B2PC, as in experiment 1.

The results for average local throughput in Figure 6.14 show what we expected for O2PC. O2PC degraded in performance with respect to B2PC, and did not overcome the overhead due to compensating transactions, which degrades around a global MPL of 100, due to the combination of deadlocks and the high rate of global aborts caused by communication failures. The difference in performance between P2PC and B2PC is larger than in the previous experiments, showing that P2PC performance does not degrade with higher communication failures.

Comparing the abort rates with the previous experiment, we notice from results in Figure 6.15 that P2PC has a slightly higher abort rate than in Figure 6.10; this indicates that not all communication failures aborts were prevented. A communication failure may result in an abort if it persists, which may happen when two consecutive messages fail. The first time a message fails, the transaction uses the second chance; the second failure has no backup, and the transaction aborts. The abort results show that B2PC and O2PC have a much higher rate than the previous experiment, reaching 2 global transactions failures per second. This high rate of abort is the cause of lower throughput and poorer performance of the system.

Finally, the number of messages in Figure 6.16 are higher than in the previous experiment (Figure 6.8). This is due to the higher abort rates, requiring restart of global transactions before a commit. The maximum number of messages is 17 per commit for both B2PC and O2PC, compared to 13.7 in Experiment 1. For P2PC it is between 12 and 13.2, compared to between 11.8 and 12.2 for Experiment 1.

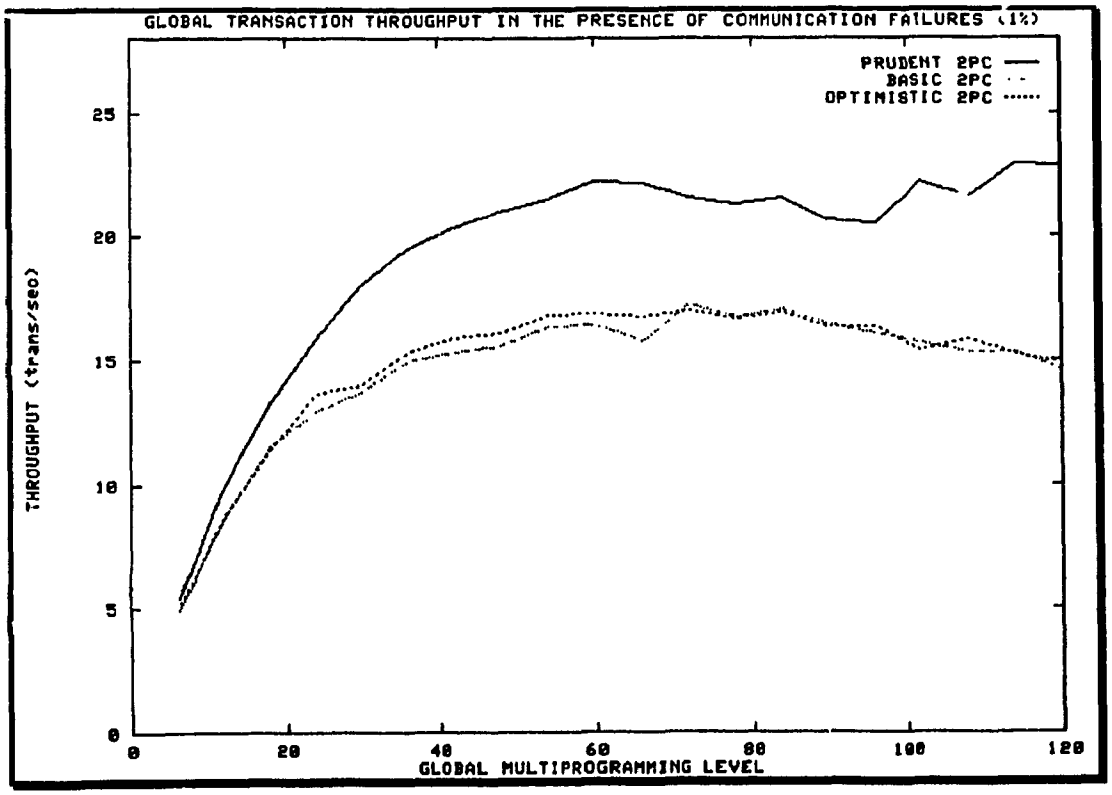

Figure 6.13. Global Transaction Throughput with Communication Failures (1%)

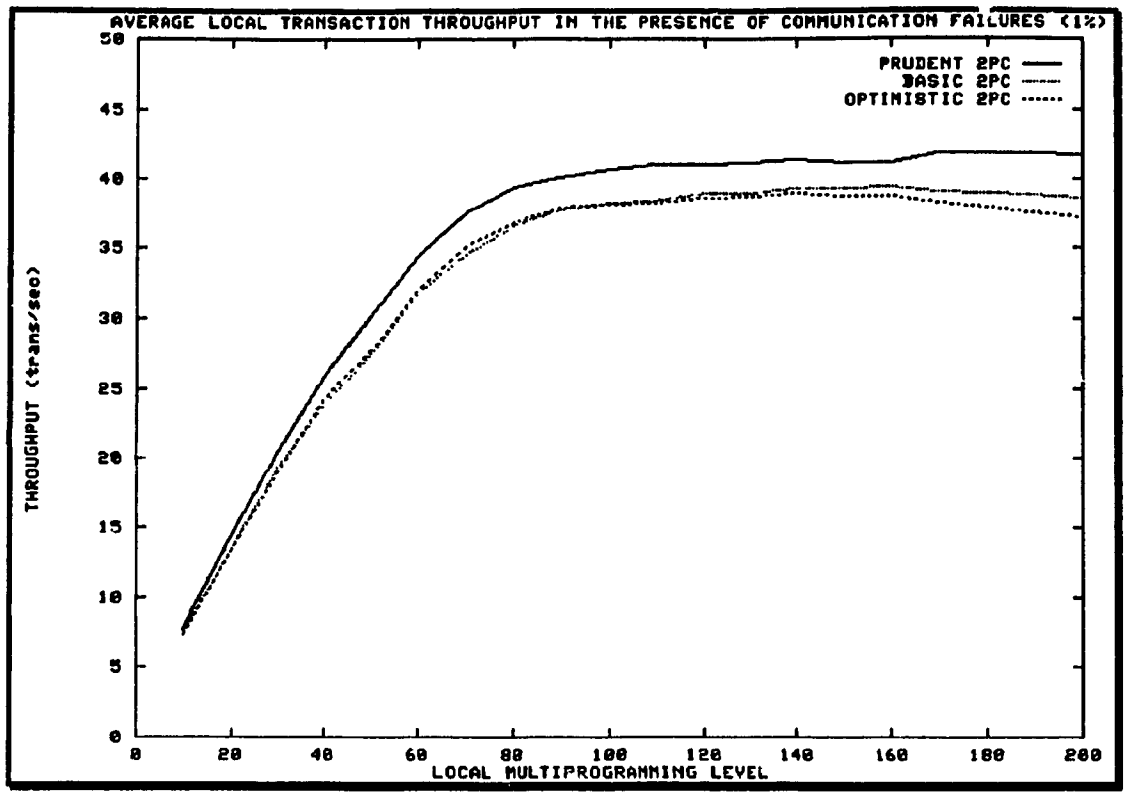

Figure 6.14. Average Local Transaction Throughput with Communication Failures (1%)

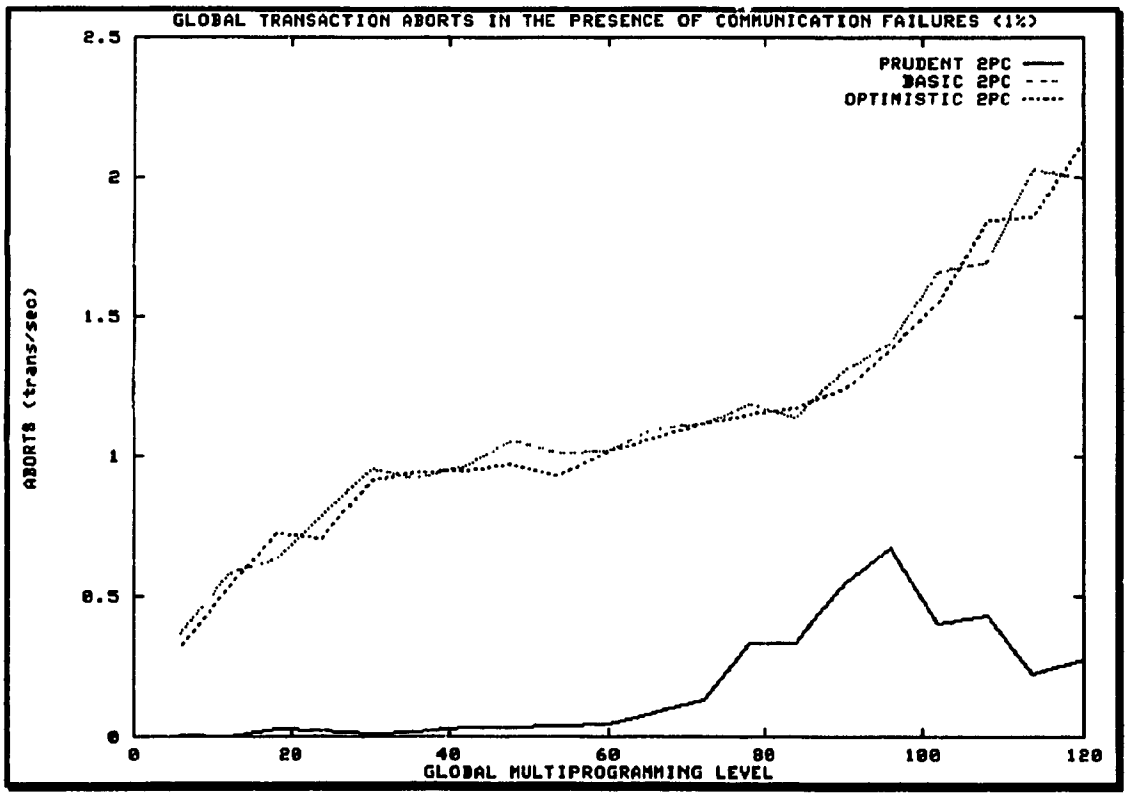

Figure 6.15. Global Transaction Aborts with Communication Failures (1%)

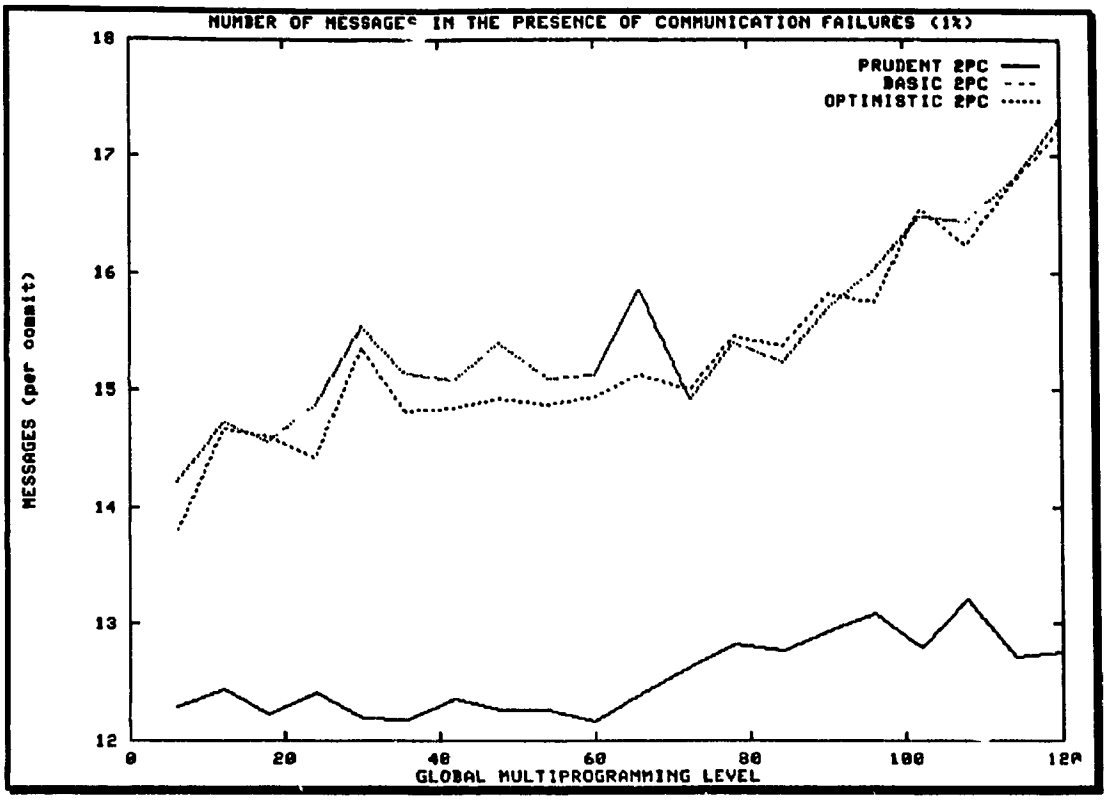

Figure 6.16. Number of Messages with Communication Failures (1%)

Our conclusion from these communication failure experiments is that for this system P2PC outperformed B2PC and O2PC in terms of throughput, number of messages exchanged and number of transaction aborts; it is equivalent in performance for response time. O2PC outperformed B2PC slightly in throughput for low MPL, and was equivalent in higher MPL for low communication failures in Experiment 1. For higher failure rates, as in Experiment 2 (1% of communication failures), O2PC performance degraded due to higher abort rates, necessitating increased numbers of compensating transactions.

6.5. Simulation Results: Site Failures

After evaluating the performance of our system in the presence of communication failures only, we move our attention to the effects of both site and communication failures. Site failures affect the throughput of the system much more than communication failures, since their effects extend over more than a single transaction, to all transactions executing on the failed site. A failed site will stop all its actions, therefore causing a delay to local transactions, and possibly an abort of subtransactions of global transactions running on its site. When the site recovers it will resume execution of the local transactions after

recovery procedures, and try to reach a decision for subtransactions that started before it failed; the algorithm that handles site failures is explained in detail in chapter 4.

B2PC is designed to tolerate site failures, and it guarantees the preservation of a transaction's atomic commitment. The price to pay is to abort the global transaction, or to block the subtransaction in the worst case. P2PC is similar to B2PC in the case of a site failure, except that it offers a second chance. This will result in additional unnecessary messages, in the case of site failures only. Fortunately, the rate of site failures is low, and as we will show the P2PC protocol compensates for these unnecessary messages by the transactions saved from an abort due to transient communication failure. O2PC participants decide unilaterally to commit by releasing their locks; this avoids blocking, but leads to compensating transactions, if the coordinator decides to abort.

The following experiments evaluate the performance of P2PC and O2PC due to site failures. In these experiments we increment site failures to the point where all protocols are equivalent or worse in performance than B2PC, to see the fault tolerance of each protocol in the environment of high site failure rates. The number of site failures is taken to be constant at all MPL for all protocols, and we increased the simulation time from 200 to 400 seconds per MPL to decrease the range of error. For example, if the number of site failures for an experiment is 3, then each MPL will have 3 site failures normally distributed over the period of each simulated MPL run, which is 400 sec. The duration of each site failure is from 8 to 10 seconds. At the same time, we choose the probability of 0.2% for communication failures, instead of a higher communication failure rate which would favor the P2PC protocol.

6.5.1. Results for 3 Site Failures

The first experiment was done simulating of 3 site failures for each level of multiprogramming for the three protocols. The results of this experiment show the degradation of performance for all three protocols as compared to Experiment 1.

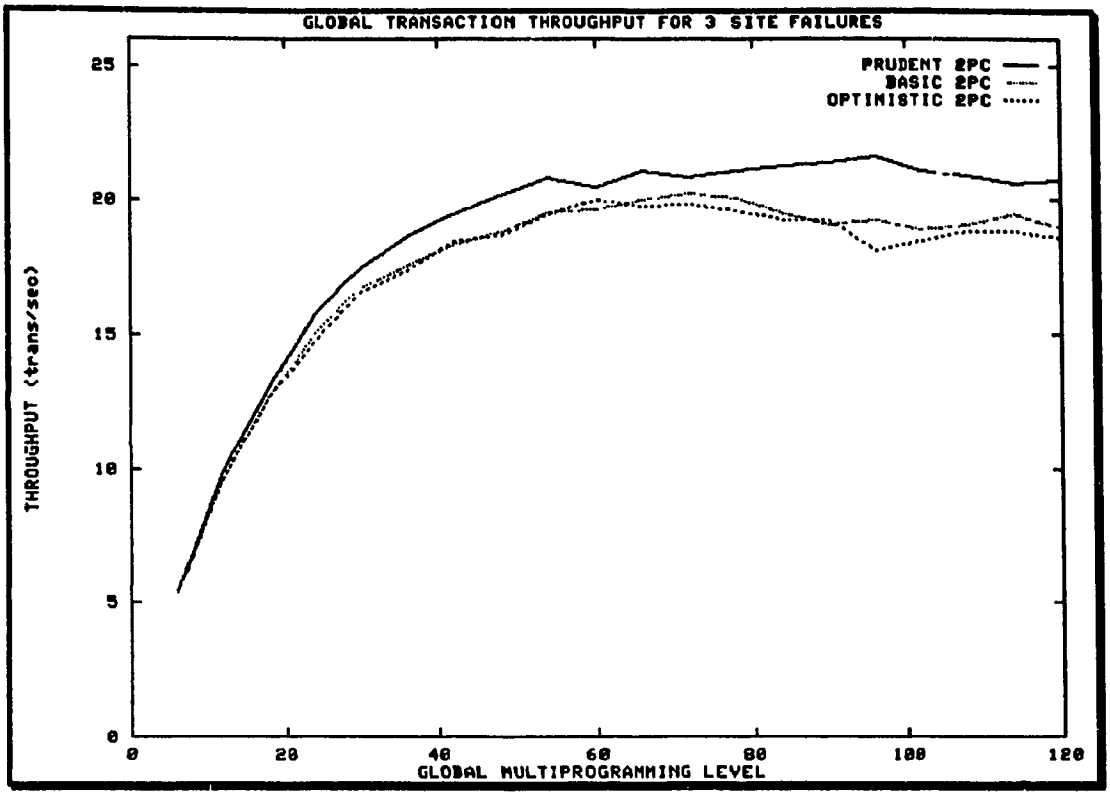

Figure 6.17. Global Transaction Throughput for 3 Site Failures

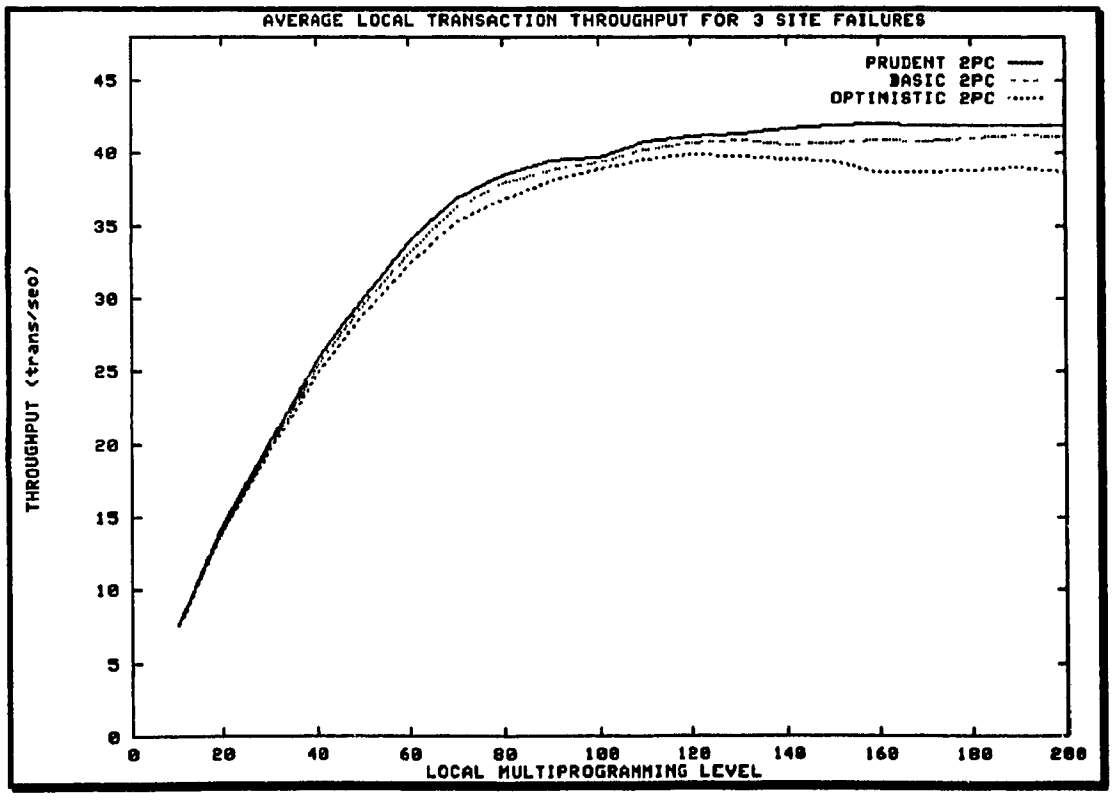

Figure 6.18. Average Local Transaction Throughput for 3 Site Failures

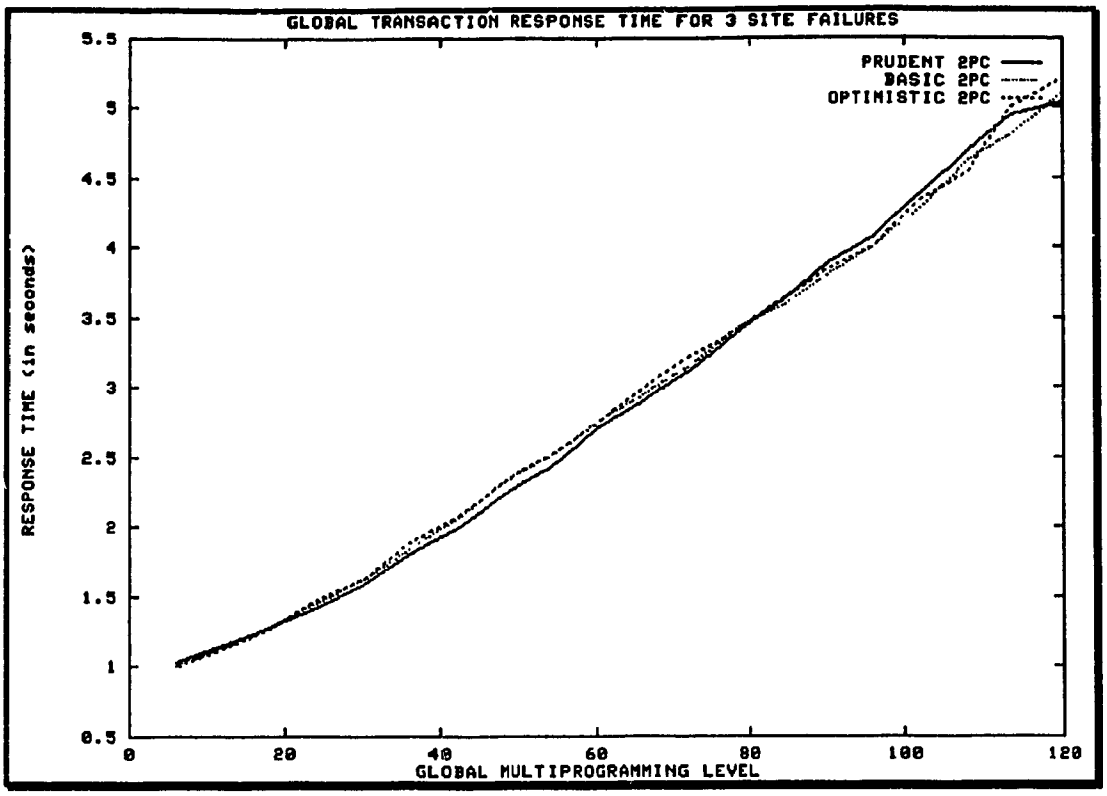

Figure 6.19. Global Transaction Response Time for 3 Site Failures

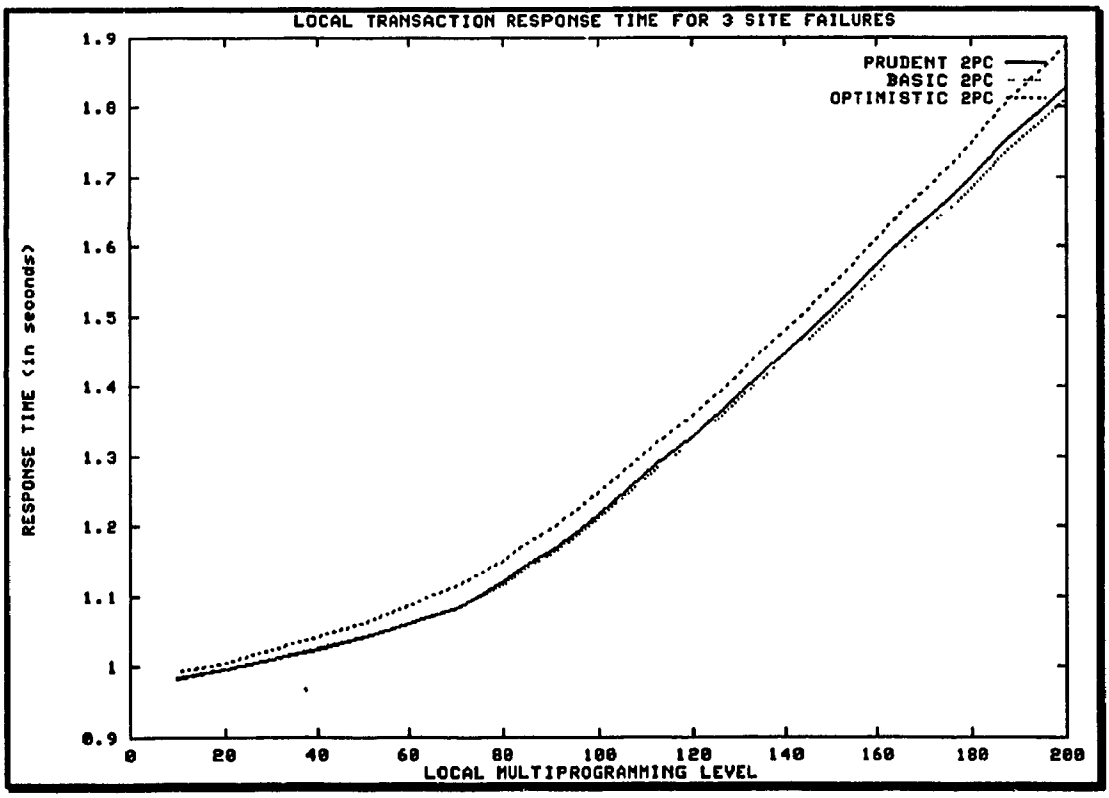

Figure 6.20. Local Transaction Response Time for 3 Site Failures

The throughput results show that P2PC still outperforms the other protocols for average local and global throughput. However, the difference is slightly less than in Experiment 1 (0.2% of communication failures), where it was nearly 3 transactions per second or 12% (Figure 6.6), while in this case the difference in performance has dropped to 2 transactions per second (10%) for the global throughput (Figure 6.17). The degradation of O2PC in the presence of failures is now more visible in Figure 6.18 for local throughput, where it performs three transactions (7%) worse than B2PC.

The global response time shown in Figure 6.19 is interleaved for the three protocols, showing that there is no relative degradation with respect to Experiment 1 (Figure 6.12). Also, the local transaction response time in Figure 6.20 is similar to Experiment 1 (Figure 6.11). One difference we notice is that O2PC has a slightly higher response time, since its throughput decreased due to the higher number of compensating transactions.

The global transaction aborts results in Figure 6.21 reveal an increase in the abort rate. For 3 site failures it increases up to 0.6 transactions aborts per second for P2PC, which is around three times the rate of Experiment 1 (Figure 6.10). In B2PC and O2PC it increases to around 1.5 times the rate of Experiment 1, with a maximum of 1.6 aborts per second. For the compensating transactions in Figure 6.22, the rate increases slightly compared to Experiment 1, and has a maximum of 0.65 compensating transactions per second which indicates that 3 site failures increases slightly the number of compensating transactions.

The number of messages exchanged with 3 site failures is higher for all protocols as shown in Figure 6.23. P2PC has a maximum of 13.5 messages exchanged per commit at the highest MPL; In the case of O2PC and B2PC, it is 14.6 messages exchanged per transaction. This shows that, for P2PC, the extra messages sent in the case of site failures were compensated for by the prevention of aborts in the case of communication failures, thus resulting in a lower message exchange per commit for that protocol.

We conclude from these simulation results that P2PC still outperforms B2PC and O2PC in the presence of 3 site failures. The additional number of messages required in case of a failure are compensated for by the decreased rate of aborts. O2PC performance degraded with respect to average local throughput, due to the number of compensating transactions issued.

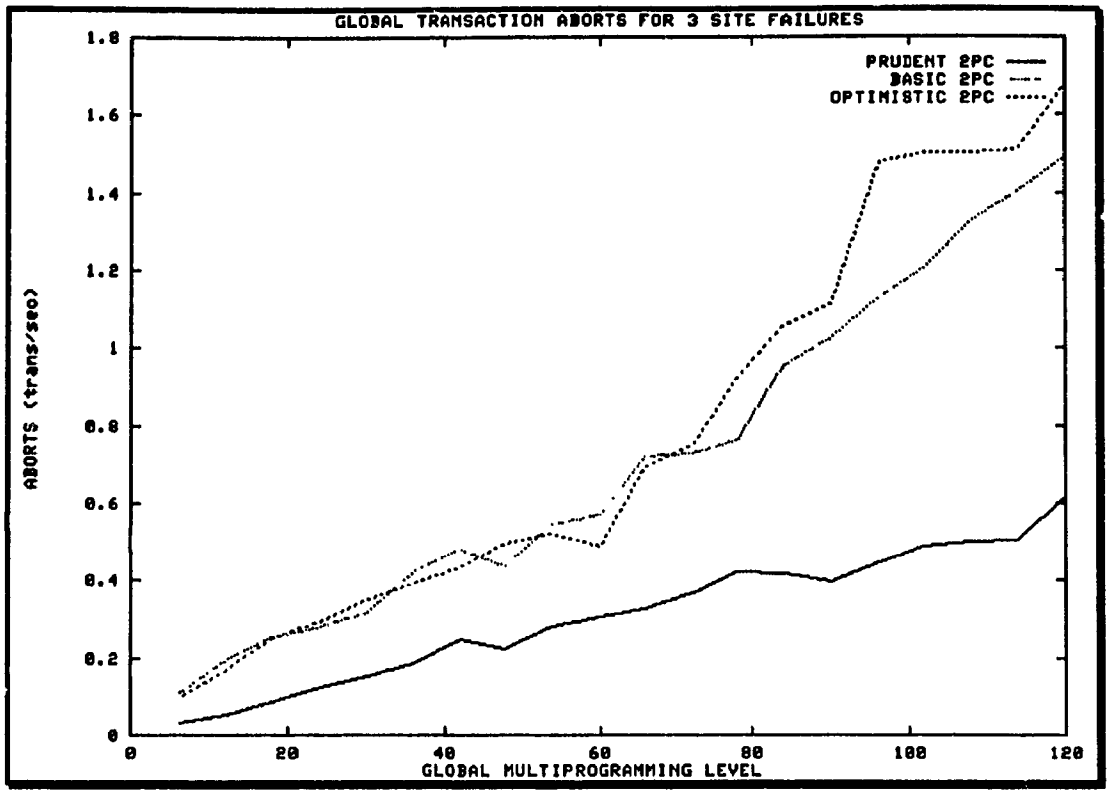

Figure 6.21. Global Transaction Aborts for 3 Site Failures

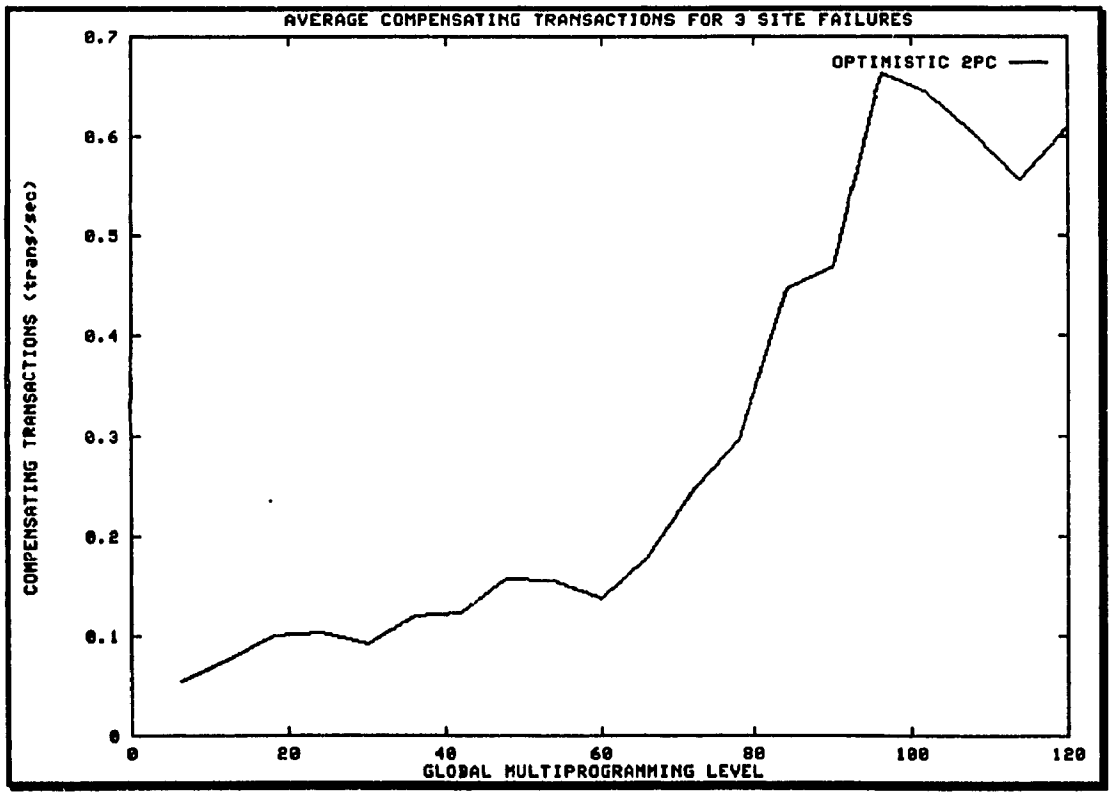

Figure 6.22. Average Compensating Transactions for 3 Site Failures

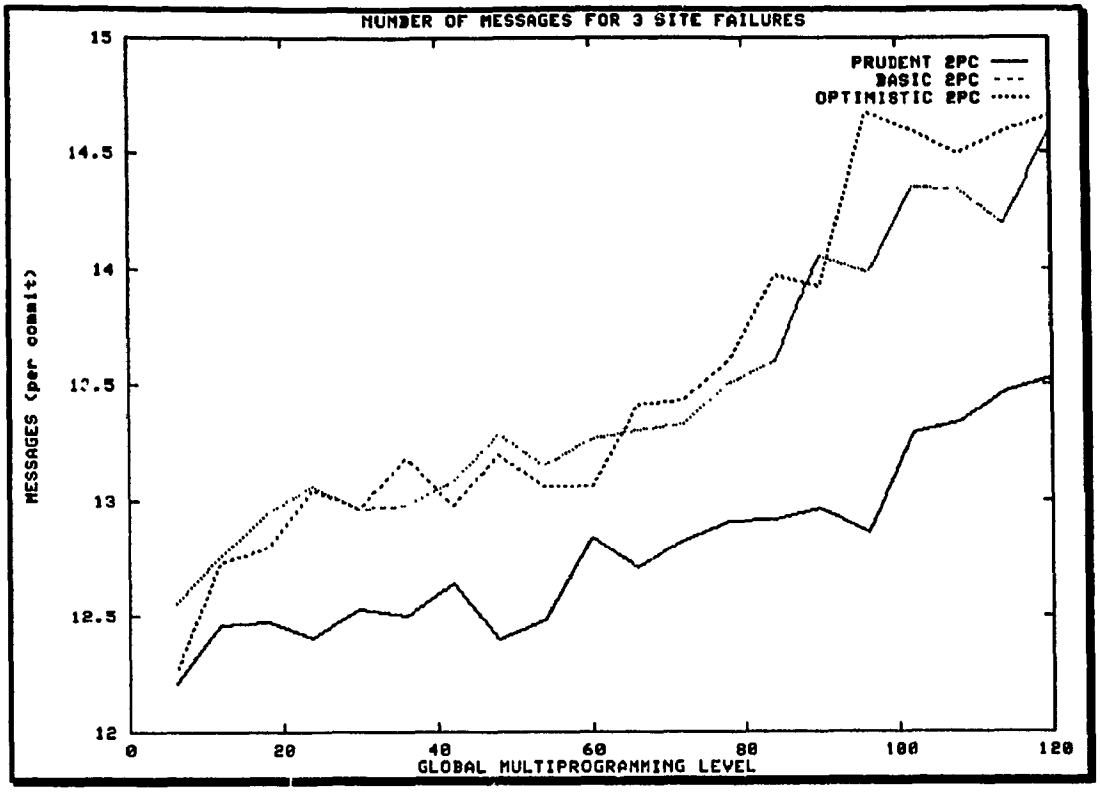

Figure 6.23. Number of Messages for 3 Site Failures

6.5.2. Results for 6 Site Failures

We increased the number of site failures from 3 to 6 per MPL. The results show that the throughput of P2PC is closer to B2PC, but did not fall below it. Figure 6.24 for the global transaction throughput shows close results for all three protocols. In this case, with higher MPL, we notice that the performance begins to fall off. P2PC performance is still distinctly better than the others, and it still outperforms slightly B2PC. The throughput reaches a maximum of 18 trans/sec at MPL of 60 in the case of P2PC, and then drops to 17 trans/sec at MPL of 120.

For the average local transaction throughput, P2PC results are same as those for B2PC at some MPL, as shown in Figure 6.25. However, on the average it outperforms B2PC, indicating the advantages of P2PC's second chance in overcoming the problems caused by the increased rate of failures. A visible difference can be seen now in the performance of O2PC, which drops dramatically to make a difference of 5 transactions per second (12%) between it and P2PC.

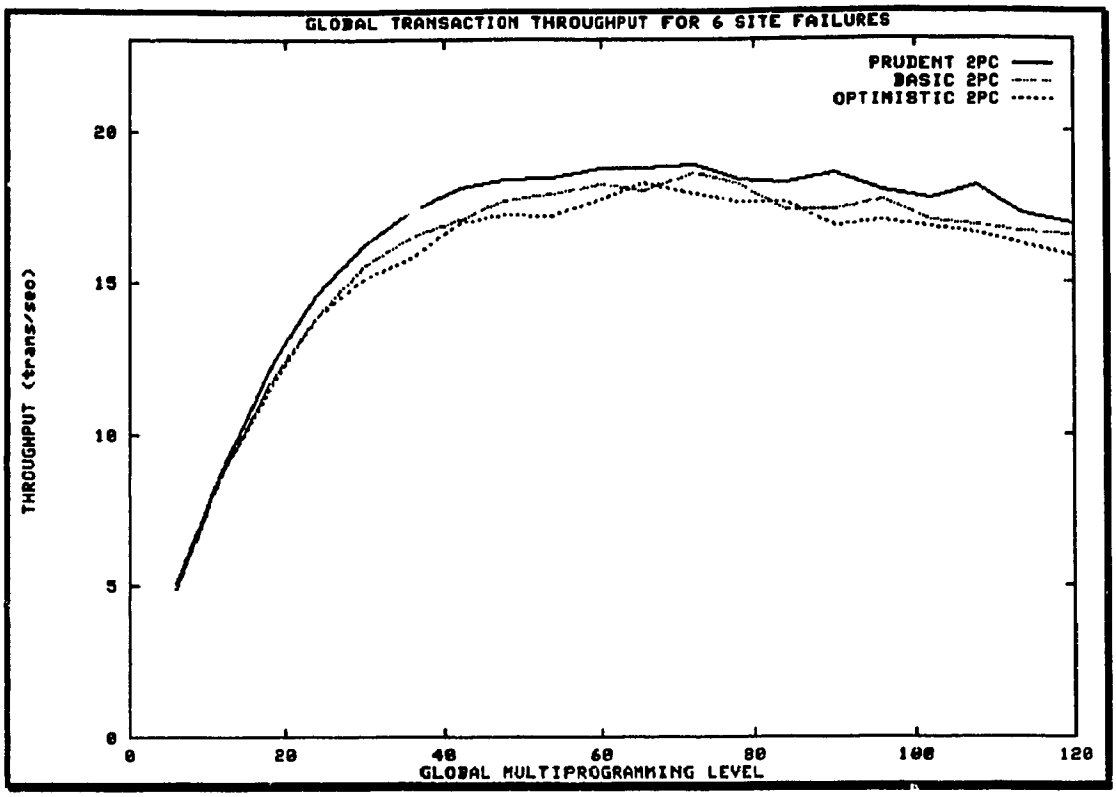

Figure 6.24. Global Transaction Throughput for 6 Site Failures

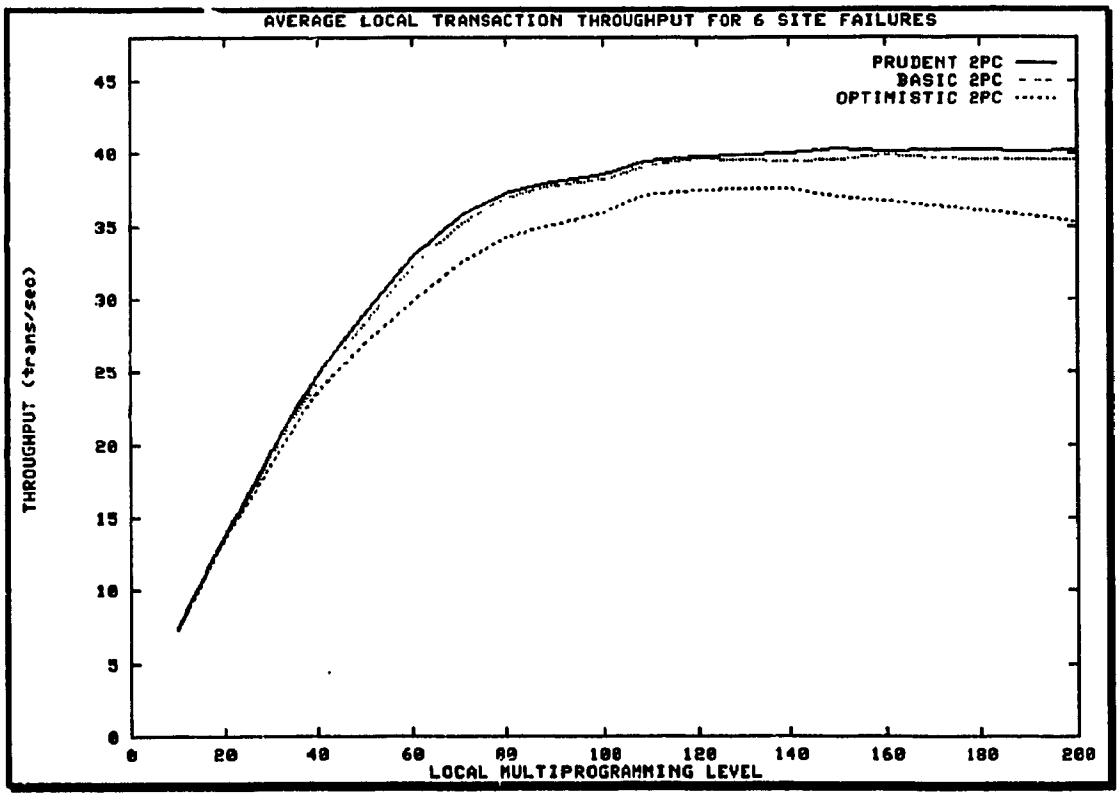

Figure 6.25. Average Local Transaction Throughput for 6 Site Failures

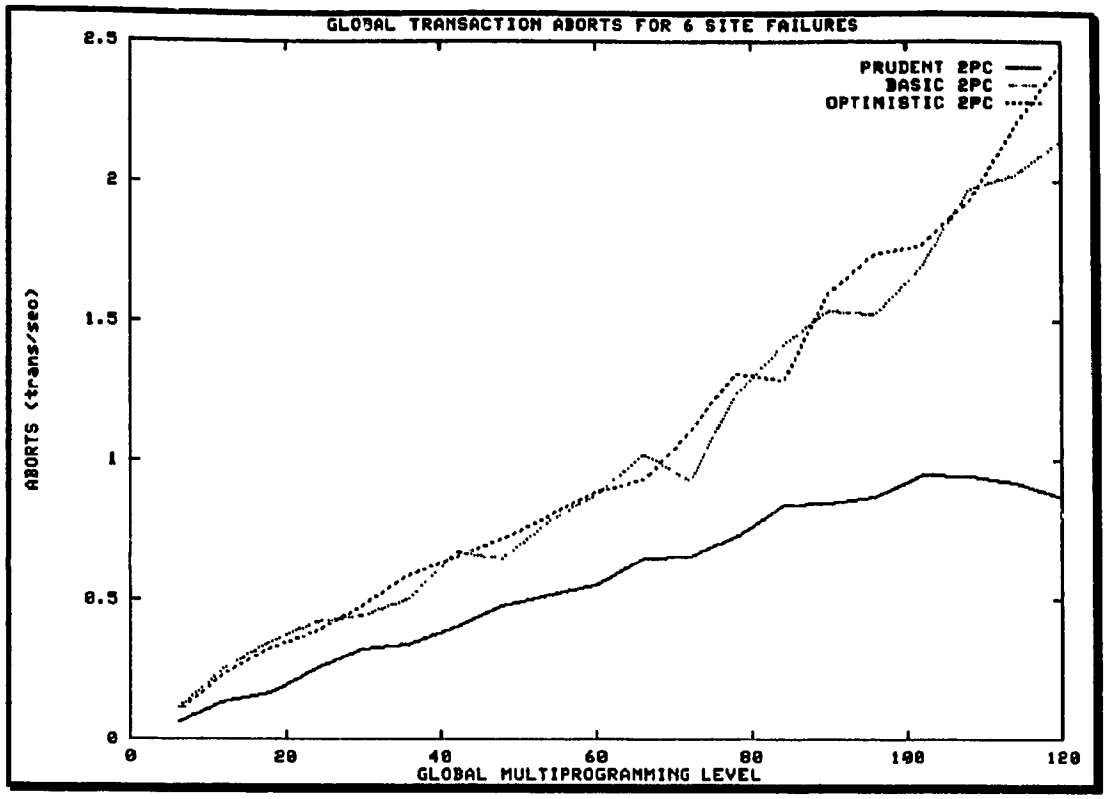

Figure 6.26. Global Transaction Aborts for 6 Site Failures

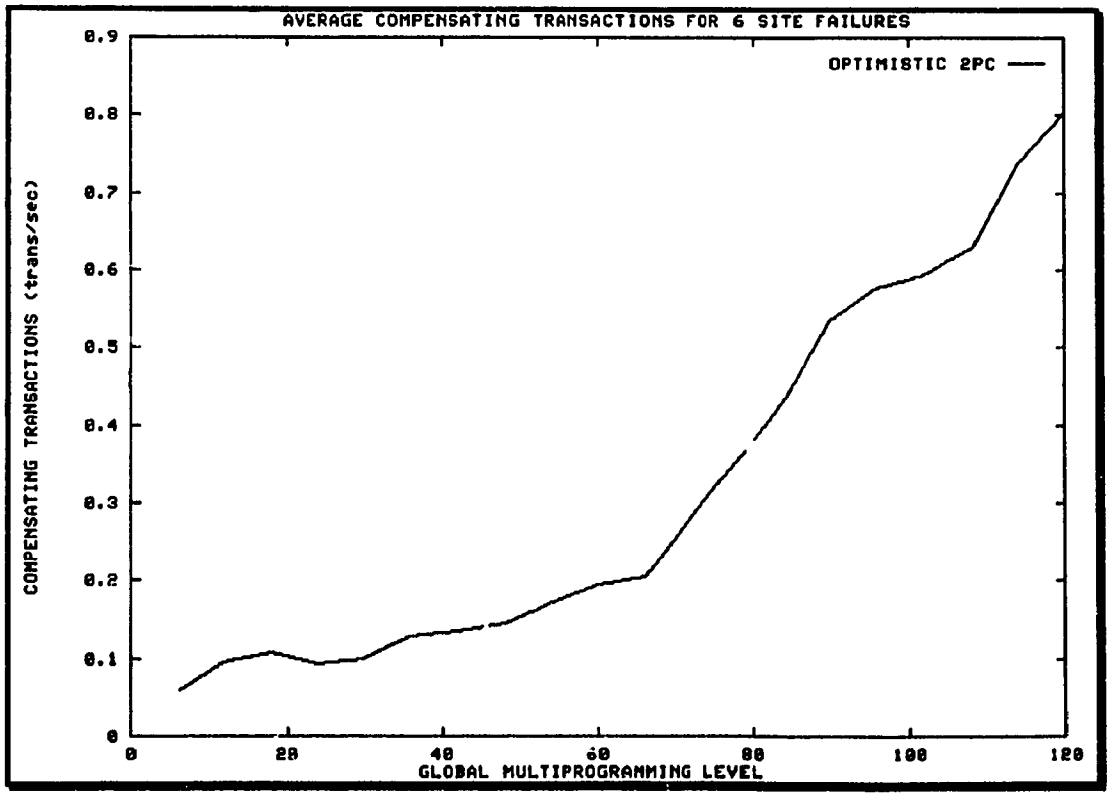

Figure 6.27. Average Compensating Transactions For 6 Site Failures

The global transaction aborts as shown in Figure 6.26 has increased for all protocols; it is close to 1 transaction per second for P2PC, and in the range of [2,2.5] for the other two protocols at the highest MPL. This shows again that P2PC has a lower rate of aborts, even in the presence of increased site failures. Also, the rate of compensations for the O2PC increased to reach 0.8 transactions per second, *versus* 0.65 for 3 site failures (Figure 6.27).

For this level of site failures, the number of messages exchange for P2PC, as shown in Figure 6.28, is closer to the other protocols, indicating that the additional messages sent in the case of a site failure are growing compared to the messages saved by not aborting the transaction in the case of a communication failure. However, this number is still lower than in the other two protocols.

Comparing the results of this experiment to the one for 3 Site failures, we can deduce that increasing the site failure rate affects the performance of P2PC and O2PC more than B2PC. P2PC still outperforms B2PC, while O2PC performance degrades dramatically.

6.5.3. Results for 9 Site Failures

In this experiment we increased the site failures to 9 for each MPL. The results of this experiment met our goal, which was to achieve a throughput for O2PC and P2PC equal to or worse than B2PC. In fact, by increasing the rate of failures to this level, the graph of P2PC and B2PC criss-crossed, as shown in Figure 6.29. This illustrates that in the worst case the performance of P2PC was not lower than B2PC. The throughput figures also show that the global transaction performance for the three protocols interleave after MPL of 60, indicating a similar performance for all three protocols. We also notice that throughput decreases with increasing MPL, which is a sign of thrashing; this happens because the system is not completing the transactions, and the abort restart rate is high due to increased site failures.

The global throughput results in Figure 6.30 demonstrate the similar level of performance of P2PC and B2PC. It is to be noted that B2PC does not outperform P2PC, even in the presence of high site failure rates. This again supports our conclusion that taking care of communication failures is not at the expense of site failures. Contrast this with O2PC, where one is at the expense of the other.

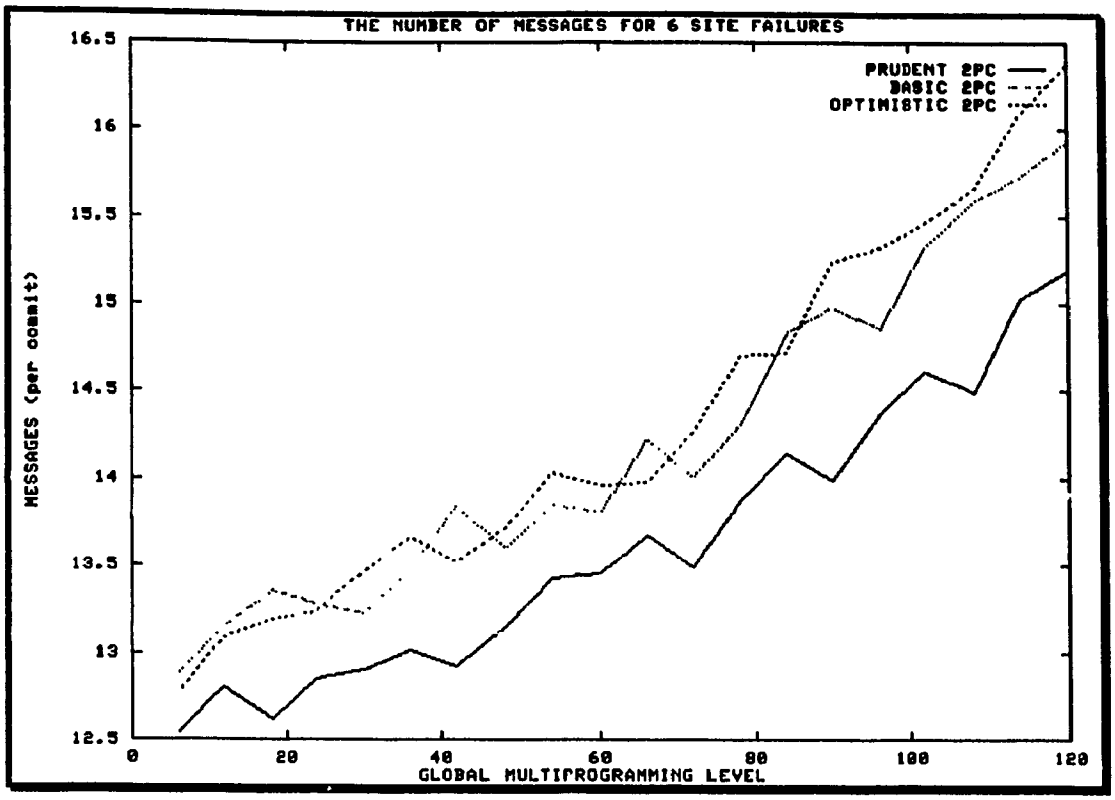

Figure 6.28. Number of Messages for 6 Site Failures

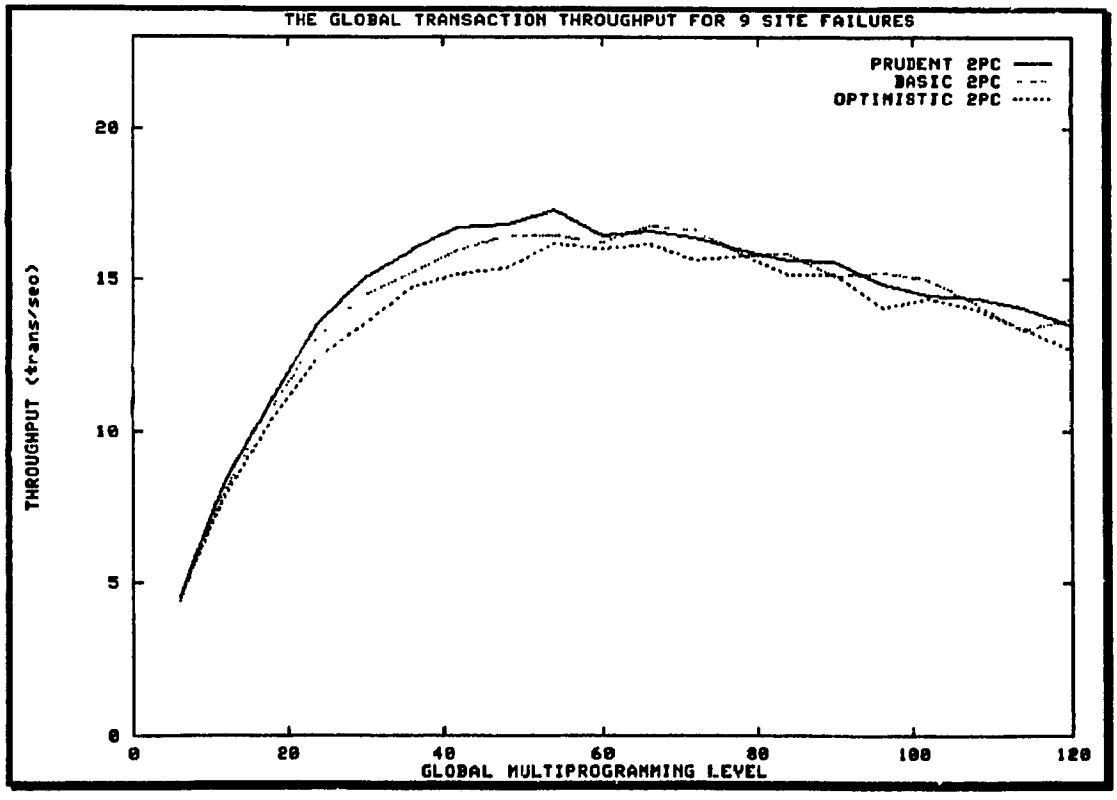

Figure 6.29. Global Transaction Throughput for 9 Site Failures

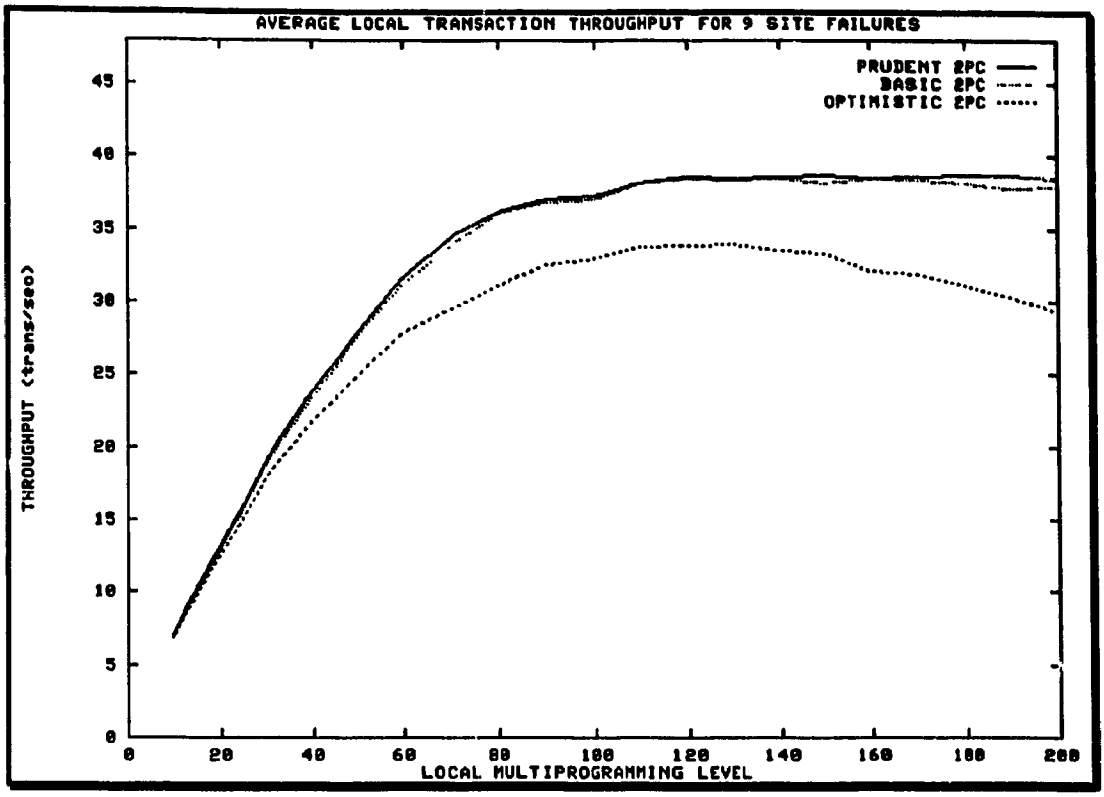

Figure 6.30. Average Local Transaction Throughput for 9 Site Failures

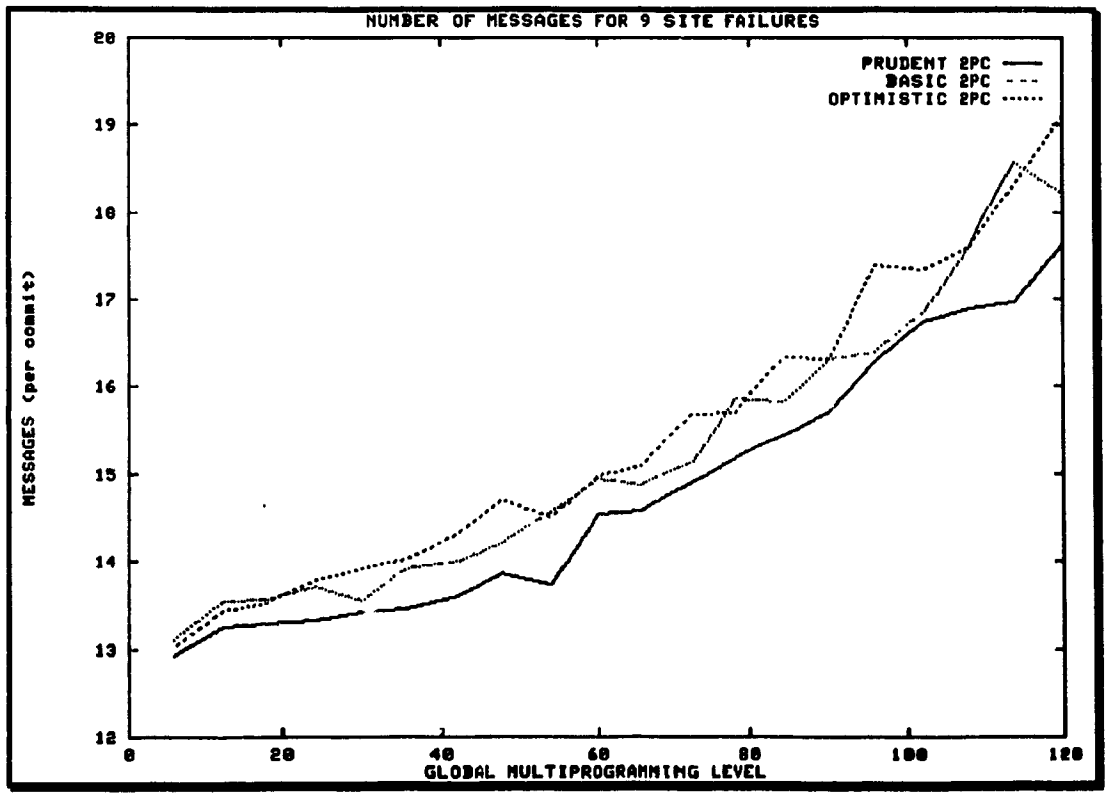

Figure 6.31. Number of Messages for 9 Site Failures

Furthermore, basing a protocol on the assumption that failures do not occur or rarely occur creates a simulation which performs badly when the assumption is wrong. This is seen clearly in Figure 6.30, where a high difference is visible between O2PC and the other two protocols.

Let us now consider the number of messages exchanged, which may be used as a measure of message complexity (Section 4.4.2.2). The results of messages exchanged, as shown in Figure 6.31, are similar for all protocols; however, the values for P2PC never exceed those of the others. This demonstrates that the message complexity for P2PC is the lowest per transaction commit in the presence of failures.

6.6. Concluding Remarks

We notice in these experiments that B2PC is tolerant of site failures, but at the expense of communication failures, which were treated the same as site failures and resulted in a lower system performance. In the absence of failures O2PC outperforms B2PC slightly, but in the presence of failures B2PC outperforms O2PC.

Finally, we conclude from these experiments that the performance of P2PC was better than the other protocols; even in the worst cases it is as good as B2PC. Communication failures are handled better with P2PC; even with high site failures P2PC overcomes their effects and avoids aborts, as in the communication failure cases.

CHAPTER 7

CONCLUSION

The Basic Two-Phase Commit protocol is a simple and elegant Atomic Commitment Protocol. It is designed to treat all failures in a distributed system as a site failure. Unfortunately, this leads to aborting global transactions even in the case of a simple communication failure, resulting in a lower system performance.

In this thesis we present the Prudent Two-Phase Commit, and study its performance using a distributed database simulation. The salient feature of P2PC is that it provides another chance to a global transaction before aborting it. P2PC gives another chance to a participant or coordinator before making a decision; this second chance rectifies many faults due to communication failure. In this way it prevents an unnecessary abort and, as a result, improves system performance.

The cost of implementing a P2PC is the additional rounds of messages and the associated Timeout. We demonstrated using our simulation that this cost is justified by abort prevention, which in turn leads to fewer messages per transaction commit, and an increase in throughput. Also, we demonstrated that in the case of high site failure rates, P2PC does not degrade, nor does its performance fall below B2PC.

We compared the performance of P2PC to B2PC and O2PC, two well-known protocols that use different strategies. O2PC is known for its optimistic assumption of low failure rates, hence permitting the participants to unilaterally commit before the final decision of the coordinator. The side effect of this protocol is in issuing compensating transactions, to undo semantically the effects of the miss-committed transaction without resorting to cascading aborts. The overhead of semantic compensation is not discussed in our study, however; we treated this compensation as an additional transaction, therefore limiting the effects of the compensation to local throughput.

The strategy we followed for testing the performance of P2PC was to test according to low levels of failures, and then to increment the rate of failures up to the point where the protocol degrades in performance to the level of B2PC. We performed our experiments in the following order: no failures, only communication failures, increased communication failures, 3 Site failures, 6 Site failures, and finally 9 Site failures.

The results of our experiments reveal that O2PC slightly outperforms B2PC in the absence of failures at medium MPL; at higher MPL its performance is similar to B2PC. With the presence of site failures O2PC's performance degrades dramatically compared to B2PC and P2PC, showing its intolerance of failures. A high rate of compensating transactions is behind this degradation of O2PC performance, revealing that O2PC was designed at the expense of robustness and fault tolerance.

P2PC outperforms B2PC and O2PC in the presence of communication failures; its improvement was 12% at the highest MPL for global throughput results, 8% for the number of messages exchanged, and 4% in the average local throughput. P2PC outperforms the others for low and medium site failure rates; it is similar in performance to B2PC in the presence of high site failure rates, demonstrating that P2PC is more robust than B2PC and O2PC in the presence of both site and communication failures.

We deduce the following rules in order to use one of the ACP studied in this thesis:

- O2PC is good for systems where site and communication failures do not occur or the probability of their occurrence is extremely low. It also requires that running semantic compensation is possible without resorting to cascading of compensating transactions.
- B2PC is good for systems where communication failures occur rarely and the site failures is high. The communication failures rate should not exceed the site failures rate.
- P2PC is good for systems with both communication failures and site failures. The site failures rate should not exceed the communication failures rate. The advantage of P2PC is that without failures or delays it is equivalent to B2PC.

Finally, we conclude that P2PC retains the elegance and ease that are inherent in B2PC, with the additional feature of taking into account communication failures by treating them differently than a site failure, thus producing a protocol that has robust system performance. Future enhancements to P2PC would involve adding the *ping* operation [DESA] to its implementation; future work would involve applying this protocol to Highly Distributed DBMS (HDDBMS) where each database itself is a HDDBMS [DESA].

References

[AGRA87] Rakesh Agrawal, Michael J. Carey and Miron Livny, "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems*, Vol. 12, No 4, December 1987, pp. 609-654.

[AGRA92] D. Agrawal, A. El Abbadi, and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases", *SIGMOD Record*, Vol. 21, No. 2, June1992, pp. 104-113.

[AGRA93] Divyakant Agrawal and Soumitra Sengputa, "Modular Synchronization in Distributed Multiversion Databases: Version Control and Concurrency Control", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 1, February 1993, pp. 126-137.

[BERN87] Philip A. Bernstein, V. Hadzilacos, and Nathan Goodman, Concurrency Control and Recovery in Distributed Database Systems, Addison Wesley Publishing Company, Reading, Massachusetts, 1987.

[CICI92] Bruno Ciciani, Daniel M. Dias, and Philip S. Yu, "Analysis of Concurrency-Coherency Control Protocols for Distributed Transaction Processing Systems with Regional Locality", *IEEE Transactions on Software Engineering*, Vol. 18, No. 10, October 1992, pp. 899-914.

[DASG90] Partha Dasgupta and Zvi M. Kedem, "The Five-Color Concurrency Control Protocol: Non-Two-Phase Locking in General Databases", *ACM Transactions on Database Systems*, Vol. 15, No.2, June 1990, pp. 281-307.

[DELI93] Alexios Delis and Nick Roussopoulos, "Performance Comparaison of Three Modern DBMS Architectures", *IEEE Transactions On Software Engineering*, Vol. 19, No. 2, February 1993, pp. 120-138.

[DESA] Bipin C. Desai, "A Prudent Commit Protocol for HDDBMS", (to appear). Also availbale as URL: href: //www.cs.concordia.ca/ ~bcdesai/ www/P2PC.html.

[DESA90] Bipin C. Desai, An Introduction to Database Systems, West Publishing Company, St. Paul, Minnesota, 1990.

[HUNG92] S. L. Hung and K. Y. Lam, "Locking Protocols for Concurrency Control in Real-time Database Systems", *SIGMOD Record*, Vol. 21, No. 4, December 1992, pp. 22-27.

[LEVY91] Eliezer Levy, Henry F. Korth, and Abraham Silberschatz, "An Optimistic Commit Protocol for Distributed Transaction Management.", *SIGMOD Record*, Vol. 20, No. 2, May 1991, pp. 88-97.

[MOHA92] C. Mohan, Hamid Pirahesh, and Raymond Lorie, "Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions", *SIGMOD Record*, Vol. 21, No. 2, June 1992, pp. 124-132.

[ÖZSU91] M. Tamer Özsu and Patrick Valduriez, Principles of Distributed Database Systems, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[RAHM93] Erhard Rahm, "Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems", *ACM Transactions on Database Systems*, Vol. 18, No. 2, June 1993, pp. 333-377.

[ROTH93] Kurt Rothermel and Stefan Pappe, "Open Commit Protocols Tolerating Commission Failures", *ACM Transactions on Database Systems*, Vol. 18, No. 2, June 1993, pp. 289-332.

[STAM90] James W. Stamos and Flavin Cristian, "A Low-Cost Atomic Commit Protocol",Proceedings, Ninth Symposium on Deliverable Distributed Systems, IEEE 1990, pp. 66-75.

[THOM91] Alexander Thomasian and In Kyung Ryu, "Performance Analysis of Two-Phase Locking", *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991, pp. 386-401.

[TRAV92] Bruno Traverson, "Optimization Strategies and Performance Evaluation of the Two-Phase Commit Protocol", Eigth International Conference on Software Engineering for Telecommunications Systems and services, Pub. IEE London, UK, April 1992, pp. 52-56.

[ULUS92] Özgür Ulusoy, "Current Research on Real-Time Databases", *SIGMOD Record*, Vol. 21, No. 4, December 1992, pp. 16-21.

[WEIK91] Gerhard Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", *ACM Transactions on Database Systems*, Vol. 16, No. 1, March 1991, pp. 132-180.

[WOLF90] Ouri Wolfson, "A Comparative Analysis of Two-Phase-Commit Protocols", INRIA. ICDT '90, Third International Conference on Database Theory Proceedings, pp. 291-304.

[YUAN91] Shyan-Ming Yuan, "An Efficient Fault-Tolerant Decentralized Commit Protocol for Single Site Failure", *BIT*, Vol. 31, No.1, 1991, pp. 53-68.

APPENDIX A. SOURCE CODE FOR THE SIMULATION PROGRAM

This appendix lists the source code for the program used in the ditsributed database simulation. This listing includes the main module; some functions are not listed due to their secondary role, such as input and printing functions. This program was written in SRC Modula-3 code; it compiles and runs under DEC-ALPHA under OSF/1.

This module simulates a distributed database system having n sites, each site is a centralized database system by itself. The locking protocol used is the two phase locking protocol, the atomic commitment for global transactions is synchronized by the two phase commit protocol. The goal of this program is to produce performance results of Basic, Optimistic and Prudent Two-Phase Commit protocols. The system performance is measured by the throughput which is the number of committed transactions per second.

```
MODULE Main;
IMPORT Thread, Rnd, Scan, Rd, Wr, Fmt ;
FROM InOut IMPORT WriteTxt, WriteCard, WriteLn, WriteInt, WriteLongReal,
WriteReal;
FROM Scheduler IMPORT Yield;
FROM IO IMPORT OpenRead, OpenWrite;

(* Defining the parameters used as constants in the system *)

CONST

    Thr = 200;
    Unit = 1000000.0; (* Unit of time with respect to one second *)
    Maxnt = 350; (* The maximum number of transactions that exist *)
    Maxdt = 200; (* The maximum number of global transactions *)
    Maxns = 700; (* The maximum number of seconds for an interval *)
    Nsites = 6; (* The number of sites *)
    Lpercent = 40; (*The percentage of local transactions in the system*)
    Messlength = 512; (* Average message length *)
    Messunit = 64; (* The unit of a message *)
    Sitef    = 2; (* 2 will give the same number of failures each Mpl,
                1 the number of site failures may vary between Mpl.*)

TYPE
  Transaction = RECORD
      id : CARDINAL;
      rw : CARDINAL;
    END;

    (* declaring the log record,  where the information of a global
      transaction   are kept *)
    LOG = RECORD
      vote_decision, commit_decision, vote_request, finished : TEXT;
      site_fail, deadlock, location : TEXT;
```

```
END;

(* Declaring the failure record where the count of the different kinds
  of failures are kept. *)
FAILURE = RECORD
    deadlock, site, message : ARRAY [1..Maxns + 1] OF CARDINAL;
END;

(* Declaring the transaction's manager closure *)
(* The parameters  should be passed to the transaction manager upon its
  creation *)
Tmclosure = Thread.Closure OBJECT
    id : CARDINAL
    OVERRIDES
     apply := TM
END;

(* Declaring the transaction closure *)
(* the different parameters are: *)
(* site: the site of the transaction *)
(* seq : the sequence of the transaction *)
(* id : The unique id of the transaction *)
(* did : The global id of the subtransaction *)
(* coordinator : The site of the coordinator *)
(* index :The index that will serve to store the log information *)
(* nparticipant : The number of the participants in the case of a global  transaction *)
(* type : The type of the transaction if it is local or global *)
(* class : The class of the transaction if it is a participant or a   coordinator *)
(* participant : is the list of participants involved in the global    transaction *)
(* SCHEDULER : is the body of the transaction,  i.e the procedure that  the transaction
    is going to take place *)

Tclosure = Thread.Closure OBJECT
    site, seq, id,  did,  coordinator, index, nparticipant : CARDINAL;
    type, class : TEXT;
    participant : ARRAY[1..Nsites] OF CARDINAL;
    OVERRIDES
     apply := SCHEDULER
END;

(* Declaring the coordinator closure *)
(* the different parameters are : *)
(* site: the site of the coordinator *)
(* id : The unique id of the coordinator *)
(* index :The index that will serve to store the log information *)
(* nparticipant : The number of the participants *)
(* participant : is the list of participants involved in the global  transaction *)

Cclosure = Thread.Closure OBJECT
     site, id , nparticipant, index : CARDINAL;
     participant : ARRAY[1..Nsites] OF CARDINAL

     OVERRIDES
      apply := COORDINATOR
```

```
END;

(* Declaring the siterecovery closure *)

Rclosure = Thread.Closure OBJECT
    Sid : CARDINAL;
    OVERRIDES
      apply := SITERECOVER
END;
(* declaring the site record with all the necessary variables *)
(* The variables are : *)
(* Trc : is the transaction count *)
(* Tid : is the transaction unique id *)
(* Nudisk : is the number of utilized disk *)
(* Nucpu is the number of utilized CPU *)
(* Ntrans: is the number of transaction left *)
(* NT: is the number of finished transactions in a second. *)
(* Conflict: is the count of conflict *)
(* TRC : is the transaction count for each type of transaction *)
(* Thruput: is the count of transactions that finish *)
(* Response : is the response time of a transaction *)
(* Fail : will record if the site is failing or not *)
(* Tra : is the thread transaction *)
SITE = RECORD
  Trc, Tid,  Nudisk, Nucpu, Ntrans : CARDINAL;
  NT,  Conflict : INTEGER;
  TRC : ARRAY[1..2] OF CARDINAL;
  NTRANS, CONFLICT : ARRAY[1..2] OF INTEGER;
  Thruput : ARRAY[1..Maxns+1] OF CARDINAL;
  Compensate : ARRAY[1..Maxns+1] OF CARDINAL;
  THRUPUT : ARRAY[1..2], [1..Maxns+1] OF CARDINAL;
  Response : ARRAY[1..Maxns+1] OF REAL;
  RESPONSE : ARRAY[1..2], [1..Maxns+1] OF REAL;
  Fail, Lfail : TEXT;
  Tra : ARRAY [1..Maxnt] OF Thread.T;
  (* Declaring mutex semaphores *)
  mutex,  mutex1, mutex3, mutex4, Mfail : MUTEX;
END;

(* Declaring the communication links *)
LINK = RECORD
  Sites : ARRAY [1..Nsites], [1..Nsites] OF CARDINAL;
  Mutex : ARRAY [1..Nsites], [1..Nsites] OF MUTEX;
END;

-------------------------------------------------------------------
            declaring the global variables
-------------------------------------------------------------------

VAR
 (* declaring global variables that are common to all threads *)
 (* Timer : is the time of the clock in terms of seconds. *)
 Timer : REAL;
 Ntime, ii, jj, Mpl, Lmpl, Dmpl, Maxcpu, Dtid,  Dpercent,  Dtri : CARDINAL;
```

```
Ndtrans, Dtrc, Nsfail : CARDINAL;
Dthru, CouFail, Dmessage : ARRAY[1..Maxns+1] OF CARDINAL;
Dresponse, Dmresponse : ARRAY[1..Maxns+1] OF REAL;
Scount : ARRAY[1..Nsites] OF CARDINAL;
Site : ARRAY[1..Nsites] OF SITE;
Log : ARRAY[1..Maxdt], [1..Nsites] OF LOG;
Link : LINK;
Failure: FAILURE;

(* declaring the coordinators threads *)
  Coord : ARRAY[1..Maxnt]OF Thread.T;

(* declaring the transaction managers threads *)
Transm : ARRAY[1..Nsites] OF Thread.T;

(* declaring the other threads *)
Clock, Throughput, Site_recover, Sitefail : Thread.T;

(* declaring the mutex semaphores that will be used for mutual exclusion *)
(* to protect global variables, and resources. *)
Moutput, MUTEXCD, DMUTEX, MUTEXF, MUTEXM := NEW(MUTEX);


**************************************************
      The variables that will be read from an input file.
**************************************************

N, Thrt, Trmin, Trmax, Stept, Debug, MaxMpl, Timeout, Transt : CARDINAL;
Timeout1, Timeout2, Ddebug, Debugt, Fdebug, Mdelay, Atimeout : CARDINAL;
Smpl, Nthr, Maxdisk, Diskaccess, Cpuaccess, Extthink, Protocol : CARDINAL;
Cpause : CARDINAL;
Psitefail, Pmesslost : REAL;
Outfile, Outfile1, Outfile2, Outfile3, Outfile4, Outfile5 : TEXT;
Outfile6, Outfile7, Outfile8, Outfile9, Outfile10, Outfile11 : TEXT;
(* N : Number of items in the database *)
(* Thrt is the unit of time that the throughput is calculated *)
(* Trmin is the minimum set of read and write actions *)
(* Trmax is the maximum set of read and write actions *)
(* Stept is the execution time of a single step of a transaction *)
(* Debug is the debugging level : 1, 2, 3 *)
(* MaxMpl is the maximum multiprogramming level *)
(* Transt is the type of the transaction 1- variable, 2- fixed*)
(* Outfile is the name of the file that the data output will be printed *)
(* Smpl is the increment that will be added to the current Mpl level i.e
  The next Mpl will be equal to Mpl + Smpl *)
(* Nthr is the number of times that the throughput should be done before
  calculating the average throughput*)
(* Maxdisk is the maximum number of disks *)
(* Diskaccess is the average time to read or write to a disk *)
(* Cpuaccess is the average time to access the cpu *)
(* Extthink is the external think time of a transaction *)
```

Pause

This procedure will make the transaction pause for a certain number of microseconds.

```
PROCEDURE Pause(time : REAL;  delay : CARDINAL ) =
BEGIN
 (* Wait until the delay + time of trans will be less than the Timer *)
 WHILE (time + (FLOAT(delay)/Unit)) > Timer
     DO
     Yield(); (* switch to the next ready process *)
 END;
END Pause;
```

LongPause

This procedure will make the transaction pause for a certain number of seconds.

```
PROCEDURE LongPause (time : REAL; delay : CARDINAL.) =
BEGIN
 (* Wait until the delay + time of trans will be less than the Timer *)
 WHILE (time + FLOAT(delay)) > Timer
     DO
     Yield(); (* switch to the next ready process *)
 END;
END LongPause;
```

EXP

This procedure will return the exponential of a real number

```
PROCEDURE EXP(V : REAL;  N : CARDINAL):REAL =
VAR i : CARDINAL;
   c : REAL;
BEGIN
   c := 1.0;
   FOR i := 1 TO N DO
     c := c * V;
   END;
   RETURN c;
END EXP;
```

MOMENT

This procedure will return the moment of a lock that is given by the formula Km = SUM k>0 k^m Pl(k)

```
PROCEDURE MOMENT(m, p : CARDINAL):REAL =
VAR mr, pr : REAL;
   i, sum : CARDINAL;

BEGIN

mr := FLOAT(m);
pr := FLOAT(p);
```

```
sum := 0;
FOR i := 1 TO p DO
  sum := sum + TRUNC(EXP(FLOAT( i), m));
END;

RETURN (FLOAT(sum)/FLOAT(p));

END MOMENT;
```

PDEADLOCK

This procedure will return the probability of deadlock for the current type of transaction.

```
PROCEDURE PDEADLOCK(sid : CARDINAL):CARDINAL =
VAR k1, k2, k3, k : REAL;
  M, Nr, pr : REAL;
BEGIN
 M := FLOAT(Site[sid].Trc); (* Number of transactions in the system *)
 Nr := FLOAT (N); (* The size of the database *)
 k := FLOAT(Trmax); (* The number of locks required by a fixed size tr *)

 (* If the transaction type is variable then apply the formula on the
   variable type, else apply the fixed type formula *)
 IF Transt = 1 THEN
    k1 := MOMENT(1, Trmax); (* Get the first moment *)
    k2 := MOMENT(2, Trmax); (* Get the second moment *)
    k3 := MOMENT(3, Trmax); (* Get the third moment *)
    pr := (M - 1.0) *(k2 - k1)* (k3 -1.0)/( (12.0*(k1 + 1.0)) *
          EXP(Nr, 2) ) ;
 ELSE
    pr :=  (M - 1.0) * EXP(k, 4) /(12.0*(EXP(Nr, 2)));
 END;
 RETURN(TRUNC (pr * 10000.0));

END PDEADLOCK;
```

PCONFLICT

This procedure will return the probability of conflict for the current type of transaction.

```
PROCEDURE PCONFLICT(sid : CARDINAL):CARDINAL =
VAR k1, k2, k : REAL;
  M, Nr, pr : REAL;
BEGIN
 M := FLOAT(Site[sid].Trc); (* Number of transactions in the system *)
 Nr := FLOAT (N); (* The size of the database *)
 k := FLOAT(Trmax); (* The number of locks required by a fixed size tr *)

 IF Transt = 1 THEN
   k1 := MOMENT(1, Trmax); (* Get the first moment *)
   k2 := MOMENT(2, Trmax); (* Get the second moment *)
```

```
  pr := (M - 1.0)*(k2 + k1)/ (2.0 *(k1 + 1.0) * Nr)

 ELSE
   pr := ( M - 1.0) * k / ( 2.0 * Nr) ;
 END;
 RETURN(TRUNC (pr * 10000.0));

END PCONFLICT;
```

DELAY

This procedure will return the delay time for a conflict according to the type of transaction.

```
ROCEDURE DELAY():CARDINAL =
VAR k1, k2, k3, k, s : REAL;

BEGIN
 k := FLOAT(Trmax); (* The number of locks required by a fixed size tr *)
 s := FLOAT (Stept); (* The step duration *)

 (* If the transaction type is variable then return the delay for the
   variable type, else return the delay for the fixed type *)
 IF Transt = 1 THEN
    k1 := MOMENT(1, Trmax); (* Get the first moment *)
    k2 := MOMENT(2, Trmax); (* Get the second moment *)
    k3 := MOMENT(3, Trmax); (* Get the third moment *)
    RETURN( TRUNC ( ((k3 - k1)/(3.0 * (k2 + k1) ) * s) + s/2 ) );
 ELSE
    RETURN (TRUNC ((s *(k - 1.0) / 3.0) + s/2) ) ;
 END;

END DELAY;
```

CREATETRANS

This procedure will create a local transaction.

```
PROCEDURE CREATETRANS(sid:CARDINAL)=
VAR ntrans : CARDINAL;
BEGIN

 LOCK Site[sid].mutex4 DO
  (* WHILE the number of transactions is less than the Mpl *)
  WHILE (Site[sid].TRC[1] < Lmpl ) DO
     Site[sid].Ntrans := Site[sid].Ntrans + 1;
     IF Site[sid].Ntrans > Maxnt THEN
        Site[sid].Ntrans := 1;
```

```
      END;
      ntrans := Site[sid].Ntrans;
      (* Increment the number of total transactions in the system *)
      Site[sid].Trc := Site[sid].Trc + 1;
      (* Increment the number of local transactions *)
      Site[sid].TRC[1] := Site[sid].TRC[1] + 1;
      (* Give an id to the transaction *)
      Site[sid].Tid := Site[sid].Tid + 1;
      IF (Debug = 3 AND (Debugt = 1 OR Debugt = 3)) THEN
        WriteTxt ("Local Transaction "); WriteCard(Site[sid].Tid);
        WriteTxt (" is created at site : "); WriteCard(sid); WriteLn();
      END;

      (* Create a transaction *)
      Site[sid].Tra[ntrans] := Thread.Fork(NEW(Tclosure, site := sid,
       seq := ntrans, id := Site[sid].Tid, type := "local",
       class := "local"));
    END;
  END;
END CREATETRANS;
```

CREATEDTRANS

This procedure will create a global transaction.

```
PROCEDURE CREATEDTRANS() =
VAR ntrans, min, i, j, coord, nsubt, temp, sno : CARDINAL;
    pick : ARRAY[1..Nsites] OF CARDINAL;
    flag : BOOLEAN;
    classt : TEXT;

BEGIN

 LOCK MUTEXCD DO

    FOR i := 1 TO Nsites DO
      pick[i] := 0;
    END;
    (* Keep on creating a global transaction until the maximum current global
      multiprogramming level is reached. *)
    WHILE (Dtrc < Dmpl ) DO

      Dtid := Dtid + 1; (* give a unique id *)
      Dtrc := Dtrc + 1; (* increment the count *)
      Dtri := Dtri + 1; (* give the index*)
      IF Dtri > Maxdt THEN
        Dtri := 1;
      END;
      Ndtrans := Ndtrans + 1;
      IF Ndtrans > Maxnt THEN
        Ndtrans := 1;
      END;
```

```
      min := Scount[1]; coord := 1;
      FOR i :=1 TO Nsites DO
          IF Scount[i] < min THEN
            min := Scount[i];
            coord := i;
          END;
      END;
      Scount[coord] := Scount[coord] + 1;
      (* pick the number of subtransactions randomely *)
      nsubt := Rnd.Subrange(Rnd.Default, 1, Nsites - 1);
      FOR i:=1 TO Nsites DO
        pick[i] := 0;
      END;

      (* pick the sites that the subtransactions would take place *)
      IF (nsubt= Nsites -1 ) THEN
        i := Nsites;
        FOR j := 1 TO Nsites DO
           pick[j] :=j;
        END;
       ELSE
        pick[nsubt+1] := coord;
        i := 0;
      END;

      WHILE (i < nsubt) DO
        temp := Rnd.Subrange(Rnd.Default, 1, Nsites);
        IF temp # coord THEN  flag := TRUE;
          ELSE flag := FALSE;
        END;
        FOR j := 1 TO i DO
           IF pick[j] = temp THEN
             flag := FALSE;
           END;
        END;
        IF flag THEN
          i:= i + 1;
          pick[i] := temp;
        END;
      END;
      (* Create the coordinator that is responsible for the
        subtransactions *)
      Coord[Ndtrans] := Thread.Fork(NEW (Cclosure,  site := coord,
      participant := pick,  nparticipant := nsubt,  id := Dtid,
      index := Dtri ));
    END; (* WHILE (Dtrc < Dmpl ) *)
  END;  (* LOCK MUTEXCD *)

END CREATEDTRANS;
```

CLOCK

This process will simulate a clock that will synchronize all sites

```
PROCEDURE CLOCK (self : Thread.Closure): REFANY =

BEGIN
 (* Create the transaction manager at each site *)
 FOR ii := 1 TO Nsites DO
    Transm[ii] := Thread.Fork(NEW(Tmclosure, id := ii));
 END;

 (* Create the throughput process *)
 Throughput := Thread.Fork(NEW(Thread.Closure, apply := THROUGHPUT));

 (* Create the site failure process *)
 Sitefail := Thread.Fork(NEW(Thread.Closure, apply := SITEFAILURE));

 (* Simulate an endless timer, until the maximum MPL is reached *)

 WHILE Mpl <= MaxMpl DO
   IF ((Ntime+1) * Maxns) < TRUNC(Timer + 0.01) THEN
     (* Ntime is used to calculate a rotatory timer, that
       will be useful in the throughput process. *)
     Ntime := Ntime + 1;
   END;
   Timer := Timer + 0.01;
   Yield(); (* Switch to the other processes *)
 END;
 RETURN NIL;
END CLOCK;
```

THROUGHPUT

This process is responsible for writing the results of the system into the corresponding files. Every certain number of iterations it will calculate the average results for one second and writes the results, i.e. if the run consists of n seconds at each Mpl, then this process will compute the metrics per 1 second in some cases, and in the other cases it will compute with respect to a transaction commit.

```
PROCEDURE THROUGHPUT(self:Thread.Closure):REFANY =


VAR
   (* Declaring all the local necessary variables. *)
   cou, Thru, thruo, Thrul, Thrud, thruol, thruod, dthru, dfail, nmessage : CARDINAL;
   dummy, ntime, i, count, countl, countd, dsitefail, deadlock, message  : CARDINAL;
   Comp, t1: CARDINAL;
   Ttime, Conflictr, Conflictrl, Conflictrd, avg, Resp, Respl, Respd, avgr : REAL;
   avgrd, avgrl, dummyr, dummyrl, dummyrd, avgd, dresponse, avga, avgpercent : REAL;
```

```
  avgsitefail, avgdeadlock, avgmessage, avgnmessage, tmessage, avgtm :REAL;
  avgcomp: REAL;

  (* Declare the files to write the results *)
  File, File1, File2, File3, File4, File5, File6, File7, File8, File9, File10 : Wr.T;
  File11 : Wr.T;
BEGIN
  Ttime := Timer;
  LongPause(Ttime, 5);  (* wait for 5 seconds until the system stabilises *)
  Ttime := Ttime + 5.0;
  cou := 1; ntime := Ntime;
  Lmpl := ( Mpl * Lpercent) DIV 100; (* compute the local MPL *)
  Dmpl := (Mpl * Dpercent) DIV 100; (* compute the global MPL *)
  File :=  OpenWrite(Outfile); (* Open the output file for throughput*)
  File1 := OpenWrite(Outfile1); (* Open the output file for conflict ratio*)
  File2 := OpenWrite(Outfile2); (* Open the output file for response time *)
  File3 := OpenWrite(Outfile3); (* Open the output file for global thruput*)
  File4 := OpenWrite(Outfile4); (* Open the output file for global abort *)
  File5 := OpenWrite(Outfile5); (* Open the output file for global response *)
  File6 := OpenWrite(Outfile6); (* Open the output file for site-failure *)
  File7 := OpenWrite(Outfile7); (* Open the output file for message-failure *)
  File8 := OpenWrite(Outfile8); (* Open the output file for deadlock *)
  File9 := OpenWrite(Outfile9); (* Open the output file for # messages *)
  File10 :=OpenWrite(Outfile10); (* Open the output file for % time in 2pc *)
  File11 :=OpenWrite(Outfile11); (*Open the output file for the # compensate*)

  (* Initializing the variables *)
  FOR dummy := 1 TO Nsites DO
    Site[dummy].NT := 0;
    Site[dummy].Conflict := 0;
  END;
  Thru := 0;  Resp := 0.0; Thrul := 0;  Thrud := 0; dthru := 0; dfail := 0;
  dresponse := 0.0; dsitefail := 0;  deadlock := 0; message := 0;
  nmessage := 0;  tmessage := 0.0; Ccmp := 0;

 (* while the maximum multiprogramming level is not exceeded *)
  WHILE Mpl <= MaxMpl DO


    (* If the number of calculation of the throughput for a mpl is  greater than the given
       number then calculate the average  throughput for that multiprogramming level
       and print the  results on the file as well as on the screen *)

    IF (cou > Nthr AND Mpl <= MaxMpl ) THEN
          (* Local Throughput *)
          avg := FLOAT(Thru)/( FLOAT(cou-1)* FLOAT(Nsites));
          (* global throughput *)
          avgd := FLOAT(dthru)/(FLOAT(cou -1));
          (* global abort *)
          avga := FLOAT(dfail)/(FLOAT(cou -1));
          (* Avg # of compensating transactions on each site *)
          avgcomp := FLOAT(Comp)/( FLOAT(cou-1)* FLOAT(Nsites));
          IF dthru # 0 THEN
            (* average global response time per commit*)
```

```
  avgrd := dresponse/FLOAT(dthru);
  (* average site fail per second *)
  avgsitefail := FLOAT(dsitefail)/FLOAT(cou - 1);
  (* average deadlock per second *)
  avgdeadlock := FLOAT(deadlock)/FLOAT(cou - 1);
  (* avg message failure per second *)
  avgmessage := FLOAT(message)/FLOAT(cou - 1);
  (* avg number of messages per commit *)
  avgnmessage := FLOAT (nmessage)/FLOAT(dthru);
  (* percentage of time spent in the 2pc *)
  avgtm := tmessage/FLOAT(dthru);
  avgpercent := (avgtm/avgrd)*100.0;
 END;
dresponse := 0.0; dsitefail := 0; deadlock := 0; message := 0;
nmessage := 0; tmessage := 0.0;
 IF Thru # 0 THEN
  avgr := Resp / FLOAT (Thru); (* Response Time *)
  ELSE
  avgr := 0.0;
 END;
(* local response time *)
 IF Thrul # 0 THEN
  avgrl := Respl / FLOAT(Thrul);
  ELSE
  avgrl := 0.0;
 END;
dummy := 0; dummyr := 0.0; dummyrl := 0.0; dummyrd := 0.0;
 FOR i := 1 TO Nsites DO
  (* calculate the conflict ratio *)
  IF Site[i].NT # 0 THEN
   dummyr := (FLOAT(Site[i].Conflict)/
          FLOAT(Site[i].NT)) + dummyr;
  END;
 END;
Conflictr := dummyr / FLOAT(Nsites); (* Conflict ratio *)
Thru := 0; Resp := 0.0; dthru := 0; dfail := 0; Comp := 0;
WriteLn();
WriteTxt("###The average local throughput for Mpl ");
WriteCard(Mpl); WriteTxt(" is = ");
WriteReal(avg); WriteLn();
WriteTxt("The global throughput for Mpl :");
WriteCard(Dmpl); WriteTxt(" is = "); WriteReal(avgd); WriteLn();
WriteTxt("The global abort :");
WriteReal(avga); WriteLn();
WriteTxt("The Conflict ratio : ");
WriteReal(Conflictr);
WriteTxt(" The local Response time : ");
WriteReal(avgrl); WriteLn();
WriteTxt(" The global Response time : ");
WriteReal(avgrd); WriteLn();
WriteTxt(" The site-failure per second is : ");
WriteReal(avgsitefail); WriteLn();
WriteTxt(" The message-failure per second is : ");
WriteReal(avgmessage); WriteLn();
```

```
WriteTxt(" The deadlock per second is: ");
WriteReal(avgdeadlock); WriteLn();
WriteTxt(" The number of messages per transaction commit: ");
WriteReal(avgnmessage); WriteLn();
WriteTxt(" The percentage of time spent in 2pc is: ");
WriteReal(avgpercent); WriteLn();
WriteTxt(" The average # of compensating transactions: ");
WriteReal(avgcomp); WriteLn();


(* write the average local throughput results *)
WRITE(Mpl, avg, File);

(* write the conflict ratio results *)
WRITE(Mpl, Conflictr, File1);

(* write the local response time results *)
WRITE(Mpl, avgrl, File2);

(* Write the global transaction results *)
(* write the throughput results *)
WRITE(Dmpl, avgd, File3);

(* write the abort results *)
WRITE(Dmpl, avga, File4);

(* write the response results *)
WRITE(Dmpl, avgrd, File5);

(* write the site-failure results *)
WRITE(Dmpl, avgsitefail, File6);

(* write the messaage-failure results *)
WRITE(Dmpl, avgmessage, File7);

(* write the deadlock results *)
WRITE(Dmpl, avgdeadlock, File8);

(* write the number of messages results *)
WRITE(Dmpl, avgnmessage, File9);

(* write the percent of time spent in 2pc *)
WRITE(Dmpl, avgpercent, File10);

(* write the average # of compensating transactions *)
WRITE(Dmpl, avgcomp, File11);

cou := 1;
FOR dummy := 1 TO Nsites DO
  Site[dummy].NT := 0;
  Site[dummy].Conflict := 0;
END;
Mpl := Mpl + Smpl;
Lmpl := ( Mpl * Lpercent) DIV 100;
```

```
      Dmpl := (Mpl * Dpercent) DIV 100;

 END; (* IF *)
 LongPause(Ttime, Thrt); (* Pause for a certain number of seconds *)
 dummy := TRUNC(Ttime) - (Maxns * ntime);

(* ACCUMULATE THE RESULTS INTO LOCAL VARIABLES *)
FOR i :=1 TO Nsites DO
  Thru := Site[i].Thruput[dummy] + Thru;
  Thrul := Site[i].THRUPUT[1, dummy] + Thrul;
  t1  := Site[i].Compensate[dummy];
  Comp  := Site[i].Compensate[dummy] + Comp;

  IF Protocol = 3 THEN
    IF t1 < Thru THEN
       Thru := Thru - t1;
     ELSE
      WriteTxt("Error 1 ");
      WriteCard(t1); WriteTxt(" ");
      WriteCard(Thru); WriteLn();
      Thru := 0;
    END;
    IF t1 < Thrul THEN
      Thrul := Thrul -t1;
     ELSE
      WriteTxt("Error 2 "),
      WriteCard(t1); WriteTxt(" ");
      WriteCard(Thrul); WriteLn();
      Thrul := 0;
    END;
  END;
  Thrud := Site[i].THRUPUT[2, dummy] + Thrud;
  thruo :=  Thru + thruo;
  Resp := Site[i].Response[dummy] + Resp;
  Respl := Site[i].RESPONSE[1, dummy] + Respl;
  Respd := Site[i].RESPONSE[2, dummy] + Respd;
END;
dthru := Dthru[dummy] + dthru;
dfail := CouFail[dummy] + dfail;
dresponse := Dresponse[dummy] + dresponse;
dsitefail := Failure.site[dummy] + dsitefail;
deadlock := Failure.deadlock[dummy] + deadlock;
message := Failure.message[dummy] + message;
nmessage := Dmessage[dummy] + nmessage;
tmessage := Dmresponse[dummy] + tmessage;

Ttime := Ttime + FLOAT(Thrt);
ntime := Ntime;
IF (Debug # 0) THEN
  (* WriteLongReal((r1-r)/1000000.0d+00); WriteLn(); *)
  WriteLn(); WriteTxt("***The average local Throughput is : ");
  WriteCard(thruo DIV Nsites);
  WriteLn(); WriteTxt("The global Throughput is : ");
  WriteCard(Dthru[dummy]); WriteLn();
```

```
      WriteTxt("The number of global failures = ");
      WriteCard(CouFail[dummy]); WriteLn();
      count := 0; countl :=0;  countd := 0;
      FOR i := 1 TO Nsites DO
        count  := Site[i].Trc   +  count;
        countl := Site[i].TRC[1] + countl;
        countd := Site[i].TRC[2] + countd;
      END;
      WriteTxt("The number of remaining local transactions: ");
      WriteCard(count DIV Nsites); WriteLn();
      WriteTxt("The number of remaining global transactions: ");
      WriteCard(Dtrc); WriteLn();

    END;
    FOR i := 1 TO Nsites DO
       Site[i].Thruput[dummy] := 0;
       Site[i].Compensate[dummy] := 0;
       Site[i].THRUPUT[1, dummy] := 0;
       Site[i].THRUPUT[2, dummy] := 0;
       Site[i].Response[dummy] := 0.0;
       Site[i].RESPONSE[1, dummy] := 0.0;
       Site[i].RESPONSE[2, dummy] := 0.0;
    END;
    Dthru[dummy] := 0;
    CouFail[dummy] := 0;
    Dresponse[dummy] := 0.0;
    Failure.site[dummy] := 0;
    Failure.deadlock[dummy] := 0;
    Failure.message[dummy] := 0;
    Dmessage[dummy] := 0;
    Dmresponse[dummy] := 0.0;

    cou := cou + 1;
  END;
  Wr.Close(File);
  Wr.Close(File1);
  RETURN NIL;
END THROUGHPUT;
```

WRITE

This procedure will save the information on a file.

```
PROCEDURE WRITE(mpl : CARDINAL ; res : REAL; File : Wr.T) =

BEGIN
 Wr.PutText(File, Fmt.Int(mpl));
 Wr.PutText(File, " ");
 Wr.PutText(File, Fmt.Real(res));
 Wr.PutText(File, "\n");
 Wr.Flush(File);
END WRITE;
```

TM

The transaction manager process is responsible for creating the transactions and forwarding them to the Scheduler.

```
PROCEDURE TM(self:Tmclosure):REFANY =
VAR i, cs : CARDINAL;

BEGIN
 cs := self.id;
 (* This loop will create number of transactions until the Mpl is reached *)
 WHILE Mpl <= MaxMpl DO (* While the Mpl is less than the MaxMpl *)
  Yield();
  CREATETRANS(cs); (* create a local transaction *)
  CREATEDTRANS(); (* create a global transaction *)
 END;
 RETURN NIL;
END TM;
```

SCHEDULER

The Scheduler is responsible for the completion of the transaction, conflicts and deadlocks are encountered in this process.

```
PROCEDURE SCHEDULER(self:Tclosure):REFANY =
VAR i, Prs, delay, dummy , sid, type, Pc, Rndm, PD2, stepf, idf: CARDINAL;
   Trans : Transaction;
   flagd, flagp, flagc : BOOLEAN;
   Ttime, Tr, Stime, elapsed, stime1 : REAL;
   typep : TEXT;

BEGIN

 sid := self.site;
 IF self.type = "local" THEN
    type := 1;
    typep := "Local Transaction ";
    idf := self.id;
  ELSE
    type := 2;
    typep := "Subtransaction ";
    idf := self.did;
 END;
 Ttime := Timer; (* Assign the current time to transaction time *)
 Stime := Ttime; (* Save the transaction start time *)


 (*** CHECK FAIL ***)
 IF Site[sid].Fail = "Y" THEN
```

```
  WAITRECOVER(sid, typep, idf, Ttime);
END;

Trans.id := self.id;
Trans.rw := Trmax;
(* Pick up a random number between Trmin Trmax, this random
  number will be the number of actions of the transaction *)
IF Transt = 1 THEN (* If the transaction is of variable type *)
  LOCK Site[sid].mutex DO
   Trans.rw := Rnd.Su rrange(Rnd.Default, Trmin, Trmax);
  END;
END;
(* if the transaction is a global one then pick the number of r/w sets *)
IF type = 2 THEN
  LOCK Site[sid].mutex DO
   Trans.rw := Rnd.Subrange(Rnd.Default, Trmin DIV 3, Trmax * 2 DIV 3);
  END;
END;

IF (Debug = 3 AND (Debugt = type OR Debugt = 3)) THEN
 LOCK Moutput DO
   WriteTxt("Site "); WriteCard(sid); WriteTxt(" :");
   WriteTxt (typep); WriteCard(self.id);
   WriteTxt( " started time:");
   WriteReal(Ttime); WriteTxt(" # r/w actions= ");
   WriteCard(Trans.rw); WriteTxt(", # transactions=");
   WriteCard(Site[sid].Trc); WriteLn();
  END;
END;
(* The probability of a deadlock of cycle 2 *)
PD2 := PDEADLOCK(sid );
(* Pick a random number to decide at which step the deadlock *)
(* is going to happen *)
LOCK Site[sid].mutex DO
   Prs := Rnd.Subrange(Rnd.Default, 1, Trmax);
END;
Pause(Ttime, Extthink); (* ext_think_time *)
Ttime := Ttime + ( FLOAT(Extthink)/Unit) ;

=====================
PERFORM THE STEPS
=====================
flagd := FALSE;
FOR i := 1 TO Trans.rw
DO

 (*** CHECK FAIL ***)
 IF Site[sid].Fail = "Y" THEN
  WAITRECOVER(sid, typep, idf, Ttime);
 END;

 (* The probability of a lock conflict Pc *)
 Pc := PCONFLICT(sid);
 LOCK Site[sid].mutex1 DO
```

```
  Rndm := Rnd.Subrange(Rnd.Default, 1, 10000);

 (* Rndm := Rnd.Real(); *)
END;
(* If there is a deadlock *)
IF ( Prs = i AND  Rndm < PD2) THEN
  delay := DELAY();
  Pause(Ttime, delay); (* Pause *)
  Ttime := Ttime + (FLOAT(delay)/Unit);
  IF (Debug # 0 AND (Debugt = type OR Debugt = 3)) THEN
    Mes3(sid,  Trans.id, Ttime, typep, "Deadlock aborting");
  END;
  flagd := TRUE; (* set the flag of deadlock to true *)
  (* If this is a subtransaction record a deadlock in its log *)
  IF type = 2 THEN
    Log[self.index, sid].deadlock := "Y";
    (* Increment the deadlock *)
    dummy := Second(Ttime);
    LOCK MUTEXF DO
      Failure.deadlock[dummy] := Failure.deadlock[dummy] + 1;
    END;
  END;
  Site[sid].NT := Site[sid].NT - 1;
  Site[sid].NTRANS[type] := Site[sid].NTRANS[type] - 1;
  EXIT; (* Abort *)
END;

(* If there is a lock conflict *)
IF Rndm < Pc THEN
  Site[sid].Conflict := Site[sid].Conflict + 1;
  Site[sid].CONFLICT[type] := Site[sid].CONFLICT[type] + 1;
  IF ( (Debug = 2 OR Debug = 3) AND (Debugt = type OR Debugt = 3)) THEN
   LOCK Moutput DO
     WriteTxt("???Site "); WriteCard(sid);
     WriteTxt(" :Conflict,  "); WriteTxt(typep); WriteTxt(" ");
     WriteCard(Trans.id);
     WriteTxt(" is blocked,  at time "); WriteReal(Ttime); WriteLn();
     WriteTxt ("Number of transactions :"); WriteCard(Site[sid].Trc);
     WriteLn();
   END;
  END;

  (* block the transaction until the lock is acquired *)
    delay := DELAY();
    Pause(Ttime, delay) ;
    Ttime := Ttime + (FLOAT(delay) / Unit);
  (* Release the transaction *)
  IF ((Debug = 2 OR Debug = 3) AND (Debugt = type OR Debugt = 3)) THEN
    Mes3(sid,  Trans.id, Ttime, typep, "has resumed its execution");
  END;
END;
(* Execute the step *)

Pause(Ttime, Stept - Diskaccess - Cpuaccess);
```

```
Ttime := Ttime + (FLOAT(Stept - Diskaccess - Cpuaccess) / Unit);
IF (Debug = 3 AND (Debugt =type OR Debugt = 3))THEN
   Mes3(sid, Trans.id, Ttime, typep, "is currently executing the step");
END;

(* ACCESS THE DISK *)
Tr := Ttime;
flagp := FALSE;
IF (Site[sid].Nudisk + 1) > Maxdisk THEN
  flagp := TRUE;
  IF ((Debug = 2 OR Debug = 3) AND (Debugt = type OR Debugt = 3)) THEN
    Mes3(sid, Trans.id, Ttime, typep,
       "is currently blocked at a Disk req");
  END;
END;

WHILE (Site[sid].Nudisk + 1) > Maxdisk DO
  Yield();
END;
IF (flagp AND (Debug =2 OR Debug = 3)AND(Debugt = type OR Debugt = 3))
 THEN
  LOCK Moutput DO
  WriteTxt("Site "); WriteCard(sid);
  WriteTxt(typep); WriteCard(Trans.id);
  WriteTxt(" has obtained the disk req, Current time= ");
  WriteReal(Timer); WriteTxt(" Waiting time= ");
  WriteReal(Timer - Tr); WriteLn();
  END;
END;
Ttime := Ttime + (Timer - Tr);
LOCK Site[sid].mutex3 DO
  Site[sid].Nudisk := Site[sid].Nudisk + 1;
END;
Pause(Ttime, Diskaccess); (* ACCESS THE DISK *)
Ttime := Ttime + (FLOAT(Diskaccess)/Unit);
LOCK Site[sid].mutex3 DO
  Site[sid].Nudisk := Site[sid].Nudisk - 1;
END;

(* ACCESS THE CPU *)
Tr := Ttime;
flagp := FALSE;
IF (Site[sid].Nucpu + 1) > Maxcpu THEN
  flagp := TRUE;
  IF ((Debug =2 OR Debug = 3)AND (Debugt = type OR Debugt = 3)) THEN
    Mes3(sid, Trans.id, Ttime, typep,
          "is currently blocked at a Cpu req");
  END;
END;

WHILE (Site[sid].Nucpu + 1)> Maxcpu DO
 Yield();
END;
IF (flagp AND (Debug =2 OR Debug =3 )AND (Debugt = type OR Debugt = 3))
```

```
  THEN
   LOCK Moutput DO
    WriteTxt("Site "); WriteCard(sid);
    WriteTxt(typep); WriteCard(Trans.id);
    WriteTxt(" has obtained the cpu req, Current time= ");
    WriteReal(Timer); WriteTxt(" Waiting time= ");
    WriteReal(Timer - Tr); WriteLn();
   END;
  END;

  Ttime := Ttime + (Timer - Tr);

  LOCK Site[sid].mutex3 DO
    Site[sid].Nucpu := Site[sid].Nucpu + 1;
  END;
  Pause(Ttime, Cpuaccess); (* ACCESS THE CPU *)
  Ttime := Ttime + (FLOAT(Cpuaccess)/Unit);
  LOCK Site[sid].mutex3 DO
    Site[sid].Nucpu := Site[sid].Nucpu - 1;
  END;

  IF ( Debug = 3 AND (Debugt =type OR Debugt = 3)) THEN
   LOCK Moutput DO
    WriteTxt("Site "); WriteCard(sid); WriteTxt(" : ");
    WriteTxt(typep); WriteCard(Trans.id);
    WriteTxt(" has finished step # "); WriteCard (i);
    WriteTxt(" time :"); WriteReal(Ttime); WriteLn();
   END;
  END;
END; (* FOR *)

*****************************************************************************
 IF THE TRANSACTION IS A SUBTRANSACTION CALL THE VOTE AND COMMIT
 PROCEDURES.
*****************************************************************************
flagc := TRUE;
IF (type = 2 ) THEN
    stime1 := Ttime;
    IF Log[self.index, sid].commit_decision # "N" THEN
       PARTICIPANT(sid, self.index, seif.did, self.nparticipant,
       self.coordinator, Ttime, self.participant);
    END;
    IF  (Log[self.index, sid].commit_decision = "N" OR
        Log[self.index, sid].vote_decision = "N" OR
        Log[self.index, sid].vote_decision = "U") THEN
        flagc := FALSE;
        Mes1(sid, self.index, Ttime, "is ABORTING");
      ELSE
        Mes1(sid, self.index, Ttime, "is COMMITING");
        stime1 := Ttime - stime1;
        dummy := Second(Ttime);
        (* store the time spent in 2pc *)
        LOCK MUTEXM DO
          Dmresponse[dummy] := Dmresponse[dummy] +
```

```
                    (stime1/FLOAT(self.nparticipant));
      END;
    END;
END;

(* Decrement the number of transactions *)
IF (Protocol # 3 OR type = 1) THEN
  Site[sid].Trc := Site[sid].Trc - 1;
END;
(* Decrment the number of distributed or local transactions *)
Site[sid].TRC[type] := Site[sid].TRC[type] - 1;
(* If the transaction was not aborted then commit it *)
IF ((NOT flagd) AND flagc) THEN
 (* Increment the number of finished transactions *)
  Site[sid].NT  := Site[sid].NT + 1;
  Site[sid].NTRANS[type] := Site[sid].NTRANS[type] + 1;

  dummy := Second(Ttime);
  (* Increment the system throughput *)
  Site[sid].Thruput[dummy] := Site[sid].Thruput[dummy] + 1;
  (* Increment the throughput of the transaction type *)
  Site[sid].THRUPUT[type, dummy] := Site[sid].THRUPUT[type, dummy] + 1;
  elapsed := Ttime - Stime;
  Site[sid].Response[dummy] := Site[sid].Response[dummy] + elapsed;
  Site[sid].RESPONSE[type, dummy] := Site[sid].RESPONSE[type, dummy] +
   elapsed;
  IF ((Debug = 2 OR Debug = 3)AND (Debugt =type OR Debugt = 3))  THEN
     Mes3(sid,  Trans.id, Ttime, typep, "finished its execution");
  END;
END;
IF (type = 2 ) THEN
   MESSAGEDELAY(sid, self.coordinator, Ttime);
   Log[self.index, sid].finished := "Y";
END;
CREATETRANS(sid);
CREATEDTRANS();

RETURN NIL;

END SCHEDULER;
```

PARTICIPANT

In this procedure the participant synchronizes the two phase commit with its coordinator.

```
PROCEDURE PARTICIPANT(sid, ind, did, np, coord : CARDINAL;  VAR Ttime : REAL;
            pick : ARRAY[1..Nsites] OF CARDINAL)=

VAR
 flagc : BOOLEAN;
BEGIN
```

```
(* Call the vote procedure *)
VOTE(sid, ind, did, coord, Ttime);
(* IF it is optimistic 2PC then release the locks *)
IF Protocol = 3 THEN
  LOCK Site[sid].mutex4 DO
     Site[sid].Trc := Site[sid].Trc - 1;
  END;
END;

(* If the transaction vote is yes the call the commit procedure *)
IF (Log[ind, sid].vote_decision # "N" AND Log[ind, sid].deadlock # "Y") THEN
  COMMIT(sid, ind, did, np, coord, Ttime, pick);
END;

END PARTICIPANT;
```

VOTE

In this procedure the transaction waits for a vote_request from the coordinator and then sends its vote-decision.

```
PROCEDURE VOTE(sid, ind, did, coord : CARDINAL; VAR Ttime : REAL)=
VAR time : REAL;
   flagf : BOOLEAN;
   timeout : CARDINAL;

BEGIN
  flagf := FALSE;
  (* Wait for the vote request *)
  Log[ind, sid].location := "WV"; (* mark the location of the participant *)
  IF Log[ind, sid].vote_request = "U"  THEN
    Mes1(sid, did, Ttime, "is waiting for the vote_request");
  END;
  time := Ttime;
  (* Timeout if the vote is not received *)
  timeout := TIMEOUT(Timeout1);
  WHILE ((Log[ind, sid].vote_request = "U" ) AND
       (Timer <(time + (FLOAT(timeout)/Unit)))) DO
    Ttime := Timer;
    (*** CHECK FAIL ***)
    IF Site[sid].Fail = "Y" THEN
      WAITRECOVER(sid, "Subtransaction", did, Ttime);
      flagf := TRUE;
    END;
    Yield();
  END;

  (**************P2PC*******************************)
  time := Ttime;
```

```
(* If it did not receive the vote_request and the protocol used is P2PC
   then give another chance *)

IF (Log[ind, sid].vote_request = "U" AND Protocol = 2) THEN
  timeout := TIMEOUT(Timeout1);
  WHILE ((Log[ind, sid].vote_request = "U" ) AND
        (Timer <(time + (FLOAT(timeout)/Unit)))) DO
    Ttime := Timer;
    (*** CHECK FAIL ***)
    IF Site[sid].Fail = "Y" THEN
      WAITRECOVER(sid, "Subtransaction", did, Ttime);
      flagf := TRUE;
    END;
    Yield();
  END;
END;
(*****************************************************************)

(* If the transaction did not receive the vote_request or a failure
  occured then abort it *)
IF Log[ind, sid].vote_request = "U" OR flagf THEN
  Mes1(sid, did, Ttime, "did not receive the vote_request");
  Log[ind, sid].vote_decision := "N";
  RETURN;
END;

Mes1(sid, did, Ttime , "has received the vote_request");

(* send your vote *)
(*** CHECK FAIL ***)
IF Site[sid].Fail = "Y" THEN
  WAITRECOVER(sid, "Subtransaction", did, Ttime);
END;

IF Log[ind, sid].deadlock = "N" THEN
  Mes1(sid, did, Ttime, "is sending the vote YES");
  MESSAGEDELAY(sid, coord, Ttime);
  (*** CHECK FAIL ***)
  IF Site[sid].Fail = "Y" THEN
    WAITRECOVER(sid, "Subtransaction", did, Ttime);
  END;
  IF (NOT MESSLOST() OR sid = coord) THEN
    Log[ind, sid].vote_decision := "Y";
    ELSE
     Mes7(sid, did, Ttime, "MESSAGE FAILURE IN VOTE Y DECISION");
  END;
 ELSE
  Mes1(sid, did, Ttime, "is sending the vote NO(DEADLOCK)");
  MESSAGEDELAY(sid, coord, Ttime);
  (*** CHECK FAIL ***)
  IF Site[sid].Fail = "Y" THEN
    WAITRECOVER(sid, "Subtransaction", did, Ttime);
  END;
  IF (NOT MESSLOST() OR sid = coord )THEN
```

```
      Log[ind, sid].vote_decision := "N";
      ELSE
       Mes7(sid, did, Ttime, "MESSAGE FAILURE IN VOTE N DECISION");
     END;
    END;

END VOTE;
```

COMMIT

In this procedure the participant waits for a commit_decision from the coordinator and then does the appropriate action.

```
PROCEDURE COMMIT(sid, ind, did, np, coord : CARDINAL;  VAR Ttime : REAL;
          pick : ARRAY[1..Nsites] OF CARDINAL) =
VAR time : REAL;
   flagf : BOOLEAN;
   terms, timeout : CARDINAL;
BEGIN
  flagf := FALSE;
  IF (Log[ind, sid].commit_decision = "U") THEN
    Mes1(sid, did, Timer, "is waiting for the decision");
  END;
  (* Mark the location of the participant *)
  Log[ind, sid].location := "WC";
  (* Timeout if the commit decision is not received *)
  timeout := TIMEOUT(Timeout2);
  time := Ttime;
  WHILE ((Log[ind, sid].commit_decision = "U") AND
       (Timer <(time + (FLOAT(timeout)/Unit)))) DO
   Ttime := Timer;
   (*** CHECK FAIL ***)
   IF Site[sid].Fail = "Y" THEN
     WAITRECOVER(sid, "Subtransaction", did, Ttime);
     flagf := TRUE;
   END;
   Yield();
  END;
  Ttime := Timer;

 (****************P2PC*******************************)

  (* If a decision was not reached give the coord another chance *)
  IF (Log[ind, sid].commit_decision = "U" AND Protocol = 2 ) THEN
    (* send your vote another time *)
    Mes1(sid, did, Ttime, "is sending the vote YES again");
    MESSAGEDELAY(sid, coord, Ttime);
    (*** CHECK FAIL ***)
    IF Site[sid].Fail = "Y" THEN
     WAITRECOVER(sid, "Subtransaction", did, Ttime);
    END;
    IF (NOT MESSLOST() OR sid = coord) THEN
```

```
    Log[ind, sid].vote_decision := "Y";
  END;

  (* wait for the decision another time *)
  timeout := TIMEOUT(Timeout2);
  time := Ttime;
  WHILE ((Log[ind, sid].commit_decision = "U") AND
      (Timer <(time + (FLOAT(timeout)/Unit)))) DO
    Ttime := Timer;
    (*** CHECK FAIL ***)
    IF Site[sid].Fail = "Y" THEN
      WAITRECOVER(sid, "Subtransaction", did, Ttime);
      flagf := TRUE;
    END;
    Yield();
  END;
END;
Ttime := Timer;
*************************************************************)


(* If a decision was not reached or a failure was detected
   Then call termination protocol *)
IF (Log[ind, sid].commit_decision = "U" ) THEN
  Mes1(sid, did, Ttime, "Calling termination protocol");
  terms := TERMINATE(sid, ind, np, pick, Ttime);
  CASE terms OF
    | 1 => Log[ind, sid].vote_decision := "N";
         Mes4(sid, did, Ttime, "TP decided to abort #1");
         RETURN;

    | 2 => Mes4(sid, did, Ttime, "TP decided to block");


    | 3 => Log[ind, sid].commit_decision := "Y";
         Log[ind, sid].location := "CS";
         Mes4(sid, did, Ttime, "TP decided to commit");
         RETURN;

    | 4 => Log[ind, sid].commit_decision := "N";
         Log[ind, sid].location := "AS";
         Mes4(sid, did, Ttime, "TP decided to abort #2");

     ***********************************************************
                              O2PC
       If The protocol is O2PC THEN do a compensating transaction
     ***********************************************************
         IF Protocol = 3 THEN
           COMPENSATE(sid, Ttime);
         END;
         RETURN;
  ELSE
     WriteTxt("ERROR in TERMINATE"); WriteLn();
  END;
```

```
    Yield();
  END;
  (* IF all the participants are in the wait state then block waiting
    for the commit decision *)
  WHILE (Log[ind, sid].commit_decision = "U" ) DO
      Yield();
  END;
  IF Log[ind, sid].commit_decision = "Y" THEN
    Log[ind, sid].location := "CS";
    Mes1(sid, did, Ttime, "has received commit decision");
   ELSE
    Log[ind, sid].location := "AS";
    Mes1(sid, did, Ttime, "has received abort decision");
    (* If The protocol is O2PC THEN do a compensating transaction *)
    IF Protocol = 3 THEN
      COMPENSATE(sid, Ttime);
    END;
  END;

END COMMIT;
```

COMPENSATE

This procedure will do a compensating transaction, that will be physically deducting a transaction from the number of finished transaction, the semantic overhead for compensating transactions are not represented in this program, the only cost is an additionnal local transaction.

```
PROCEDURE COMPENSATE(sid : CARDINAL; Ttime:REAL)=
VAR dummy : CARDINAL;
BEGIN
   dummy := Second(Ttime);
   (* Increment the number of compensating transactions *)
   LOCK Site[sid].mutex4 DO
      Site[sid].Compensate[dummy] := Site[sid].Compensate[dummy] + 1;
   END;
   WriteTxt("Compensate is performed");
   WriteLn();
END COMPENSATE;
```

TERMINATE

This procedure will send a message to other participants asking them if they have received a vote decision or not, and according to their reply the participant will decide either to abort or commit or block waiting for the decision.

```
PROCEDURE TERMINATE(sid, ind, np : CARDINAL; pick:ARRAY[1..Nsites] OF
CARDINAL;   VAR Ttime: REAL):CARDINAL =
VAR i, loc : CARDINAL;
   lc : TEXT;
BEGIN
```

```
  BROADCASTDELAY(sid, np, Ttime, pick); (* broadcast the request *)
  BROADCASTDELAY(sid, np, Ttime, pick); (* wait for the answers *)
  loc := 1;
  FOR i := 1 TO np DO
    lc := Log[ind, pick[i]].location;
    IF lc = "PP" THEN
      RETURN 1;
    END;
    IF lc = "WV" THEN
      RETURN 1;
    END;
    IF lc = "WC" AND pick[i] # sid THEN
      loc := 2;
    END;
    IF lc = "CS" THEN
      RETURN 3;
    END;
    IF lc = "AS" THEN
      RETURN 4;
    END;
  END;
  IF loc = 1 THEN
    WriteTxt("TERMINATE UNKNOWN"); WriteLn();
  END;
  RETURN loc;
END TERMINATE;
```

COORDINATOR

This process is the coordinator that is responsible of the atomic commitment of the global transaction. This process coordinates according to the 2PC protocol, first it sends vote requests, then when it receives the vote decisions, it decides and sends the decision to the participants, and it waits for their acknowledgments.

```
PROCEDURE COORDINATOR(self : Cclosure) : REFANY =
VAR
  nsubt, coord, sno, ntrans, ind, id, timeout, dummy, cou: CARDINAL;
  pick, nack : ARRAY[1..Nsites] OF CARDINAL;
  classt : TEXT;
  ctime, time, stime, elapsed : REAL;
  flagv, flagc, flage : BOOLEAN;

BEGIN
    (* Initialze the local variables *)
    nsubt := self.nparticipant +1;
    coord := self.site;
    pick := self.participant;
    ind := self.index;
    id := self.id;
    ctime := Timer;
    stime := ctime;
    (** SF **)
```

```
SITEFAIL(coord, id, ctime); (* SEE IF SITE has failed *)

(* Initialize the log record to undecidable before creating the
  transactions*)
FOR i := 1 TO Nsites DO
   Log[ind, i].vote_decision := "U";
   Log[ind, i].commit_decision := "U";
   Log[ind, i].vote_request := "U";
   Log[ind, i].finished := "N";
   Log[ind, i].deadlock := "N";
   Log[ind, i].site_fail := "N";
   Log[ind, i].location := "PP";
END;

(* Creating the subtransactions at all the partipating
  sites *)
IF (Ddebug = 1) THEN
  LOCK Moutput DO
   WriteTxt("Creating subtransactions at sites:");
   FOR i := 1 TO nsubt  DO
    WriteCard(pick[i]); WriteTxt(", ");
   END;
   WriteTxt(".Coord is site:");  WriteCard(coord); WriteTxt(", ID#");
   WriteCard(self.id);
   WriteLn();
  END;
END;

FOR i := 1 TO nsubt  DO
  sno := pick[i];
  IF sno = coord THEN
    classt := "coordinator";
   ELSE
    classt := "participant";
  END;
  LOCK Site[sno].mutex4 DO
    Site[sno].Ntrans := Site[sno].Ntrans + 1;
    IF Site[sno].Ntrans >  Maxnt THEN
      Site[sno].Ntrans := 1;
    END;
    ntrans := Site[sno].Ntrans;
    Site[sno].Trc := Site[sno].Trc + 1;
    Site[sno].TRC[2] := Site[sno].TRC[2] + 1;
    Site[sno].Tid := Site[sno].Tid + 1;
    (* Creating each subtransaction at its site *)
    Site[sno].Tra[ntrans] := Thread.Fork(NEW(Tclosure,
    site := sno,  seq := ntrans, id := Site[sno].Tid,
    type := "distributed",  class := classt,  did := self.id,
    coordinator := coord;  index := ind,  participant := pick,
    nparticipant := nsubt));
  END;
END;

ctime := Timer;
```

```
flagv := FALSE;
stime := ctime;

(* WAIT BEFORE SENDING THE VOTE REQUEST *)
Mes2(self.id, ctime, "is waiting before sending a vote_request");
Cpause := TIMEOUT(Timeout)DIV 10;
Pause(ctime, Cpause);
ctime := ctime + (FLOAT(Cpause)/Unit);

---------------------------------
 Send Vote requests
---------------------------------
(* CHECK IF SITE OF COORDINATOR HAS FAILED *)
IF Site[coord].Fail = "Y" THEN
  SITEFAIL(coord, id, ctime);  (* block the coordinator *)
END;

(* send the vote request *)
Mes2(self.id, ctime, "is sending a vote_request");
BROADCASTDELAY(coord, nsubt, ctime, pick);
(* CHECK IF SITE OF COORDINATOR HAS FAILED *)
IF Site[coord].Fail = "Y" THEN
  SITEFAIL(coord, id, ctime);  (* block the coordinator *)
END;

FOR i := 1 TO nsubt DO
   IF (NOT MESSLOST() OR pick[i] = coord) THEN
     Log[ind, pick[i]].vote_request := "Y";
    ELSE
     Mes6(id, pick[i], ctime, "MESSAGE FAILURE IN VOTE_REQUEST");
    END;
END;

(** SF **)
SITEFAIL(coord, id, ctime); (* SEE IF SITE WILL FAIL *)



---------------------------------
      PHASE 1
 Wait that all sites send their
 vote decision
---------------------------------
Mes2(self.id, ctime, "is waiting for vote decisions");
time := ctime;
timeout:= TIMEOUT(Timeout);  (* call the timeout according to Mpl*)
WHILE (Timer < (time + (FLOAT(timeout)/Unit))) DO
    flagv := TRUE;
    FOR i := 1 TO nsubt DO
      IF Log[ind, pick[i]].vote_decision = "U" THEN
         flagv := FALSE;
       END;
    END;
    ctime := Timer;
```

```
    IF flagv THEN
      Mes2(self.id, ctime, "Has received all the votes");
      EXIT;
    END;
    (* CHECK IF SITE OF COORDINATOR HAS FAILED *)
    IF Site[coord].Fail = "Y" THEN
      SITEFAIL(coord, id, ctime); (* block the coordinator *)
    END;

    Yield();
  END;
  IF NOT flagv THEN

    cou := 0;
    FOR i := 1 TO nsubt DO
      IF Log[ind, pick[i]].vote_decision = "U" THEN
        cou := cou + 1;
        nack[cou] := pick[i];
      END;
    END;

   IF Ddebug = 1 THEN
     LOCK Moutput DO
     WriteTxt("Coordinator #"); WriteCard(self.id);
     WriteTxt("->did not receive votes from sites:");
     FOR i := 1 TO nsubt DO
       IF Log[ind, pick[i]].vote_decision = "U" THEN
         WriteCard(pick[i]); WriteTxt(", ");
       END;
     END;
     WriteTxt("Time:"); WriteReal(ctime); WriteLn();
     END;
   END;
  END;
**********P2PC **********
 give another chance by resending the vote-requests to
 the appropriate sites, and by giving another timeout.
*************************************************
 IF (NOT flagv AND Protocol = 2) THEN
   (* send the vote-requests *)
   Mes2(self.id, ctime, "is resending the vote_requests");
   BROADCASTDELAY(coord, cou, ctime, nack);
   (* CHECK IF SITE OF COORDINATOR HAS FAILED *)
   IF Site[coord].Fail = "Y" THEN
     SITEFAIL(coord, id, ctime); (* block the coordinator *)
   END;
   FOR i := 1 TO cou DO
     IF (NOT MESSLOST() OR nack[i] = coord) THEN
       Log[ind, nack[i]].vote_request := "Y";
     ELSE
       Mes6(id, nack[i], ctime, "MESSAGE FAILURE IN VOTE_REQUEST");
     END;
   END;
   (* wait for a second time *)
```

```
  Mes2(self.id,  ctime, "is waiting again for vote decisions ");
  time := ctime;
  timeout:= TIMEOUT(Timeout);
  WHILE (Timer < (time + (FLOAT(timeout)/Unit))) DO
       flagv := TRUE;
       FOR i := 1 TO nsubt DO
         IF Log[ind, pick[i]].vote_decision = "U" THEN
           flagv := FALSE;
         END;
       END;
       ctime := Timer;
       IF flagv THEN
         Mes2(self.id,  ctime, "Has received all the votes");
         EXIT;
       END;
       (* CHECK IF SITE OF COORDINATOR HAS FAILED *)
       IF Site[coord].Fail = "Y" THEN
         SITEFAIL(coord, id, ctime);  (* block the coordinator *)
       END;
       Yield();
    END;
  END;

-------------------------------------
      PHASE 2
 Send decision to participants.
-------------------------------------

(* See if all sites have decided yes *)
flagc := TRUE;
FOR i := 1 TO nsubt DO
   IF Log[ind, pick[i]].vote_decision # "Y" THEN
        flagc := FALSE;
    END;
END;

IF NOT flagv THEN
   flagc := FALSE;
END;


(* If all sites have voted yes then send commit decision*)

(* CHECK IF SITE OF COORDINATOR HAS FAILED *)
IF Site[coord].Fail = "Y" THEN
  SITEFAIL(coord, id, ctime);  (* block the coordinator *)
END;

IF flagc THEN
  Mes2(self.id,  ctime, "is sending the commit decision");
  BROADCASTDELAY(coord, nsubt, ctime, pick);

  FOR i := 1 TO nsubt DO
    IF (NOT MESSLOST() OR pick[i] = coord) THEN
```

```
      Log[ind, pick[i]].commit_decision := "Y";
     ELSE
      Mes6(id, pick[i], ctime, "MESSAGE FAILURE IN COMMIT_DECISION");
     END;

   END;
  ELSE
   Mes2(self.id,  ctime, "is sending the abort decision");
   BROADCASTDELAY(coord, nsubt. ctime, pick);

   FOR i := 1 TO nsubt DO
     IF (NOT MESSLOST() OR pick[i] = coord) THEN
       Log[ind, pick[i]].commit_decision := "N";
     ELSE
       Mes6(id, pick[i], ctime, "MESSAGE FAILURE IN COMMIT_DECISION");
     END;
   END;
END;


(* CHECK IF SITE OF COORDINATOR HAS FAILED *)
IF Site[coord].Fail = "Y" THEN
  SITEFAIL(coord, id, ctime);  (* block the coordinator *)
END;

-----------------------------------
 Wait for acknowledgments
-----------------------------------
flage := FALSE;
time := ctime;
timeout := TIMEOUT(Atimeout);
WHILE (NOT flage) DO
    (* If timeout then resend commit or abort decision *)
    IF (Timer > (time + (FLOAT(timeout)/Unit))) THEN
       time := Timer;
       IF flagc THEN
          Mes5(self.id,  time,
              "is sending the commit decision again");
          BROADCASTDELAY(coord, cou, time, nack);
          FOR i := 1 TO cou DO
            Log[ind, nack[i]].commit_decision := "Y";
          END;
        ELSE
          Mes5(self.id,  time,
             "is sending the abort decision again");
          BROADCASTDELAY(coord, cou, time, nack);
          FOR i := 1 TO cou DO
           Log[ind, nack[i]].commit_decision := "N";
          END;
        END;
       Yield();
    END;
    flage := TRUE;  cou :=0;
    FOR i := 1 TO nsubt DO
```

```
            IF Log[ind, pick[i]].finished = "N" THEN
              cou := cou +1;
              nack[cou] := pick[i];
              flage := FALSE;
            END;
          END;
          Yield();
        END;
        ctime := Timer;

        (* dummy := TRUNC(ctime) -(Maxns * Ntime);
        IF (dummy = 0 OR dummy > Maxns + 1) THEN
          dummy := Maxns;
          WriteCard(dummy); WriteTxt(" dummy1"); WriteLn();
        END; *)
        dummy := Second(ctime);
        elapsed := ctime - stime;
        (* DO THE PROCESSING NECESSARY TO THE THROUGHPUT MANAGER *)
        LOCK DMUTEX DO
         (* If the global transaction has commited then increment the
           throughput, else increment the failure *)
         IF Log[ind, coord].commit_decision = "Y" THEN
           Dthru[dummy] := Dthru[dummy] + 1;
           Dresponse[dummy] := Dresponse[dummy]+elapsed;
         ELSE
           CouFail[dummy] := CouFail[dummy] + 1;
         END;
         (* Decrement the number of global transactions *)
         Dtrc := Dtrc - 1;
        END;
        Mes2(self.id, Timer, "***HAS FINISHED***");

RETURN (NIL);
END COORDINATOR;
```

Second

This procedure will return the current rotating second

```
PROCEDURE Second(time :REAL):CARDINAL=
VAR dummy : CARDINAL;
BEGIN
 IF (Maxns*Ntime) > TRUNC (time) THEN
   dummy := 0;
 ELSE
   dummy := TRUNC(time) -(Maxns * Ntime);
 END;
 IF (dummy = 0 OR dummy > Maxns + 1) THEN
   WriteCard(dummy); WriteTxt(" dummy"); WriteLn();
   dummy := Maxns;
 END;
 RETURN dummy;
```

```
END Second;
```

SITEFAILURE

This process will check if a site fails according to a random number.

```
PROCEDURE SITEFAILURE(self : Thread.Closure):REFANY =
VAR ctime : REAL;
  i, nfail, tpause, cmpl, psite, dummy   : CARDINAL;
  flagp : BOOLEAN;
BEGIN

  IF Sitef = 1 THEN
    ctime := Timer;
    WHILE Mpl <= MaxMpl DO
      Pause(Timer, 75000); (* pause for 75 milliseconds *)
      FOR i := 1 TO Nsites DO
        PICKSITE(i, Timer);
      END;
    END;
  ELSE
    nfail := Nsfail;
    tpause := Nthr DIV (Nsfail + 1);
    WriteCard(tpause);  WriteLn();
    cmpl := Mpl;

    WriteCard(Mpl);  WriteLn();
    WHILE Mpl <= MaxMpl DO
      IF cmpl # Mpl THEN
        cmpl := Mpl;
        nfail := Nsfail;
      END;
      IF nfail > 0 THEN
        nfail := nfail - 1;
        LongPause(Timer, tpause);
        flagp := TRUE;
        WHILE flagp DO
          psite := Rnd.Subrange(Rnd.Default, 1, Nsites);
          IF Site[psite].Fail # "Y" THEN
            flagp := FALSE;
          END;
        END;
        Site[psite].Fail := "Y";
        IF Fdebug = 1 THEN
          WriteTxt("Site #"); WriteCard(psite);
          WriteTxt(" Has failed,  Time:");
          WriteReal(Timer); WriteLn();
        END;
        dummy := Second(Timer);
        LOCK MUTEXF DO
          Failure.site[dummy] := Failure.site[dummy] + 1;
        END;
```

```
      Site_recover := Thread.Fork(NEW(Rclosure, Sid := psite));
     END;
     Yield();
    END;
   END;
  RETURN NIL;

END SITEFAILURE;
```

PICKSITE

This procedure will decide if the site will fail randomely.

```
PROCEDURE PICKSITE(sid:CARDINAL; VAR ctime: REAL) =
VAR rndm, srnd, dummy : CARDINAL;
BEGIN

 LOCK Site[sid].Mfail DO
  IF Site[sid].Fail = "N" THEN
     rndm := Rnd.Subrange(Rnd.Default, 1, 100000);
     srnd := TRUNC(Psitefail * 100000.0);
     IF ( rndm <= srnd )THEN
      Site[sid].Fail := "Y";
      IF Fdebug = 1 THEN
       WriteTxt("Site #"); WriteCard(sid); WriteTxt(" Has failed, Time:");
       WriteReal(ctime); WriteLn();
      END;
      (* Increment the site failure *)
      dummy := Second(ctime);
      LOCK MUTEXF DO
       Failure.site[dummy] := Failure.site[dummy] + 1;
      END;
      Site_recover := Thread.Fork(NEW(Rclosure, Sid := sid));
     END;
  END;
 END;

END PICKSITE;
```

SITEFAIL

This procedure is called by the coordinator, it checks if there is a site failure, if there is one it will block the coordinator until the site recovers.

```
PROCEDURE SITEFAIL(sid, id:CARDINAL; VAR ctime: REAL) =
VAR rndm, srnd : CARDINAL;

BEGIN
```

```
 (* Wait until the site recovers *)
 IF Site[sid].Fail = "Y" THEN
  LOCK Moutput DO
   IF Fdebug = 1 THEN
     WriteTxt("Coordinator #"); WriteCard(id);
     WriteTxt(" has stopped, Failure site:"); WriteCard(sid);
     WriteTxt(".Time:"); WriteReal(ctime);
     WriteLn();
   END;
  END;
  WHILE Site[sid].Fail = "Y" DO
    Yield();
  END;
  LOCK Moutput DO
   IF Fdebug = 1 THEN
     WriteTxt("Coordinator #"); WriteCard(id);
     WriteTxt(" has resumed, Recover site:"); WriteCard(sid);
     WriteTxt(".Time:"); WriteReal(Timer);
     WriteLn();
   END;
  END;
 END;
 ctime := Timer;

END SITEFAIL;
```

WAITRECOVER

This procedure will make a transaction wait until its site recovers.

```
PROCEDURE WAITRECOVER(sid:CARDINAL; typep: TEXT; id :CARDINAL;
            VAR Ttime : REAL) =
BEGIN
  IF Fdebug = 1 THEN
    Mes3(sid, id, Timer, typep, "Has stopped, Site failure");
  END;
  WHILE Site[sid].Fail = "Y" DO
    Yield();
  END;
  IF Fdebug = 1 THEN
    Mes3(sid, id, Timer, typep, "Has resumed, Site recovered");
  END;
  Ttime := Timer;
END WAITRECOVER;
```

SITERECOVER

This procedure will make a failed site wait for a random time, before it can recover.

```
PROCEDURE SITERECOVER(self : Rclosure):REFANY =
VAR rndm, srnd : CARDINAL;

BEGIN

 rndm := Rnd.Subrange(Rnd.Default, 8, 10);
 LongPause(Timer, rndm);
 Site[self.Sid].Fail := "N";
 IF Fdebug = 1 THEN
   WriteTxt("Site #"); WriteCard(self.Sid);
   WriteTxt(" Has recovered, Time:"); WriteReal(Timer); WriteLn();
 END;
 RETURN NIL;

END SITERECOVER;
```

MESSLOST

This procedure is called every time a mesage is sent, it will decide if the message will be lost or no according to a random number generator.

```
PROCEDURE MESSLOST():BOOLEAN =
VAR rndm, srnd, dummy : CARDINAL;
BEGIN

 rndm := Rnd.Subrange(Rnd.Default, 1, 100000);
 srnd := TRUNC(Pmesslost * 100000.0);
 IF ( rndm <= srnd )THEN
   (* Increment the message failure *)
   dummy := Second(Timer);
   LOCK MUTEXF DO
    Failure.message[dummy] := Failure.message[dummy]+ 1;
   END;
   RETURN TRUE;
  ELSE
   RETURN FALSE;
 END;

END MESSLOST;
```

MESSAGEDELAY

This procedure will delay the process according to the bandwidth of the system.

```
PROCEDURE MESSAGEDELAY(source, dest : CARDINAL; VAR Ttime : REAL)=
VAR site1, site2, buffer, waitunit, pauseunit, pause, dummy : CARDINAL;
BEGIN
 (* if it is the same site then return without a delay *)
 IF dest = source THEN
```

```
  RETURN;
END;
(* Increment the number of messages *)
dummy := Second(Ttime);
LOCK MUTEXM DO
 Dmessage[dummy] := Dmessage[dummy] + 1;
END;
IF dest < source THEN
  site1 := dest;
  site2 := source;
 ELSE
  site1 := source;
  site2 := dest;
END;
LOCK Link.Mutex[site1, site2] DO
   Link.Sites[site1, site2] := Link.Sites[site1, site2] + Messlength;
   buffer := Link.Sites[site1, site2];
   pause := (buffer * Mdelay) DIV Messlength;
END;

waitunit := Messlength DIV Messunit;
pauseunit := pause DIV waitunit;
WHILE waitunit # 0 DO
    waitunit := waitunit - 1;
    Pause(Ttime, pauseunit);
    Ttime := Ttime + (FLOAT(pauseunit)/Unit);
    LOCK Link.Mutex[site1, site2] DO
       Link.Sites[site1, site2] := Link.Sites[site1, site2] - Messunit;
    END;
END;

END MESSAGEDELAY;
```

BROADCASTDELAY

This procedure will broadcast the message and delay the message according to the highest bandwidth.

```
PROCEDURE BROADCASTDELAY(coord, np : CARDINAL;  VAR Ttime : REAL;
                pick : ARRAY[1..Nsites] OF CARDINAL)=

VAR site1, site2, buffer, waitunit, pauseunit, pause, i, max, dummy : CARDINAL;
BEGIN
 max := 0;
 IF np = 0 THEN
   RETURN
 END;
 FOR i := 1 TO np DO
   IF pick[i] < coord THEN
     site1 := pick[i];
     site2 := coord;
     ELSE
```

```
      site1 := coord;
      site2 := pick[i];
    END;
    IF site1 # site2 THEN
      LOCK Link.Mutex[site1, site2] DO
       Link.Sites[site1, site2] := Link.Sites[site1, site2] + Messlength;
       IF max < Link.Sites[site1, site2] THEN
         max := Link.Sites[site1, site2] ;
       END;
      END;
    END;
  END;

  (* Increment the messages count *)
  dummy := Second(Ttime);
  LOCK MUTEXM DO
   FOR i := 1 TO np DO
    IF pick[i] # coord THEN
      Dmessage[dummy] := Dmessage[dummy] + 1;
    END;
   END;
  END;
  pause := (max * Mdelay) DIV Messunit;
  waitunit := Messlength DIV Messunit;
  pauseunit := pause DIV waitunit;
  WHILE waitunit # 0 DO
      waitunit := waitunit - 1;
      Pause(Ttime, pauseunit);
      Ttime := Ttime + (FLOAT(pauseunit)/Unit);
      FOR i := 1 TO np DO
       IF pick[i] < coord THEN
         site1 := pick[i];
         site2 := coord;
        ELSE
         site1 := coord;
         site2 := pick[i];
        END;
        IF site1 # site2 THEN
          LOCK Link.Mutex[site1, site2] DO
            Link.Sites[site1, site2] := Link.Sites[site1, site2]-Messunit;
          END;
        END;
       END;
   END;

END BROADCASTDELAY;
```

TIMEOUT

This procedure will return the timeout accoring to the current Mpl and the load of the system.

```
PROCEDURE TIMEOUT(maxtimeout: CARDINAL): CARDINAL =
VAR
  basic, remain : CARDINAL;
BEGIN
basic := 1000000;
remain := maxtimeout - basic;
RETURN (basic +(remain * Mpl DIV MaxMpl));
END TIMEOUT;
```

MAIN PROGRAM

```
BEGIN (* Main *)

(* Initialize the global variables *)
FOR ii := 1 TO Nsites DO
   Site[ii].Trc := 0;
   Site[ii].TRC[1] := 0;
   Site[ii].TRC[2] := 0;
   Site[ii].Tid := 0;
   Site[ii].Nudisk := 0;
   Site[ii].Nucpu := 0;
   Site[ii].Ntrans := 0;
   Site[ii].mutex := NEW(MUTEX);
   Site[ii].mutex1 := NEW(MUTEX);
   Site[ii].mutex3 := NEW(MUTEX);
   Site[ii].mutex4 := NEW(MUTEX);
   Site[ii].Mfail  := NEW(MUTEX);

   Site[ii].Fail := "N";
   Scount[ii] := 0;
   FOR jj := 1 TO Maxns +1 DO
    Site[ii].Thruput[jj] := 0;
    Site[ii].Compensate[jj] := 0;
    Site[ii].THRUPUT[1, jj] := 0;
    Site[ii].THRUPUT[2, jj] := 0;
    Site[ii].Response[jj] := 0.0;
    Site[ii].RESPONSE[1, jj] := 0.0;
    Site[ii].RESPONSE[2, jj] := 0.0;
   END;
   FOR jj :=1 TO Nsites DO
    Link.Sites[ii, jj] := 0;
    Link.Mutex[ii, jj] := NEW(MUTEX);

   END;
END;
FOR jj := 1 TO Maxns +1 DO
   Dthru[jj] := 0;
   CouFail[jj] := 0;
   Dmessage[jj] := 0;
   Failure.site[jj]:= 0;
   Failure.deadlock[jj]:= 0;
```

```
    Failure.message[jj] := 0;
    Dmresponse[jj] := 0.0;
END;

Timer := 1.0;  Ntime := 0;  Dpercent := 100 - Lpercent;
Dtrc := 0; Ndtrans := 0; Dtid := 0;
(* Read the data from the input file *)
ReadInput();
Mpl := Smpl;
Maxcpu := Maxdisk DIV 2;
Nsfail := TRUNC(Psitefail * FLOAT(Nthr) * FLOAT(Nsites) * 13.0);
WriteCard(Nsfail); WriteLn();

IF Maxcpu = 0 THEN
   Maxcpu := 1;
END;

(* Initialize the random number *)
Rnd.Start(Rnd.Default, 1);

(* Start the system *)
Clock := Thread.Fork(NEW(Thread.Closure,  apply := CLOCK));
EVAL (Thread.Join(Clock));

END Main.
```