Development and Evaluation of Multidatabase Schedulers

BY

ARUNA ADIL

A Thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

DEVELOPMENT AND EVALUATION OF MULTIDATABASE SCHEDULERS

BY

ARUNA ADIL

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

The autonomy of local database systems in multidatabase environment poses consistency problems in transaction scheduling. Several approaches have been proposed to overcome this problem. In this thesis, we consider the algorithms using two major scheduling approaches. The aggressive approach submits transaction concurrently, but, to ensure the consistency, transactions may have to be aborted and restarted several times. The other approach is serial (or near serial) submission that ensures consistent ordering of transactions. This prevents subsequent aborts.

Two serial schedulers using serial submission approach are developed in this thesis and the results are compared with the best known aggressive "Ticket Method" algorithm. A generic simulator is developed using SMS libraries to implement and evaluate the schedulers. First, the aggressive approach used by Ticket Method is problematic because it does not provide any load control. Second, tuning the Ticket Method is extremely difficult and it does not react well to changes in the load on the local databases. Lastly, the overheads due to rollback and re-executions of this aggressive algorithm makes it less feasible. In this simulation study our serial schedulers perform much better than the Ticket Method in terms of residence time and number of aborts under different levels of load in local databases. This study suggests that it is worthwhile to concentrate on developing schedulers that submit transactions to guarantee that they will not be aborted at the cost of less concurrency.

Acknowledgements

I express my deep sense of appreciation to my thesis supervisor Dr. Ken Barker for his valuable suggestions, constructive criticisms and cooperation during various stages of the research. My thanks are also due to my thesis committee members Dr. David C. Blight and Randal J. Peters for their time, effort and suggestions.

I wish to acknowledge the financial support provided by the Department of Computer Science, University of Manitoba in the form of research assistantship.

I would like to thank Ramon Lawrence for his guidance and assistance in designing and developing a simulation model.

I am thankful to my friends Farook, Allwyn, Karl and others for encouragement and company which has made my stay at Winnipeg pleasant.

Finally, I would like to dedicate this dissertation to my husband Dr. Gajendra Kumar Adil and daughter Deeksha who have been a constant source of support during my studies in M.Sc. program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The need for an application to access multiple heterogeneous databases arises in a wide variety of industries for a number of reasons. Examples include company mergers [HFNL96], the introduction of new technology [HFNL96] or integrating information across several functional units within the organization [BHP92]. This can be achieved in two ways. First, by re-engineering all the systems to a common database model and single access method. This process is expensive and complicated. Second, incorporating multidatabase system (MDBS), gives users a common interface to multiple databases and minimizes the impact on existing database operation. Therefore, multidatabases (MDBs) are important area of research [HFNL96].

A multidatabase system is a facility that allows user to access data located in multiple autonomous and possibly heterogeneous local database systems (LDBS) [BGMS95]. Local transactions (confined to a single database) are submitted directly to the LDBS, while the global transactions (not necessarily confined to a single database) are channeled through the MDBS interface.

Transaction processing in a database management system (DBMS) is accomplished by a transaction manager (TM), scheduler and data manager (DM) as shown in Figure 1.1 [Bar90]. The TM performs two tasks: it interacts with users and coordinates the atomic execution of transactions. The scheduler ensures the correct execution and interleaving of

all the transactions presented to the TM. The DM maintains the database consistency by reflecting the effect of committed transactions while ensuring none of the effects of aborted transactions are made permanent.

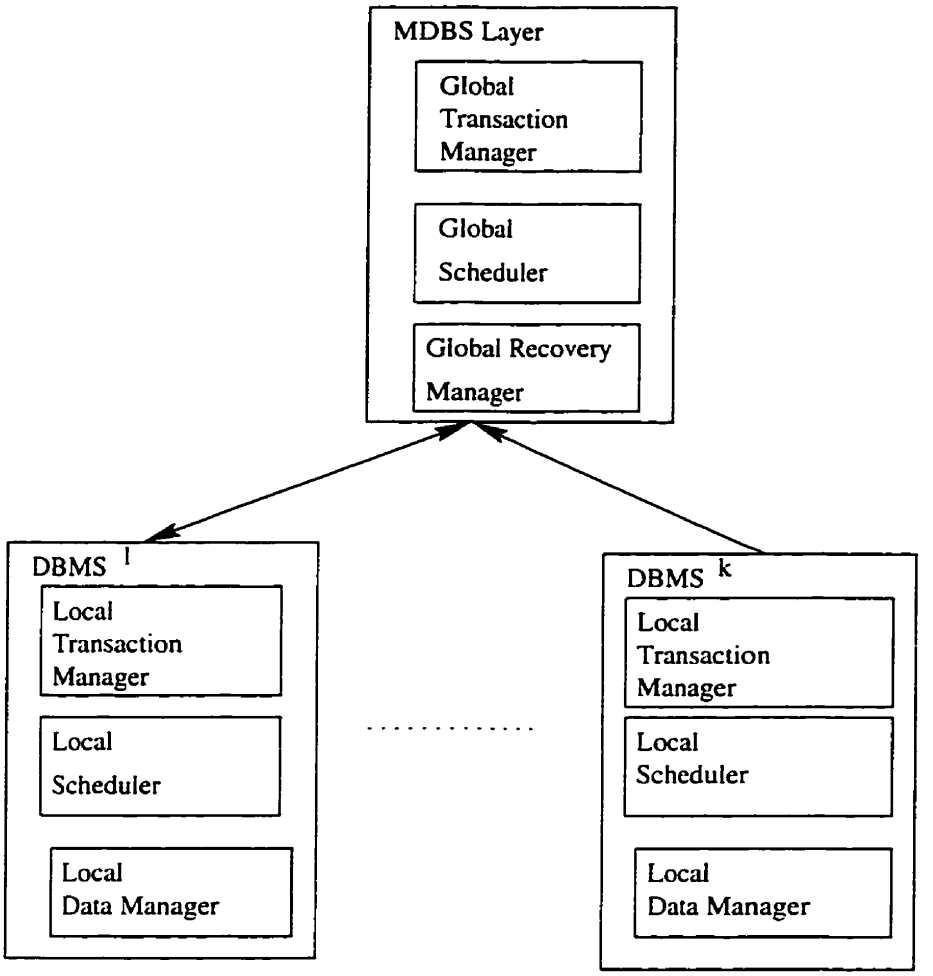


Figure 1.1: MDB Architecture ([Bar90])

The scheduler at each LDBS schedules global sub-transactions submitted by the MDBS and local transactions submitted directly by the user. However, it does not distinguish between local and global sub-transactions.

The MDBS scheduler performs two functions: (i) determines the submission time of global sub-transactions to the LDBSs, and (ii) ensures the correctness of global transactions. LDBS autonomy makes correct execution of global transactions difficult as the MDBS

scheduler is unaware of indirect conflicts caused by local transactions.

The mechanism used to submit global transactions to LDBSs and the correctness criterion employed by the MDBS scheduler affect system performance metrics, such as, transaction residence time and resource utilization. For a given scheduler, the system performance is generally influenced by operating conditions, such as, the nature of transactions, the number of LDBSs, the type of LDBS scheduler and traffic loads. The above interactions are shown in Figure 1.2.

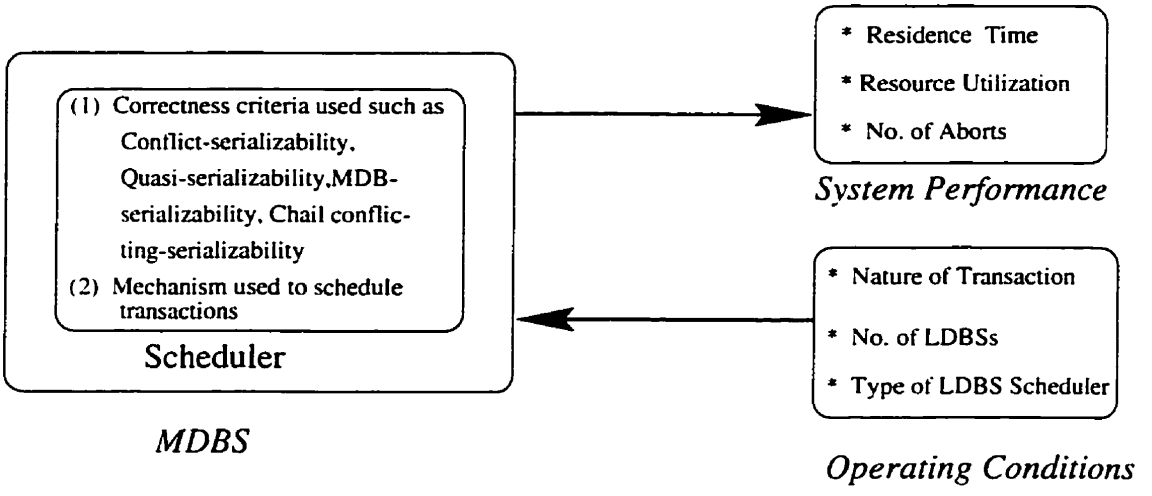


Figure 1.2: Factors that influence MDBS performance

Most of the past research has focused on developing correctness criteria, such as, conflict-serializability (CSR) [GRS94], quasi-serializability (QSR) [DE89], MDB-serializability (MDBSR) [Bar90] and chain-conflicting serializability (CCSR) [ZE93]. These developments form the basis for designing multidatabase schedulers.

Some schedulers can provide a high degree of concurrency but are unsuitable for an MDBS environment because of the autonomy of LDBS. In other words, it cannot guarantee that the concurrent execution will not cause serializability problem. Although it is possible to detect such inconsistency and abort some of the running transactions to rectify the problem, this may lead to unnecessary, expensive re-executions. If transaction are

submitted serially so these aborts are avoided, an increased throughput can be realized.

Conventional databases enforce consistency by strict locking and commit protocols which guarantee that the data is always consistent and trustworthy. Implementation of these protocols in a distributed environment using global locking and two phase commit provide the necessary consistency at a much greater cost than in a centralized system.

1.1 Contribution

To the best of our knowledge, there have been no studies that evaluate the two types of schedulers namely, one that allows concurrent execution but are subject to abort and the other which executes serially but guarantees no rollbacks. In this thesis, we consider the Ticket Method in the first type and develop two serial schedulers (in the second type) based on earlier work reported in [Bar90]. We then develop simulation model to evaluate these two schedulers, and implemented the Ticket Method [GRS94] to determine the real gain achieved in throughput as a result of concurrent processing of transactions. Our findings suggest that an efficient serial submission has good performance characteristics in a MDBS.

1.2 Organization of Thesis

The thesis is organized as follows. Related literature are reviewed in Chapter 2. Two existing schedulers: Ticket Method [GRS94][GRS91] and Global Serial Scheduler (GSS) [Bar90] are also described in Chapter 2. The efficient serial schedulers are developed based on the GSS, are presented in Chapter 3. Chapter 4 presents a simulation model that has been developed for evaluating MDBS schedulers. Chapter 5 gives experimental details and results. Finally, Chapter 6 gives conclusions and recommendation for future work.

Chapter 2

Related Work

The majority of work on MDB is on schema integration [BHP92]. Many projects are still at research level. Some commercial homogeneous MDB systems, "Empress" and "Sybase" are also reported [LZ88].

The transaction management aspect of MDBS is still an open problem. One of the transaction management issues, namely, scheduling of global transactions has been the emphasis in this thesis.

When transactions are executing concurrently they interleave their operations to form an execution schedule or history. The scheduler controls the execution of transactions by restricting the order in which the data manager (DM) executes the Reads, Writes, Commits, and Aborts of different transactions.

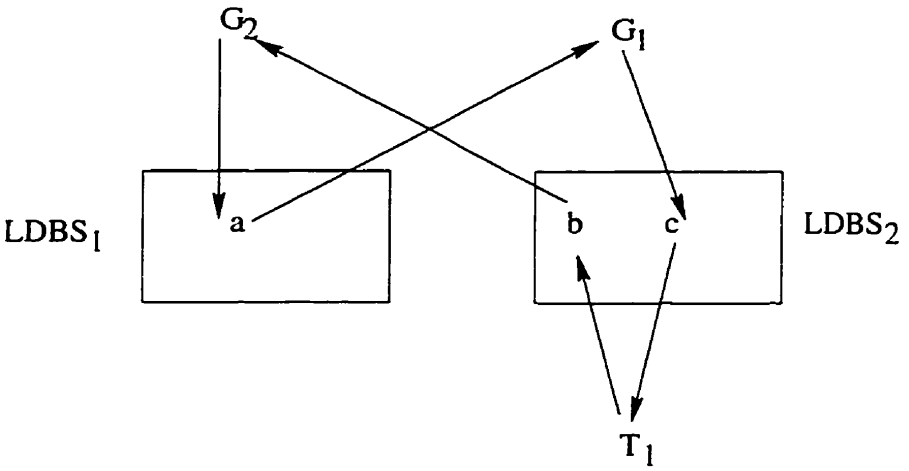
To execute a database operation, a transaction passes that operation to the scheduler. The scheduler then tries to pass it to DM if it can do so without producing a nonserializable execution. If it decides that executing the operation may produce an incorrect result, then it either delays or rejects the operation [BHG87].

Hence, the scheduler has a very important role in concurrency control in databases. In fact, the study of concurrency control techniques is the study of scheduler algorithms that attain serializability [BHG87]. This chapter reviews concurrency control in MDB environment and two existing algorithms to schedule global transactions followed by a

literature survey on simulation studies.

2.1 Concurrency Control in MDB Environment

In MDB environment, autonomous local databases may not communicate any information related to concurrency control to the global transaction manager (GTM) [BHP92] [MRB+92]. Further, the GTM is unaware of indirect conflicts between global transactions at the local DBMSs. This can be explained with the example shown in Figure 2.1 from [GRS94].



$LDBS_1$: $r_{G_1}(a)w_{G_2}(a)$, $G_1 \rightarrow G_2$

$LDBS_2$: $r_{T_1}(c)w_{G_1}(c)r_{G_2}(b)w_{T_1}(b)$, $G_2 \rightarrow T_1 \rightarrow G_1$

Figure 2.1: Indirect Conflict

There are two global transactions G_1 and G_2. Global transactions have sub-transactions in both LDBSs. In $LDBS_1$, G_1 reads a and G_2 later writes it. G_1 and G_2 directly conflict, so the serialization order of the transactions is $G_1 \rightarrow G_2$. In $LDBS_2$, G_1 and G_2 access different data items. G_1 writes c and later G_2 reads b. Hence, there is no direct conflict between G_1 and G_2. However, since local transaction T_1 writes b and reads c, G_1 and G_2 conflict indirectly. This indirect conflict is caused by the presence of local transaction T_1.

The serialization order becomes $G_2 \rightarrow T_1 \rightarrow G_1$. Because of the local autonomy, the MDBS has no information about local transactions. Therefore, it cannot detect indirect conflicts between G_1 and G_2 in $LDBS_2$. Although, both local schedules are serializable, the global schedule is globally non-serializable (with respect to conflict-serializability).

This phenomena is a cause of major difficulties in trying to ensure global serializability in a multidatabase environment [BGMS95].

There are several correctness criteria specified in the MDBS literature that form a basis for scheduling transactions correctly. We review the major ones.

- Conflict-Serializability: A schedule is serializable if and only if it is conflict equivalent to a serial schedule [OV91]. Two schedules S_1 and S_2 are conflict equivalent if for each pair of conflicting operations O_i and O_j such that O_i precedes O_j in S_1, then O_i precedes O_j in S_2 [CP84].

  [GRS94] [GRS91] proposed the optimistic ticket method using conflict-serializability (CSR). This method demonstrates that the serialization order of global sub-transactions in a local site can be determined at the global level without violation of local autonomy. However, in this method, global restarts are possible at each site. Further, acquisition of the ticket may introduce additional conflicts between global transactions that would not have been introduced otherwise. Some improvements to this basic strategy such as, the conservative ticket method [GRS94], the cascade-less ticket method [GRS94], the implicit ticket method [GRS94] and the mixed method [GRS94] are developed to partially overcome these problem.

- Quasi-serializability: [DE89] introduced a less restrictive criterion than conflict-serializability. A global schedule in an heterogeneous distributed database system is quasi-serializable if it is conflict equivalent to a quasi-serial schedule in which all global transactions are submitted sequentially. A global schedule is quasi-serial if all local schedules are

conflict-serializable and there exists a total order of all global transactions such that for every two global transactions G_i and G_j where G_i precedes G_j in the order then, all G_i's operations precede G_j's operations in all local histories in which they both appear. The difference between quasi-serializability and conflict-serializability is that the later treats global and local transactions in the same way while the former treats them differently. More specifically, this theory is primarily based on the behavior of global transactions.

[ZE93] presents three correctness criteria: (a) chain-conflicting serializability, (b) sharing serializability and (c) hybrid ((a) & (b)) serializability. They showed that global serializability can be ensured at the global level by utilizing the intrinsic characteristics of global transactions and controlling their execution. Quasi-serializability is a superset of these three criteria while conflict-serializability (used by optimistic ticket method) is a subset of these three criteria.

- MDB-serializability: [Bar90] proposed MDB-serializability which is an extension of conflict-serializability theory. This generates the same class of scheduler as allowed by quasi-serializability. The global serial scheduler (GSS) is developed based on this criterion.

- Semantic based approach: [SO93] suggested a semantic based approach to allow more concurrency by exploiting the semantics of the transaction. The main idea is to specify acceptable violations of global serializability based on the semantic information of transactions. They developed a semantic-based correctness criterion for MDB transactions. In addition, they developed and implemented LTM based concurrency control algorithms that uses precedence graphs for checking global serializability.

2.2 Scheduling Algorithms

Based on the two correctness criteria, conflict-serializability and MDB-serializability, two existing schedulers Optimistic Ticket Method (OTM) and Global Serial Scheduler (GSS) are considered next. The OTM allows concurrent execution and is based on the conflict-serializability correctness criterion [GRS94]. The GSS allows a global serial execution of transaction and is based on the MDB-serializability correctness criterion [Bar90]. Both schedulers handle the problem of indirect conflicts but do so differently.

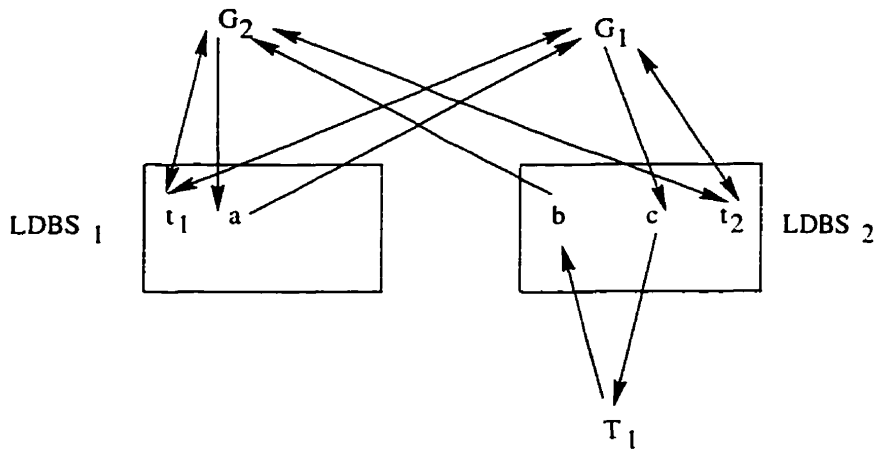
2.2.1 Optimistic Ticket Method (OTM)

To enforce global serializability, the MDB transaction manager must take into account the indirect conflicts between multidatabase transactions caused by local transactions (as explained in Figure 2.1). To overcome these difficulties, [GRS94] proposed to incorporate additional data manipulation operations known as tickets in the sub-transactions of each global transaction. These operations create direct conflicts between sub-transactions at each participating LDBS, and thereby facilitate resolving indirect conflicts even if the multidatabase system is not aware of their existence.

Handling Indirect Conflicts in OTM

The OTM uses *tickets* to determine the relative serialization order of the sub-transactions of global transactions at each LDBS. A ticket is a time stamp whose value is stored as a regular data item in each LDBS. Each sub-transaction of a global transaction is required to issue the *Take-A-Ticket* operation which consist of reading the value of ticket and incrementing it. Figure 2.2 illustrates the effects of the Take-A-Ticket process.

The ticket data items at $LDBS_1$ and $LDBS_2$ are denoted by t_1 and t_2, respectively. In $LDBS_1$, the t_1 values obtained by the sub-transactions of G_1 and G_2 reflect their relative serialization order (it really forces the LDB to order transactions in a certain way because



$LDBS_1$: $r_{G_1}(t_1)w_{G_1}(t_1+1)r_{G_1}(a)r_{g_2}(t_1)w_{G_2}(t_1+1)w_{G_2}(a)$. *i.e.*. $G_1 \rightarrow G_2$

$LDBS_2$: $r_{T_1}(c)r_{G_1}(t_2)w_{G_1}(t_2+1)w_{G_1}(c)r_{G_2}(t_2)w_{G_2}(t_2+1)r_{G_2}(b)w_{T_1}(b)$. i.e..

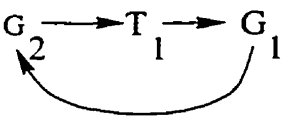


Figure 2.2: The effect of the Take-A-Ticket approach

there is a forced read/write operation). This schedule will be permitted by the local concurrency controller at $LDBS_1$. In $LDBS_2$, the local transaction T_1 causes an indirect conflict such that $G_2 \rightarrow T_1 \rightarrow G_1$. However, by requiring the sub-transactions to take tickets we force an additional conflict $G_1 \rightarrow G_2$. This additional ticket conflict causes the execution at $LDBS_2$ to become locally non-serializable. Therefore, the local schedule:

$$r_{T_1}(c)r_{G_1}\ (t_2)w_{G_1}(t_2+1)w_{G_1}(c)r_{G_2}(t_2)w_{G_2}\ (t_2+1)r_{G_2}(b)w_{T_1}(b)$$

will not be allowed by the local concurrency controller, (eg: the sub-transaction of G_1 or the sub-transaction of G_2 or T_1 will be blocked or aborted.)

Enforcing Global Serializability by OTM

The ticket acquisition order is a valid serialization order forced by the direct conflicts introduced through tickets. The global manager can use this information to maintain

consistency in execution of global transactions. The OTM ensures that the sub-transactions of each global transaction have the same relative serialization order in their corresponding LDBSs. The sub-transactions of each global transactions is allowed to proceed but commits them only if their ticket values have the same relative order in all participating LDBSs.

A global serialization graph (GSG) is constructed in order to record the serialization order of global sub-transaction at each LDBS. Whenever, a global sub-transaction enters its prepared to commit state, it is validated using a GSG.

Transaction processing in the OTM takes place as follows. Initially, it sets a timeout for a global transaction G and submits its sub-transactions to their corresponding LDBSs. All sub-transactions are allowed to interleave under the control of the LDBSs until they enter their prepared-to-commit state. If they all enter their prepared-to-commit states, they wait for the OTM to validate G. The validation is performed using the global serialization graph (GSG) test. The nodes in the GSG correspond to the committed global transactions.

Initially, the GSG contains no cycles. During the validation of a global transaction G, the OTM first creates a node for G in the GSG. Then, it attempts to insert edges between G's node and nodes corresponding to every recently committed multidatabase transaction. Depending on the ticket value obtained by a sub-transaction of G at some LDBS, an edge is added to GSG corresponding to this transaction. If all such edges can be added without creating a cycle in the GSG, G is validated. Otherwise, G does not pass validation, its node together with all incident edges is removed from the graph, and G is restarted. G is also restarted, if at least one LDBS forces a sub-transaction of G to abort for local concurrency control reasons or its timeout expires.

OTM: Algorithm Outline

The execution sequence of a transaction in the OTM can be summarized as follows.

(1) Transaction G arrives.

(2) Set the "timeout" for G and submit all of its sub-transactions ($GSTs$) to related LDBSs.

(3) Before starting any operation on the database, sub-transactions perform Take_A_Ticket operation.

(4) Wait for all $GSTs$ of transaction G to enter the prepare-to-commit state.

(5) Perform a GSG check.

(6) If G passes the GSG check, commit G otherwise abort and restart G after a specified "re-submission time".

(7) G is also aborted if any sub-transactions abort or "timeout" is expired. Restart G after specified "resubmit time".

"timeout" and "resubmit time" must be specified. The "timeout" is defined to resolve global deadlock problem. If the global transaction does not commit by the specified "timeout" period, it will be aborted. The "resubmit time" is the time to submit the transaction again once it is aborted.

2.2.2 Global Serial Scheduler (GSS)

Scheduling of transactions in a multidatabase system is accomplished at both local and global levels. If we assume that each DBMS is capable of generating locally serializable histories, the only requirement of the MDBS is to submit global transactions to each DBMS so that any local ordering can only produce correct schedule. The corresponding schedule is called MDB-serializable schedule [Bar90].

Definition: MDB-Serializability

A schedule is MDB-Serial iff every local schedule is conflict serializable and if an operation of a global transaction precedes an operation of another global transaction in one local

schedule. then all operations of the first global transaction must precede any operation of the second in all local schedules. A schedule is MDB-serializable iff it is equivalent to a MDB-Serial schedule [Bar90].

The GSS schedules transactions so that MDB-serializability is maintained. Hence, the GSS does not have to check for global cycles, as any serialization ordering by the LDBS does not create global inconsistency.

Global Sub-transaction Submission in GSS

The following criterion is used to determine if a transaction is eligible to be submitted for processing.

- The global scheduler determines if there are active global transactions which access more than one of the databases accessed by the transaction G being scheduled. If there is such a set of active global transactions, then G is passivated. If no overlapping global transactions are present, the global sub-transactions of G are submitted.
- A special case occurs when the global transaction G passes the above test because G has only one sub-transaction.

The above test is carried out for new transaction at the time they arrive and for all waiting transactions whenever a running transaction completes because this may make it possible to submit some of the blocked global sub-transactions.

An Example to Illustrate Transaction Submission in GSS

The following example illustrate the transaction submission mechanism of the GSS. Let there be two global transactions G_1 and G_2 running at the databases (see Figure 2.3). Suppose two new transactions G_3 and G_4 arrive as shown in Figure 2.4. G_3 will be submitted because it needs only one database. The active set of databases at this time are DB_1, DB_2 & DB_3. The G_4 requires DB_1 and DB_3 which overlap at more than one of the

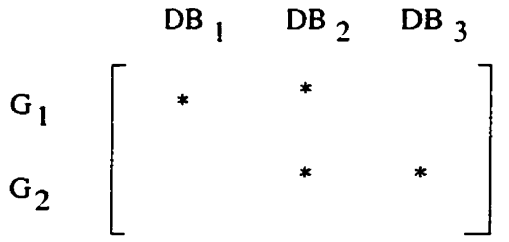


Figure 2.3: Running Transactions

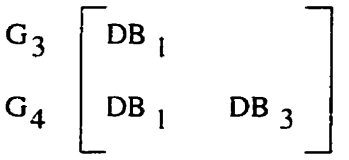


Figure 2.4: New Transactions

active databases. G_4 will not be submitted because the currently running transactions (G_1, G_2, G_3)(Figure 2.5) overlap with those at G_4. In the next chapter, two new schedulers based on the concept of MDB-serializability are described.

2.3 Simulation Studies

The performance of a scheduler cannot be judged on the basis of correctness criterion. It employs operating conditions such as loads (transaction inter-arrival-times, database processing speed) and nature of transactions can greatly influence the performance. Thus, simulation can be a very appropriate tool to evaluate the existing schedulers. Literature on simulation studies in the area of transaction processing on database systems is briefly described next.

[SLSV95] developed a simulation model using the SIM package [ADW92] for centralized database systems to evaluate the performance of random transactions (chopped into pieces) running concurrently. Some research has been reported in simulating distributed databases. For example, [ZYL95] developed a petri-net model to simulate message and transaction queuing in a mobile computing environment.

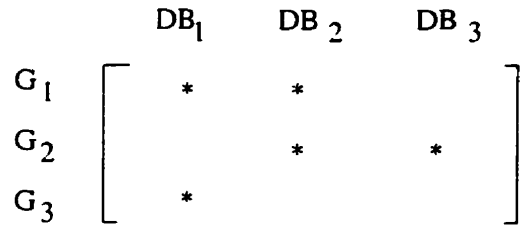


Figure 2.5: Currently Running Transactions

[Tri97] developed simulation model for an MDBS based on a closed queuing model to evaluate the performance of their proposed deadlock detection algorithms. Their performance metric is the ratio of the number of unnecessarily-aborted global transactions over the number of committed global transactions. They conclude that their method of deadlock detection is superior to the one based on time-out in the Ticket Method.

2.4 Motivation and Objectives

It is evident from the literature review that no research has been reported to compare the performance of multidatabase schedulers. Simulation appears to be the most appropriate tool to conduct a detailed study comparing MDB schedulers. The objectives of this thesis can be summarized as:

- Develop a MDBS simulation model which can accommodate different types of GTMs and the schedulers required in this study.
- Suggest possible improvements in existing serial and/or concurrent schedulers. This may lead to development of new schedulers.
- Evaluate the performance of serial and concurrent (Ticket Method) [GRS94] [GRS91] schedulers with metrics such as transaction residence time and resource utilization.

Assumptions

The following situations will be assumed in the simulation model:

- LDBS: The number of LDBS is known. The processing speed of each LDBS is known. Each LDBS follows the Strict Two-Phase Locking rule for concurrency control.
- Transactions: The division of global transactions into sub-transactions is known.

Chapter 3

Development of Schedulers

This chapter develops two serial schedulers (NGSS-1 and NGSS-2). These schedulers use the concept of MDB-serializability proposed by Barker [Bar90]. In NGSS-1, either all the sub-transactions of a transaction are submitted together or none of them will be submitted.

NGSS-2 follows the same idea but submits the transactions more aggressively than NGSS-1. We describe NGSS-1 and NGSS-2 in detail in the following sections.

3.1 New Global Serial Scheduler-1 (NGSS-1)

This section first presents the basics in developing the NGSS-1. Secondly, it provides the implementation details, an example of the transaction submission, and algorithm description.

3.1.1 Preliminaries

The basic idea used in the NGSS-1 is the same as the GSS. However, the GSS is further improved by allowing submission of transactions more aggressively while maintaining MDB-serializability.

To explain the difference between the GSS and the NGSS-1, consider the following example. Let there be two global transactions G_1 and G_2 currently running in their respective databases and a new global transaction G_3 arrives as shown in Figure 3.1. Now, we consider

Currently Active Global Transactions	G_1	DB_1 DB_3
	G_2	DB_2 DB_4
New Global Transaction	G_3	DB_1 DB_2

Action: Do not submit G_3

Figure 3.1: Transaction submission in GSS

the submission criterion employed by the GSS. The GSS checks if there are active global transactions (Active_GTs) which access no more than one database to be accessed by the new transaction. The set of databases accessed by all the Active_GTs considered together in this example are: $\{DB_1, DB_2, DB_3, DB_4\}$. Since this overlaps with the active databases required by new global transaction G_3 is $\{DB_1, DB_2\}$, (i.e. in more than one database), G_3 cannot be submitted.

By using the NGSS-1, G_3 is not blocked as indicated in Figure 3.1. The ordering indicated in Figure 3.2 is possible. G_1, G_2 and G_3 at the local databases cannot create MDB-serializablilty problem because:

(i) G_1 and G_2 do not conflict with each other; and

(ii) G_3 does not conflict with G_1 and G_2 individually in more than one databases.

The NGSS-1 is developed using the concept of grouping to allow the submission of such transactions (G_3).

3.1.2 Implementation Details

To reflect the difference in submission process of the NGSS-1 and the GSS, the test for submission of global transaction in NGSS-1 is modified as follows.

Currently active	G_1	DB_1 DB_3 |
Global transactions	G_2	| DB_2 DB_4
New Global Transaction	G_3	DB_1 | DB_2

Action: Submit Transaction G_3

Figure 3.2: Transaction Submission in NGSS-1

- Group all the global transactions which are currently running so that transactions within a group directly or indirectly conflict with each other. This means that any two transactions which are in two different groups do not conflict with each other.
- If the new global transaction overlaps with active transactions in each group collectively in less than two databases, then it can be submitted.

Next, we describe the mechanism that enforces the submission test of a new global transaction NG (newly arrived or waiting). The test is performed for the new transaction when it arrives or for the waiting transactions at the completion of an active (executing) global transaction.

A newly arrived transaction with one sub-transaction passes the submission test automatically, and hence it is always submitted directly. However, the newly arrived transactions with more than one sub-transaction and transactions in a wait queue need to pass the test to be submitted because they may create inconsistencies. The following procedure is applied.

We consider each database in which the new transaction (NG) requires processing one at a time. For example, if NG requires two databases DB_1 and DB_2 then DB_1 and DB_2 are used as "seeds" to the algorithm. For each seed database (DB_i), the following steps

are performed:

Step 1. Form Group: Identify all the currently active global transactions which need to access database DB_i. These transactions directly conflict with NG. These conflicting transactions form a set called the Conflict_set. The remaining active global transactions which do not directly conflict with NG are put into the Complementary_set. To identify active transactions which may indirectly conflict with NG the following procedure is applied. From the Complementary_set, we identify those transactions that conflict with any of the transactions in the currently identified Conflict_set. Such transactions are removed from Complementary_set and are included in the Conflict_set. This procedure is repeated until no such transaction is found. All the transactions in Conflict_set thus obtained form the "Group" corresponding to the seed DB_i.

Step 2. Determine Overlap: The overlap between the databases required by the new transaction and the databases required by all of transactions in the group considered collectivaly is determined. This information is used to decide if the transaction can be submitted.

After performing Step 1 and 2 for all the seed values, a submission decision is made. If the transaction overlapped with any of the groups in more than one database, it is put on wait queue (Wait_Q), otherwise it is submitted. This procedure is illustrated with an example in the next section.

3.1.3 Illustration of Transaction Submission in NGSS-1

Let there be four global transactions running in five databases DB_1, DB_2, DB_3, DB_4 and DB_5 as shown in Figure 3.3. Thus, currently active global transactions,

$$\text{Active\_GTs} = \{G_1, G_2, G_3, G_4\}.$$

Now, suppose a global transaction (NG) arrives that requires processing in databases DB_1 and DB_2. The grouping is done as follows. First we consider seed DB_1. In DB_1, currently

Active_GTs		
	G_1	DB_1 DB_3
	G_2	DB_3
	G_3	DB_2 DB_4
	G_4	DB_5
New Transaction	NG	$\uparrow$ DB_1 DB_2

$\uparrow$ determine transactions conflicting with NG at DB_i

Figure 3.3: Grouping of Transactions for seed as DB_1

only G_1 is active. G_1 directly conflicts with NG. Hence:

$$\text{Conflict\_set} = \{ G_1 \} \text{ and}$$

$$\text{Complementry\_set (the remaining active GTs)} = \{ G_2, G_3, G_4 \}.$$

Now, we check for any indirect conflict between NG and transaction in Complementry_set. This requires checking for direct conflict between G_1 and $\{G_2, G_3, G_4\}$. G_1 conflicts with G_2 only. Thus:

$$\text{Conflict\_set} = \{G_1, G_2\}. \text{ Complementry\_set} = \{G_3, G_4\}.$$

G_1 does not conflict with G_3 and G_4 nor does G_2 conflict with G_3 and G_4. So, the final conflict set is:

$$\text{Conflict\_set} = \{ G_1, G_2 \}$$

using DB_1 as the seed. The grouped data is presented in Figure 3.4

Next, we determine overlap of transactions in group 1 with NG (Step 2);

$$\{DB_1, DB_3\} \cap \{DB_1, DB_2\} = \{ DB_1 \}$$

Thus, the overlap is one database.

By applying the same procedure with seed as DB_2, we get the following.

$$\text{Final Conflict\_set} = \{G_3\} \text{ (see Figure 3.5) and the}$$

$$\text{Overlap} = \{DB_2\} \cap \{DB_1, DB_2\} = \{DB_2\}$$

Since there is no more seed database remaining, the procedure of grouping stops. As we have found in Step 2, NG does not overlap in more than one database with the two groups: $\{G_1, G_2\}$ and $\{G_3\}$ formed so it can be submitted.

Conflicting Transactions	G_1	DB_1 DB_3	
	G_2	DB_3	
Non-Conflicting Transactions	G_3		DB_2 DB_4
	G_4		DB_5
Overlap with Conflicting Transaction	NG	$\uparrow$ DB_1	DB_2

Figure 3.4: Grouped transactions for seed as DB_1.

Conflicting Transactions	G_3	DB_2 DB_4	
Non-Conflicting Transactions	G_1		DB_1 DB_3
	G_2		DB_3
	G_4		DB_5
Overlap with Conflicting Transaction	NG	$\uparrow$ DB_2	DB_1

Figure 3.5: Grouping of Transactions for seed as DB_2.

3.1.4 Algorithm Description

The following are the list of data structures and functions necessary to present the algorithm.

- **DBMS_set(G_i)**: The set of DBMSs where the sub-transactions of G_i are executed.
- **Active_set($DBMS^k$)**: The set of global transactions which have an active sub-transaction executing in $DBMS^k$.
- **Wait_Q**: Global transactions which cannot be submitted immediately are put in this queue.
- **Conflict_set**: Set of DBMSs where a new transaction may directly or indirectly conflict.
- **Complementary_set**: Set of DBMSs where a new transaction may not directly or indirectly conflict.
- **Card(s)**: The cardinality function returns the number of elements in the set s.

The algorithm is described with four segments:

SEG 1: Procedure "Process newly arrived transactions (Figure 3.6)".

SEG 2: Function "Form group (Figure 3.7)".

SEG 3: Function "Determine overlap (Figure 3.8)".

SEG 4: Procedure "Process Wait_Q (Figure 3.9)".

As the transaction arrives **SEG 1** is activated. This checks if the transaction requires only one database or more. It directly submits if this is the case or it calls the **SEG 2** Function Group() to group the currently running transactions. Once the group is formed.

it calls **SEG 3** Function Test() to check if this transaction can be submitted. Based on the returned value of function Test(), it submits the transaction or places it in the Wait_Q.

As the currently running transaction completes its execution, the sub-transaction Termination procedure (**SEG 4**) is called which is responsible for re-testing transactions in the Wait_Q in first come first served order.

3.2 New Global Serial Scheduler-2 (NGSS-2)

We introduce NGSS-2 using an example shown in Figure 3.10. Assume there is a global transaction G_1 running on databases DB_2 and DB_3. A new global transaction G_2 arrives which requires databases DB_1, DB_2 and DB_3. If we apply NGSS-1, G_2 will not be submitted

Seed	Overlap
DB_1	no overlap
DB_2	$\{DB_2, DB_3\}$
DB_3	$\{DB_2, DB_3\}$

because of the overlap in two databases DB_2 and DB_3 with seeds DB_2 and DB_3. NGSS-1 follows an all or nothing principle. However, the overlap with seed DB_1 is null. This suggests that a sub-transaction of G_2 requiring database DB_1 could be permitted to execute while sub-transactions requiring databases DB_2 and DB_3 are blocked. We develop NGSS-2 to allow such submission. Thus, the sub-transactions (seeds), which passes submission test can be potentially submitted by NGSS-2. This submission does not cause any serializability problem with those that are fully submitted (i.e., have all the sub-transactions running). However, this may cause problems, for transactions that are not fully submitted.

An example will be illustrative. Let there be a global transaction G_1 which is active in databases DB_3 and DB_4. A new transaction G_2 (newly arrived or waiting transaction) requires DB_1, DB_3 and DB_4. Sub-transaction requiring DB_1 will be submitted because there is no overlap on this database (seed). Sub-transactions requiring databases DB_3 and DB_4 will not be submitted and will be put on Wait_Q. Now, suppose another new

transaction G_3 which requires DB_1, DB_2 and DB_3 arrives. Since there is no overlap in DB_2, sub-transaction requiring DB_2 will be submitted. Other sub-transactions will enter the Wait_Q. Now, Transaction G_1 completes. The Wait_Q for each database is processed. For the Wait_Q of database DB_3, transaction G_2 and G_3 will not be submitted ever as shown in Figure 3.11. Since G_2 and G_3 are conflicting transactions the test for submission of both transactions will fail. To avoid this situation an extra check on the group of transactions is performed as follows. In NGSS-1, to create the Conflict_set we check the overlap with only the seed database being checked. In NGSS-2, once this check is done we make additional tests as follows. We block submission of a sub-transaction (requiring databse DB_i) of transaction NG if any partially submitted transaction is waiting whether or not they access database DB_i.

The NGSS-2's partial submission policy makes it more aggressive than NGSS-1 but the submission test in NGSS-2 is more rigorous than in NGSS-1. This tradeoff makes it difficult to predict which of the two is more aggressive. Simulation studies discussed in Chapter 4 are required to answer this question.

3.2.1 Algorithm Description

The list of data structures necessary to describe the algorithm are the same as in NGSS-1 except for the Wait_Q. In NGSS-1, the entire transaction is placed in Wait_Q if it cannot be processed immediately. In NGSS-2, part of a transaction can be submitted. Thus the sub-transactions wait in the Wait_Q. An additional structure "GSTs_complete" is required in NGSS-2.

- **Wait_Q($DBMS$)**: Global sub-transactions which cannot be submitted immediately are placed on a queue waiting to access the DBMS. There is one Wait_Q for each DBMS.

- **GSTs_complete(G_i):** Set of DBMSs which have completed the global sub-transactions submitted by G_i.

The algorithm is explained in segments described below.

SEG 1: Procedure "Process new transaction (Figure 3.13)".

SEG 2: Function "Group (Figure 3.12)".

SEG 3: Function "Determine overlap (same as in NGSS-1) (Figure 3.8)".

SEG 4: Procedure "Process wait queue (Figure 3.14)".

As the new transaction arrives. **SEG 1** is activated. For each database (required by transaction) it calls **SEG 2** to form groups of active transactions. Once a group is formed. this segment (**SEG 2**) is called again to make groups of partially active transactions. After both groupings are done. **SEG 3** function is called to determine the overlap. Based on the overlaps. **SEG 1** decides whether to submit a transaction or put it in Wait_Q.

When an active transaction completes its execution. the transactions waiting in the Wait_Q of each database is processed in the first come first served order. **SEG 4** process the waiting transactions by calling **SEG 2** and **SEG 3**.

```
NGSS-1: Submission of A New Transaction ( at its arrival)
  input  NG: new global transaction to be submitted;
  var    Active_GTs: set of active transactions;
         DBMS_set(NG) : set of DBMSs accessed;
  output Submit NG: submit transaction;
         Wait_Q: Put NG in Wait_Q;
    begin
     if card(DBMS_set(NG) == 1) then
        begin
            Active_set(DBMS^k)← Active_set(DBMS^k)∪ NG;
            submit NG to DBMS^k
        end
     else begin
        Active_GTs ← ⋃_k Active_set(DBMS^k);
        for each DBMS^k ∈ DBMS_set(NG) do
            begin
                call Function group();
            end
        if test() == pass then
            begin
                for each DBMS^k ∈ DBMS_set(NG) do
                  begin
                    Active_set(DBMS^k) ← Active_set(DBMS^k) ∪NG
                    submit NG
                  end
                end
        else begin
            put Transaction NG in Wait_Q
        end
    end
```

Figure 3.6: Initial Submission Algorithm

```
NGSS-1: Function Group (Grouping of Conflicting Transactions)
    input G_i: global transaction to be submitted:
          seed = DBMS^k, Active_GTs:
          Conflict_set: set of conflicting transactions:
          Complementary_set: set of non-conflicting transactions:
        begin
          for each G_j ∈ Active_GTs do
              if card((DBMS_set(G_j) ∩ DBMS^k) != 0)
                  then begin
                      Conflict_set ← Conflict_set ∪ G_j;
                  end
              else begin
                      Complementry_set ← Complementry_set ∪ G_j:
              end
          end for
          for each G_j ∈ Complementry_set do
              for each G_k ∈ Conflict_set do
                  if card((DBMS_set(G_j) ∩ DBMS_set(G_k)) != 0)
                      then begin
                          Conflict_set ← Conflict_set ∪ G_j:
                          Complementry_set ← Conflict_set - G_j:
                      end
              end for
          end for
        end
```

Figure 3.7: Algorithm to Form Group

```
NGSS-1: Function Test (Test for Transaction Submission)
 input G_i: Global Transaction to be submitted;
    begin
        if card (Conflict_set ∩ DBMS_set(G_i)<2)
           then begin
                return pass
           end
        else begin
                return fail
        end
    end
```

Figure 3.8: Algorithm to Determine Overlap

```
NGSS-1: Resubmission of Waiting Transactions
 input: G_i: Completed Global Transaction:
 var: Active_GTs:
 output: Submit: Submit the transaction:
       Wait_Q: Put transaction in Wait_Q;
 begin
   for eachDBMS^k ∈DBMS_set(G_i) do
       begin
          Active_set(DBMS^k) ← Active_set(DBMS^k) - G_i;
          DBMS_set(G_i) ← DBMS_set(G_i) - DBMS^k;
       end
   if DBMS_set(G_i) == φ then
       begin
          for each G_n in  Wait_Q do
              begin
                 Active_GTs ← ⋃_k Active_set(DBMS^k);
                 for each DBMS^k ∈ DBMS_set(G_n) do
                    begin
                       call Function group():
                    end
                 if test() == pass then
                    begin
                       for each DBMS^k ∈ DBMS_set(GT_n) do
                           begin
                              Active_set(DBMS^k) ← Active_set(DBMS^k) ∪G_n:
                              Wait_Q ← Wait_Q - G_n;
                              submit G_n :
                           end
                    end
                 else begin
                    put G_n in Wait_Q:
                 end
              end for
       end if
 end
```

Figure 3.9: Algorithm to Process Wait_Q

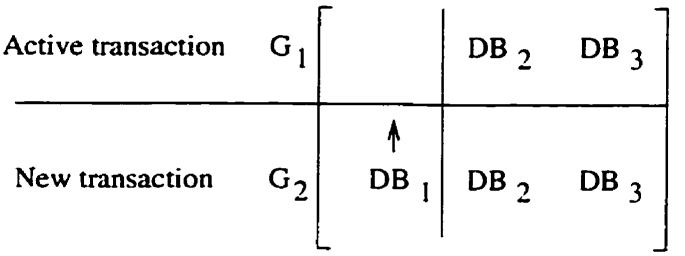


Figure 3.10: Submission Process in NGSS-2

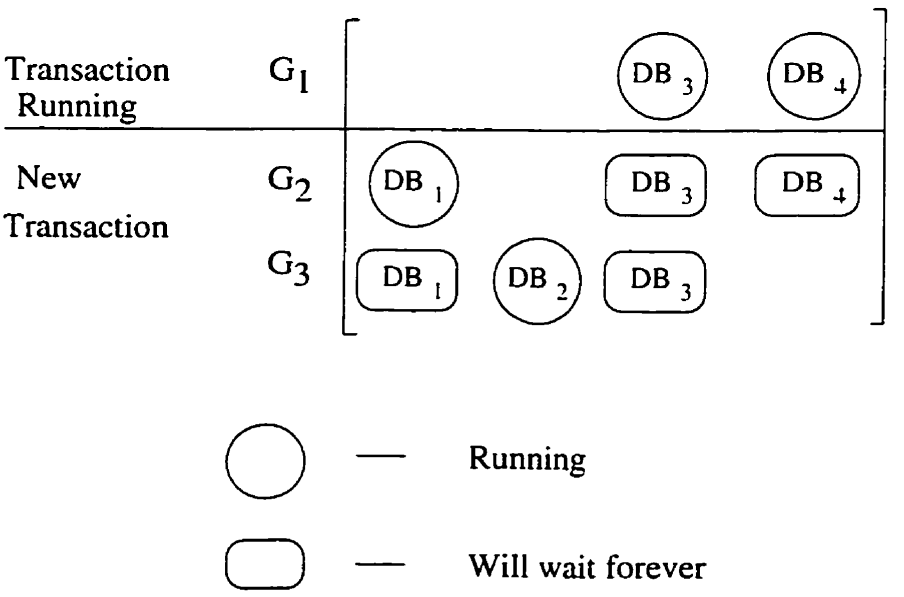


Figure 3.11: Deadlock Condition

```
NGSS-2: Function Group (Grouping of conflicting Transactions)
  input G_i: Global Transaction to be submitted:
        seed = DBMS^k, Active_GTs, n:
        Conflict_set: set of conflicting transactions:
        Complementary_set: set of non-conflicting transactions:
      begin
       if n = 1 then
          temp = DBMS^k;
       else if  n = 2 then
          temp = DBMS_set(G_i);
       for each G_j ∈ Active_GTs do
          begin
             if card((DBMS_set(G_j)∩ temp) != 0)
                then begin
                     Conflict_set ← Conflict_set ∪ G_j:
                end
             else begin
                Complementry_set ← Complementry_set ∪ G_j:
             end
          end for
       for each G_j ∈ Complementry_set do
          for each G_k ∈Conflict_set do
             if card((DBMS_set(G_j) ∩ DBMS_set(G_k)) != 0)
                then begin
                   Conflict_set ← Conflict_set ∪ G_j:
                   Complementry_set ← Conflict_set - G_j:
                end
          end for
       end for
      end
```

Figure 3.12: Function for Grouping

```
NGSS-2: Scheduling A Newly Arrived Transaction (NG)
  input NG: new global transaction;
  var   Active_GTs: set of active transactions;
        DBMS_set(NG) : set of DBMSs accessed;
    begin
     if card(DBMS_set(NG) = 1)then
        begin
            Active_set(DBMS^k)← Active_set(DBMS^k)∪ NG;
            submit NG to DBMS^k;
        end
     else begin
        Active_GTs ← ⋃_k Active_set(DBMS^k);
     for each DBMS^k ∈ DBMS_set(NG) do
        begin
            call Function group(n=1);
           if (test()==1) then
               begin
                  Active_GTs ← ⋃_k Active_set(DBMS^k) ∩ ⋃_k Wait_Q(DBMS^k);
                   call Function group(n = 2);
                  if (test()== pass) then
                     begin
                         Active_set(DBMS^k) ← Active_set(DBMS^k) ∪NG;
                         submit NG in DBMS^k :
                     end if
               end if
           if (test(for n=1)!=1 || test(for n=2)!=1) then
               begin
                  put Transactin in Wait_Q of DBMS^k;
               end
        end for
     end if
    end
```

Figure 3.13: Process New Transaction

```
NGSS-2: Scheduling A Waiting Transaction
  input G_i: global transaction completed in a database:
  var   Active_GTs: set of active transactions;
        GSTs_complete: sub-transactions of a global transaction completed:
        Active_set(DBMS^k): active transactions in each DBMS^k:
        DBMS_set(G_i): set of DBMSs accessed by G_i;
    begin
     GSTs_complete ← GSTs_complete ∪ G_i;
     Active_set ← active_set - G_i;
     if (card(GSTs_complete ∩ DBMS_set(G_i))) then
        begin
          for each DBMS^k in Wait_Q do
             begin
                for each G_n ∈ Wait_Q(DBMS^k) do
                   begin
                      call Function group(n=1):
                     if test()== pass then
                        begin
                           Active_GTs ← ⋃_k Active_set(DBMS^k) ∩ ⋃_k Wait_Q(DBMS^k):
                           call Function group(n = 2):
                           if test() == pass then
                              begin
                                 Active_set(DBMS^k) ← Active_set(DBMS^k) ∪G_n:
                                 Wait_Q(DBMS^k) ← Wait_Q(DBMS^k) - G_n:
                                 submit G_n :
                              end if
                        end if
                   if (test(for n=1)!= 1 || test(for n=2)) then
                      begin
                        put G_n in Wait_Q of DBMS^k:
                      end
                   end for
             end for
        end if
    end
```

Figure 3.14: Process Wait_Q

Chapter 4

Simulation Model

Simulation is used to study the dynamics of a system without building the actual system. Furthermore, the simulation approach gives more flexibility and models system dynamics not easily achieved with analytical models. A wide variety of simulation software tools are available to facilitate the analysis and development of simulation models. Simulation models can be built using one of the two basic approaches [SSM96]. First, using a general purpose simulation languages (such as, SIMULA, SIMSCRIPT II.5) to write the required functions. This approach provides great flexibility in modeling a system's characteristics. However, it requires substantial expertise in simulation programming. The second approach uses high level simulation and prototype tools, which provides an easier means to model the system. However, these offer less flexibility in modeling specific details. To develop a generic simulator for multidatabase the first approach is adapted here. A discrete event simulation model is developed that uses a simulation library augmented by user developed routines to capture the functionality of the multidatabase system. In this chapter, the system is described for transaction processing in the multidatabase environment and then the development of the simulation model is presented.

4.1 Description of the System: Transaction Processing in Multidatabase Environment

A multidatabase is a collection of one or more autonomous databases participating in a global federation for the exchange of data. Basic scheme of transaction processing in multidatabase (MDB) environment is shown in Figure 4.1.

The MDBS is composed of two layers: (i) the multidatabase (MDB) layer and (ii) the local database (LDB) layer. Global transactions (GTs) (requiring processing on more than one database) are submitted to appropriate LDBs through the MDB while local transactions are submitted directly to the LDBs. Next, we describe the transaction manager and database objects at each layer.

Global Transaction Manager: The MDB layer's GTM divides GTs into sub-transactions (GSTs) and handles their execution. GTMs differ in the type of transaction scheduling and monitoring mechanism used (eg. Ticket Method) for enforcing the correct execution of global transactions. Global transactions begin executing by submitting some/all of its sub-transactions to global transaction server (GTS) depending on the global transaction manager (GTM). The GTS is the interface between the MDBS and the LDBS. The GTS is responsible for inferring the global view provided by the LDBS and initializing structures to maintain a communication channel between the two layers. It is also responsible for creating a new local transaction from a global sub-transaction and returning the result after its execution.

Global Database Objects (GDOs): GDOs are the objects in the global view. All objects in the global view are stored in some LDBS. A GDO has a name unique across the entire MDBS, which may be a local object promoted to the global view. That is, the set of GDOs provided by the LDBS for the global view may only be a subset of the objects in the local database. However, it is entirely possible that there exists global objects in the

LDBS that are not accessible by local transactions [1] .

There are several concurrency control strategies used at the local level including locking or time stamp ordering. This gives rise to many local transaction managers.

Local Transaction Manager (LTM): The LTM is responsible for all aspects of transaction management including concurrency control, reliability, deadlock resolution, etc. The simulation model currently implements strategies that is known to produce only correct execution sequences.

Local Database Objects: Objects can be anything we chose to represent in the database. They can be actual data objects or control objects used by the system. It can be stored in many forms, i.e., as relational database, as object-oriented database etc. These are the resources used to process local transactions and global sub-transactions.

4.1.1 Requirements of Simulation Model

Based on our discussion in the previous section, simulation of transaction processing in MDB environment requires:

- Modeling database functionality and resources at both local and global levels (model static objects).
- Modeling dynamic activities, i.e., transaction arrival and processing at local and global levels.

Our simulation models for static and dynamic objects are described next.

4.2 Simulating Static/Passive Objects

Transaction managers and database objects at local and global databases, and the GTS are modeled as static objects. Behavior of these objects does not change during the simulation run.

[1] This permits modeling strategies that use a partitioning strategy for local and global data

4.2.1 Local Database

First, we describe our simulation of local database elements: local database objects and the local transaction manager itself.

Local Database Objects: There are a variety of database models for representing data. A general database simulator must not only be able to handle the more common relational and object-oriented models, but must also capture legacy models such as the hierarchical model. It is infeasible to implement each model in isolation, so the challenge in modeling database objects is to define a generic database object usable by many different database and transaction models.

A generic database object is defined independent of any database model. This independence is achieved by realizing that fundamental to all models and information representation are two basic components: objects and object references.

An object is a generic container for storing information whose use is dependent on the information stored. Objects in isolation are useless because there is no way to determine their relationship to other objects in the environment. The meaning of this relationship may vary with the type of object or how the object is used, but allowing a generic relationship between objects captures any type of relationship. The meaning of the relationship can be defined externally to the object, but the object knows the existence of the relationship. A special type of relationship is also defined for object-oriented models. The subtype or subobject relationship allows a hierarchy to be imposed on the objects instead of the generic relationship which implies equality in the object reference.

The generic database object, **db_object**, has a name unique across the local and the multidatabase domains. Instead of storing actual data, an integer value stores the object size. A db_object gathers statistics on object usage and partially mediates object access. Current statistics gathered on db_objects include:

- lock time - The fraction of the time that the object is locked.
- queue size - The number of transactions waiting for the objects.
- number of uses - The number of times that the object is used.

Since locking is common in many transaction management protocols, a db_object implements locking as an interface. A call to lock a db_object by the transaction manager may result in the object blocking the transaction, by putting it on a FCFS wait queue for the object, if access cannot be immediately granted. The system allows an unlimited number of read locks as long as no writer is waiting. Once a transaction process is put on a queue it stops execution. Control is passed back to the transaction manager, which exits and waits for the next simulation event from the simulation controller. When a transaction unlocks an object, the db_object class checks the queue for waiting transactions and will restart any transactions in the simulation that have been waiting for the object. After a write completes all reads in the queue are processed before another write is started.

Defining database objects does not fully define a database. Each database model will have a different way of combining these database objects to form a workable database. A simulated database is constructed by the virtual **base_db class** and the classes derived from it. The base_db class defines virtual functions for loading and saving a database configuration file and for defining database performance. Database performance is a simple integer value representing the number of bytes it can process per second. There are no restrictions on parallelism and dividing this resource. For example, many read transactions can be executing simultaneously and each will receive the maximum processing speed.

The virtual base_db class serves as the basic definition for the different database models. A relational database model called the **rel_db class** is defined. The rel_db class has methods for adding/removing relations from the database, locking/unlocking relations, and loading/saving database configurations. The database consists of an integer storing

the number of relations in the system and a B+Tree storing the unique object names as keys and pointers to db_objects as data. Database configurations are simulation parameters that are simply retrieved from text files.

Local Transaction Manager: The transaction manager at the local level is simply referred to as TM. A lot of database functionality normally provided by the TM has been divided among the other units to simplify the implementation of the transaction manager. For example, the database objects help maintain locks and the transactions themselves help in their execution. Nevertheless, there remains several requirements of a TM that must be defined and are specific to each individual TM. They include the handling of transaction initialization, commit, abort, and the execution of operations. The only TM implemented is relational strict two-phase locking. No deadlocks occur in this implementation as resource ordering is used. Since the system will never deadlock, transactions will never be aborted.

Transaction initialization in the strict-2PL locking implementation assigns a unique transaction id and records the time of the transaction's arrival. Transaction commit releases all locks and records the transaction residence time in the system. Transaction abort is not required so it is not implemented.

Execution of transaction operations varies depending on how the transaction manager allocates resources. The strict-2PL locking implementation uses locks to mediate object access. If a transaction can acquire the necessary locks, it is allowed to execute the operation. Executing an operation involves waiting for a given time depending on the operation and then being restarted by the simulation system after this wait is completed. The wait time of the transaction process equals the time it takes to perform the operation. If a transaction fails to acquire a lock, it is placed in a queue for the object (in db_object) and the transaction manager returns control to the simulation system to pick a new transaction to execute. Time for lock acquire/release is assumed to be zero. Lock activities are

assumed to take zero time because the time to access the lock in real system is negligible as compared to the time to read/write the relation. A zero lock time does not affect the validity of comparing GTMs as all GTMs must use the identical databases with the same relative performance.

The time to execute an operation is linear in object size. The database has an associated speed in bytes/second. and the time to execute the operation is the size of the operand divided by the database speed.

The TM gathers statistics on the number of transactions (committed and aborted) and throughput in both transactions/sec. and bytes/sec. An important statistic is average transaction execution time which is a good comparison between transaction managers. Additional results and metrics are described later.

4.2.2 Multidatabase

The multidatabase simulator is general enough to allow multiple MDBS configurations and different global transaction managers (GTMs). The MDBS simulator is divided into a set of classes which provide the required functionality of a MDBS. Testing different MDBS transaction managers only requires redefining the class associated with transaction manager specific functions. This allows for greater code reuse and consistency across simulations of different transaction managers.

Global Database Objects: Two entities control access to global objects. The MDBS class is responsible for the entire definition of the MDBS. It has functions for adding and removing a database (and the global objects they contain). adding global objects to the global view, and maintaining the list of global objects and local databases. References to the local databases are stored in a B+Tree using the unique DB name (the key) and a pointer to the global transaction server (GTS) managing the local database (the data). Pointers to all global objects in the global view are stored in the B+Tree.

Global Transaction Server (GTS): The GTS is responsible for loading the global view provided by the LDBS and initializing structures to maintain a communication channel between the two entities. It creates a new local transaction from a global sub-transaction and returns the result after its execution. A GTS maintains a list of global objects which the local database provides to the global view. It also maintains a list of local transactions submitted from global transactions currently executing on the database. It is assumed that the time to communicate between the global level (MDBS) and the LDBSs through the GTS is zero.

Global Transaction Manager (GTM): The global transaction manager insures that the submission of global transactions and their sub-transactions are executed serializabily. A GTM has virtual functions for initializing, running, and committing transactions. It also maintains statistics on the number of committed and aborted transactions and the residence time of global transactions in the system.

Three transaction management algorithms are implemented: the Ticket Method GTM, NGSS-1 and NGSS-2 GTM. Each is described below.

Ticket GTM Implementation: The Ticket Method GTM implementation adds a ticket object to each LDBS participating in the MDBS. A ticket consists of a 1-tuple relation of size 10 bytes. The ticket for each LDBS is also added to the global view.

When initializing transactions, the global transaction is registered (assigned a unique id) and its initialization time is recorded. Since the Ticket Method imposes no restrictions on the order of sub-transaction submission, all sub-transactions of the GT are submitted immediately. The GT is then passivated while waiting for results from sub-transaction completion. However, in addition to the normal operations of the sub-transaction, the Ticket Method adds an operation to increment the ticket counter.

When sub-transactions complete, the GTS reactivates the GT. If there are still outstanding GSTs, the GT is passivated again. Otherwise, it attempts to commit.

In the commit phase, the GTM determines if there were any global conflicts using the global serializability graph (GSG) test. If there were conflicts, all GSTs are aborted. Otherwise, a "signal" is sent to all GSTs telling them to commit. This is not two-phase commit. The simulation assumes that once the signal is sent, all LTs successfully commit.

Implementation of NGSS-1 and NGSS-2: When a global transaction GT_i is initiated, it is determined if the transaction should be scheduled. Note that a GT_i with only one sub-transaction can always be submitted. In NGSS-1, if the DBMS_set of GT_i does not conflict with the Conflict_set of all active GTs (i.e. they have no more than one database in common) the GT_i is submitted. If the test fails, GT_i goes into a wait queue. In NGSS-2, the same check is performed for each sub-transaction individually and sub-transactions are submitted individually.

A GT reaches the commit phase after completion of all its sub-transactions. DBMS_set and Active_set are updated at the completion of GT. Subsequently the wait queue is checked, and all the eligible transactions are scheduled.

4.3 Simulating Dynamic Objects

Object behavior changes over time. Section 4.3.1, describes the basic functions provided by the standard C++ library used in this research. In Section 4.3.2, transaction generation and processing is described.

4.3.1 Basic Framework (C++ Simulation Library)

The simulation framework is provided by the simulation modeling support (SMS) library designed at Vrije Universiteit [BE95] using C++. The SMS library uses Discrete Event Simulation and supports both an event and process-oriented approach in developing simulations. Essentially modeled components causes events that change the system's state. Events exist autonomously and are discrete so "nothing" occurs between two events.

The SMS library provides several classes including:

- *Session*: The Session class derives application from SMS library.
- *Simulation*: This class schedules the events. It employs a calendar of events and repeatedly extracts the events, that should execute.
- *Event and Entity*: These classes are used to model dynamic objects.
- *Generator*: It permits a variety of random number streams and probability distributions for generating random numbers.
- *Resource*: A Resource represents a passive objects to be used by events.
- *Queue*: It maintain queued events, for example waiting for a resource to become free.
- *Histogram*: This class gathers statistics and print the results of simulation.
- *Additional classes* (to support graphical display, animation etc.)

A simulation program, employing the SMS library, derives an application from the *Session* class and overwrites its *main* function. The *session::main* function then creates a simulation object which includes the required resources and queues (static objects) and histogram and analysis objects for gathering and analyzing results. The dynamic objects are derived from the classes *event* and *entity*, and are given functionality by overriding the *application operator* of these classes. Before running the simulation, initialization events are scheduled. The *simulation* class manages the schedulers that control which event should be activated.

Once the simulation is initialized it begins by invoking the *simulation::run* method. The simulation then runs until the *simulation::quit* method (i.e., there are no events left or for a specified number of time units) is invoked. When an event is due to be activated, it is extracted from the scheduler and the main simulation routine executes the code from

the *application operator* of that event. Before executing the events, the simulation clock is updated to the activation time of the current event. Furthermore it maintains a conditional list where events can be put that could not execute but may execute at a future time.

When an event occurs, it can be managed in various ways. The event can be appended to a queue or it can be rescheduled. On its way it changes state. Some of the states as shown in Figure 4.2 can be in: passive, active, queued, pending, conditional etc.

4.3.2 Transaction Generation and Processing

In this section, generation and processing of local transactions followed by global transactions are described.

Local Transaction Generation: The local database simulation involves two processes: the transaction processes and the transaction generator process. The transaction generator process runs for the duration of the simulation and generates new transactions which enter the system at a given interval. Currently, a new transaction enters the system every 1.5 time unit, but this is tunable parameter. Figure 4.3 illustrates the scheduling of the processes in the system. A dashed line represents initiation of a process at startup time. A solid line represents process scheduling that is always performed, and a dotted line represents other scheduling that may occur. Notice that the generator process continually schedules itself while also creating and scheduling new transactions that enter the system. A transaction may also schedule another transaction to run after it releases a resource required by the other transaction.

Local Transaction Processing: Transactions are added to the system by a transaction generation process which generates them at a set interval. The transaction generator loads in a query configuration file consisting of a given number of read and write queries and the probabilities of their occurrence. Queries consist of a list of read/write operations. The transaction generator randomly chooses a query for a transaction to execute.

Transaction execution is performed by executing the read/write operations. Since each transaction is associated with a simulation process, a transaction only runs when it is allowed to proceed by the simulation (i.e., it is not queued for a resource or waiting for work to complete.) When a transaction does run, it may be in one of four phases: INIT, RUN, COMMIT, and ABORT. The INIT phase is used when the transaction first begins executing and registers the transaction with the TM. A transaction is in the RUN phase while it is executing its operations and has not yet completed. When the transaction has completely evaluated its query or must abort for some reason, it enters either the COMMIT or ABORT phases, respectively. These phases call the TM to either commit or abort the transaction. After the commit or abort is completed, the transaction process is removed from the system.

Global Transaction Generation: At the global level, there are two simulation processes: the global transaction process and the global transaction generator process. Both of these processes behave similar to their local database counterparts. Then, the global transaction generation process runs for the duration of the simulation, generating new global transaction processes at a set interval. Global transaction processes are similar to local transactions processes. Each process represents a single global transaction which may run, be blocked waiting for local transaction completion, and then terminate after completion of the global transaction. Figure 4.4 shows how processes are scheduled in the simulation. A dashed line represents an initiation of a process at startup time. A solid line represents process scheduling that is always performed, and a dotted line represents other scheduling that may occur.

Global Transaction Processing: A global transaction is generated periodically by the GT_gen process. The GT_gen process has a set of all the possible global queries and generates them according to their probabilities. After a new GT is created, it is put in the initialization phase and begins its execution.

Initialization depends on the GTM algorithm but may involve submission of some/all of the global sub-transactions. The sub-transactions of a GT are in the form of a list of operations (read or write), which may not be the form that the LDBS expects its transactions. It is the responsibility of the GTS to convert the GST into the suitable form for the LDBS.

Local transactions created from a global sub-transaction enter the prepare-to-commit phase instead of automatically committing. When a local transaction enters the prepare-to-commit phase, it calls its GTS. The GTS reactivates the appropriate GT which may choose to commit or abort the transaction. The local transaction stays in the prepare-to-commit phase until it receives some signal from the GTM.

A GT remains in the RUN phase until all sub-transactions return results or an error occurs. The GT handles the completion of a GST. If the GST was aborted, presumably the GTM should abort and take steps to abort the remaining GSTs. Otherwise, the GST will correspond to a LT in the prepare-to-commit phase which will be holding resources and locks waiting for a signal from the GT. For most GTMs, the GT cannot commit the LT until all other GSTs complete. Therefore, the LT is moved into a holding list which contains all GSTs processed that are in the prepare-to-commit phase, waiting for the rest of the GSTs. If there exists more GSTs not yet completed, the GT passivates itself again waiting to be restarted by the completion of another GST.

When all the GSTs complete, the GT can attempt to commit the GSTs. The GTM validation procedures are now applied. If the GT fails validation, the GSTs are sent the signal to abort, otherwise they are told to commit. Each LDBS will handle the commit, and the GT can be removed from the system. The system does not implement two-phase commit (2PC) as there is no reply from the LDBS on the success of the commit request. It is assumed that there are no transmission errors, and once the GTM sends the signal to commit, all LDBSs will successfully commit.

4.4 Overall Structure of the Simulation Program

Simulation initialization involves setting up the MDBS configuration and initializing the simulation processes. Figure 4.5 details how this is accomplished. Basically, the global structures for the MDBS, GT_gen and GTM are initialized, and GT_gen is started immediately. Each local database is added to the MDBS in a process similar to loading a single local database. It is only the forth step, adding the LDBS to the MDBS, which is new. In this step, a global transaction server (GTS) is created for each LDBS to communicate with the MDBS. Each LDBS provides a list of objects accessible by the global view. The objects the LDBS provides to the MDBS may be a subset of the objects in the local view. After all LDBSs are initialized, a GTM specific initialization routine is called as many GTMs may need to setup structures after seeing the entire MDBS configuration.

The flow of control shown in Figure 4.6 is best described from the perspective of a global transaction. The local flow of control at each LDBS is unaffected by global transactions and vice versa, but it is important to remember that although each LDBS and the global level are sharing the same scheduler, logically there are separate run-time environments for each LDBS and the global level. Each LDBS has its own local transactions and global transactions can only work in a LDBS by issuing local transactions. Thus, each LDBS is a separate logical entity with its own scheduling mechanism. Similarly, the global level scheduling can be considered a separate entity with global transaction generation and execution separate from any local scheduling. Note however that global transaction execution depends on local scheduling decisions as global transactions must submit transactions and retrieve results from local databases.

The life of a global transaction begins when the transaction is generated by the global transaction generator (GT_gen). The GT_gen creates a new global transaction process and sets it to run immediately. The global transaction then begins its execution in the

initialization phase which allows it to initialize its structures and possibly submit some or all of its sub-transactions depending on the GTM. After initialization is completed the global transaction is set to run immediately. However, this does not mean that the global transaction will begin executing immediately. It only means that the next time it is restarted, it will start in the run phase. It is entirely possible that the initialization will passivate the transaction (remove it from execution) for a given time. For example, in the Ticket Method GTM, all local transactions are submitted at the initialization phase, so the GTM passivates the transaction until a local sub-transaction returns with a result.

A global transaction in the run phase means that initialization has been completed and the transaction is running.

In the commit phase, the transaction is first validated. If it fails to commit, it enters the abort phase. After the commit phase completes, the global transaction process is destroyed and removed from the simulation. Otherwise the aborted transaction is reinitialized and resubmitted. A global transaction continues execution in the system until it commits.

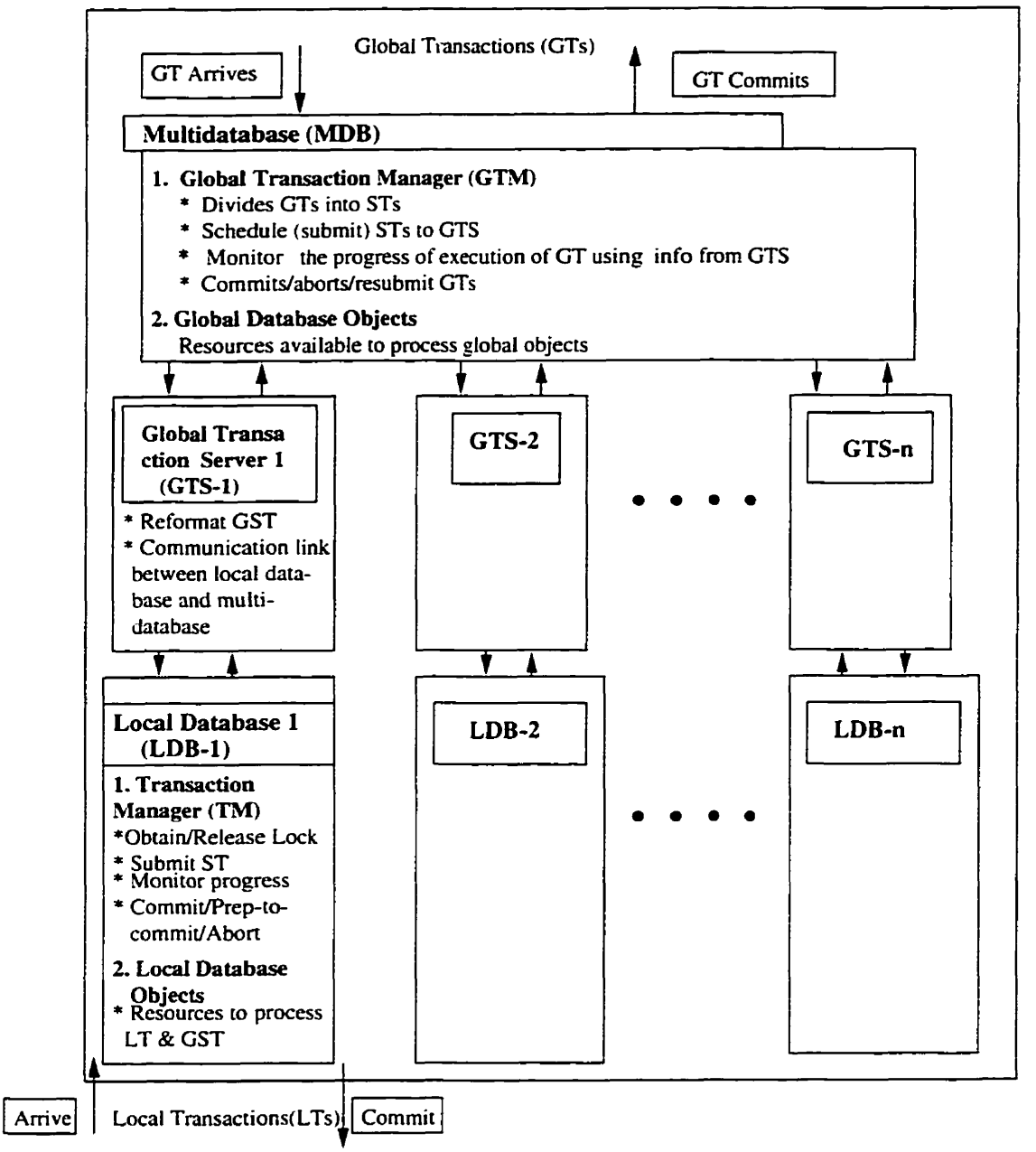


Figure 4.1: Transaction Processing in MDB Environment

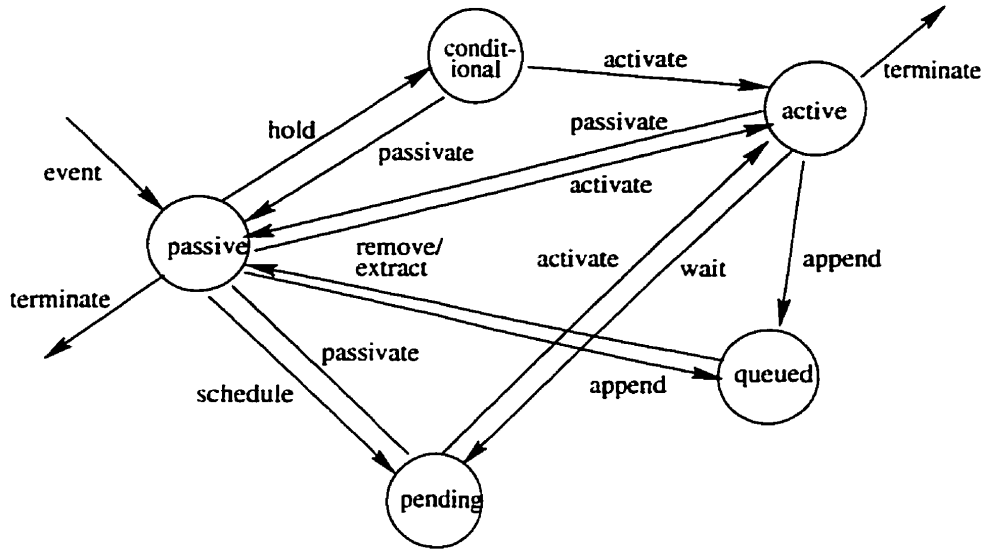


Figure 4.2: Event State Transition Diagram

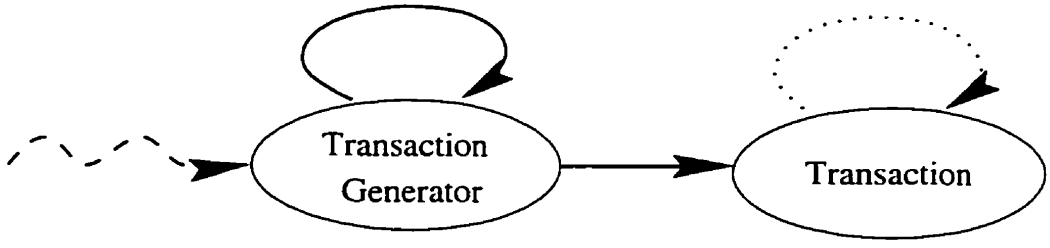


Figure 4.3: Local Process Scheduling

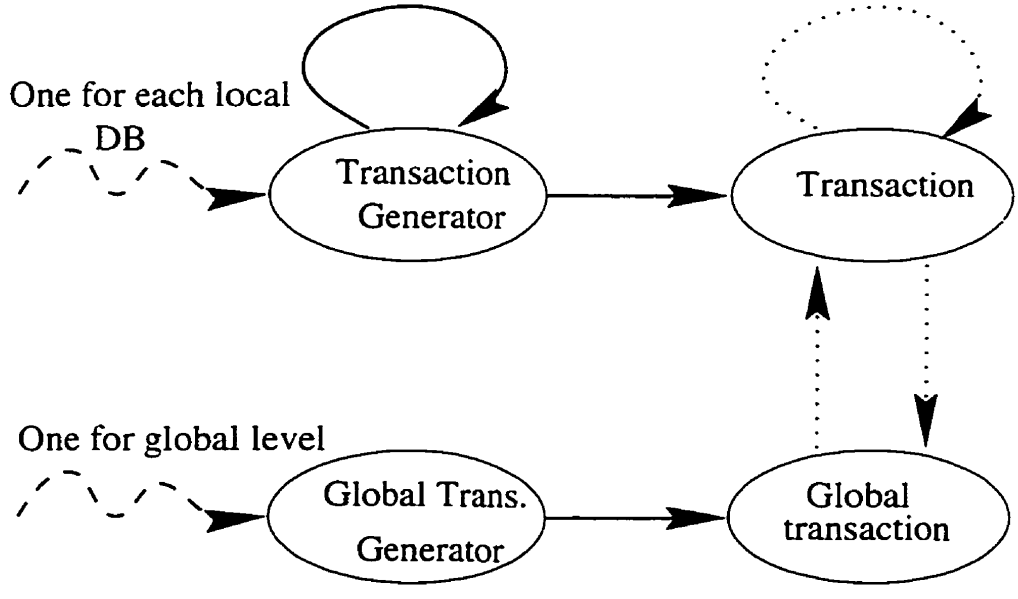


Figure 4.4: Global Process Scheduling

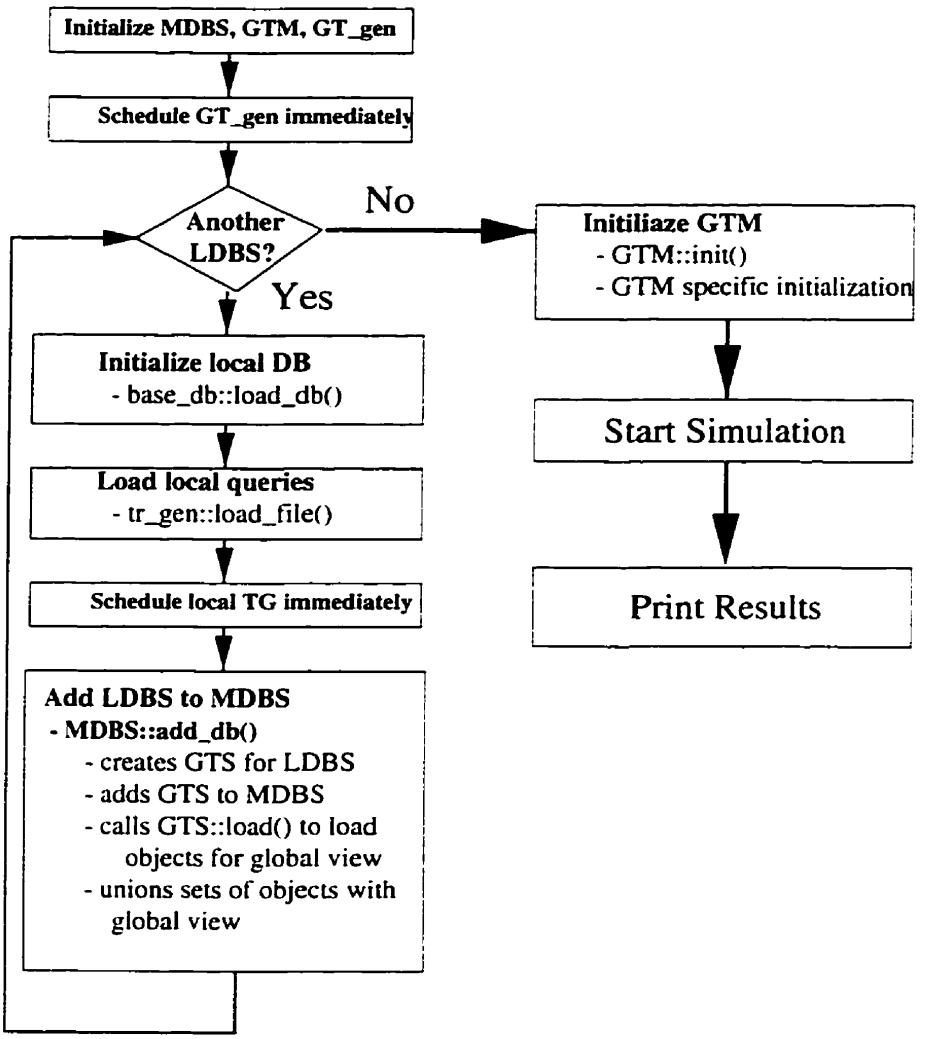


Figure 4.5: Simulation Initialization

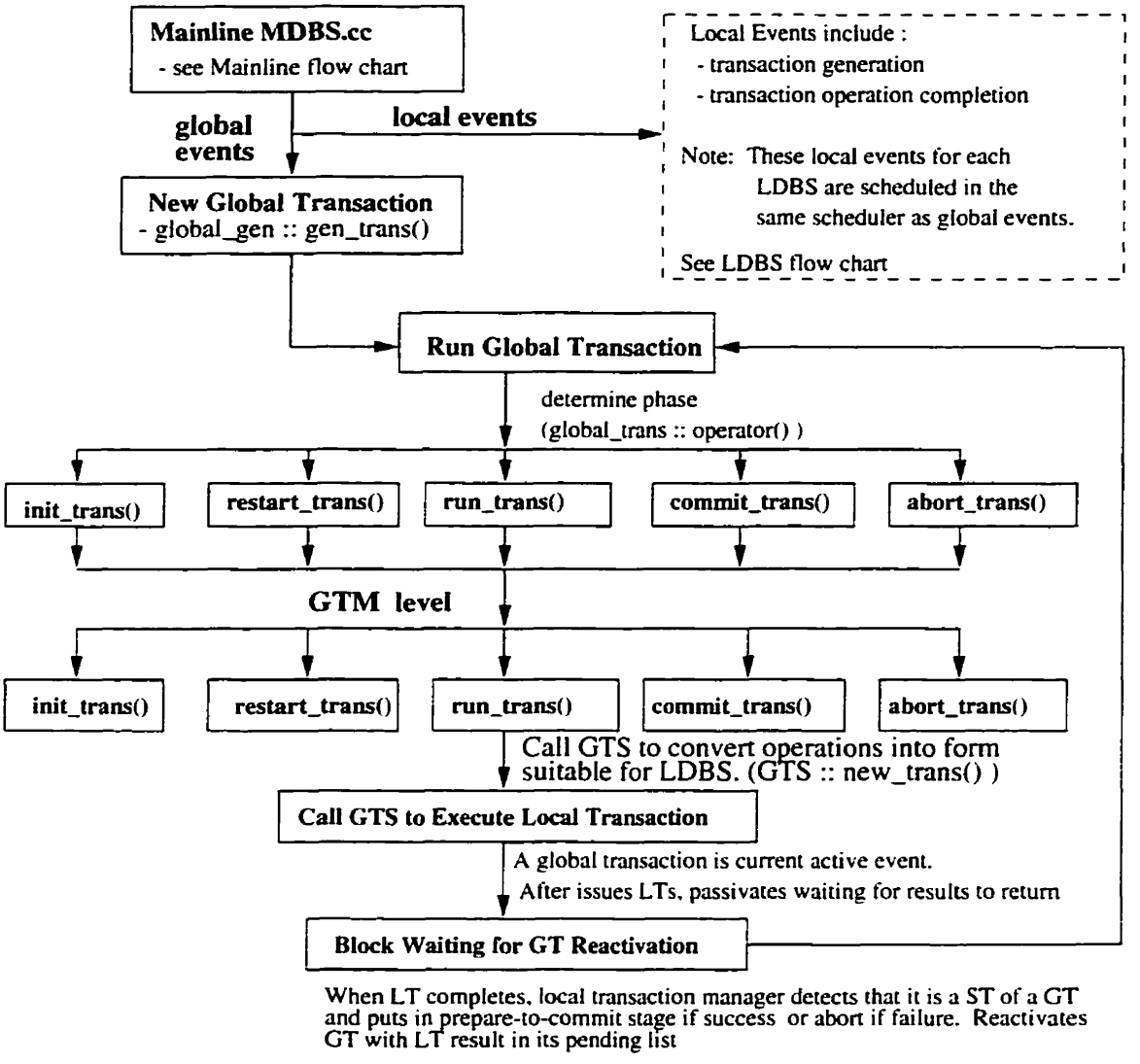


Figure 4.6: MDBS Flow Control

Chapter 5

Performance Comparison of MDB Schedulers

Previous chapters developed a simulation model for multidatabase system, described two existing MDB schedulers and developed two new schedulers. This chapter describes simulation results that compare performance of these schedulers. First, experimental details are described followed by results and discussion.

5.1 Experimental Details

In this section the schedulers are described and evaluated. Performance metrics presented include variables considered in the experiments.

5.1.1 Schedulers Compared

Two existing schedulers, Ticket Method and GSS (Chapter 2) and two new schedulers NGSS-1 and NGSS-2 (Chapter 3) are evaluated. The purpose of describing GSS is to present the basis of development of new schedulers NGSS-1 and NGSS-2 but it is not simulated in this research. Only three schedulers Ticket Method, NGSS-1 and NGSS-2 are compared.

5.1.2 Performance Metrics

Four metrics are used to compare the performance of schedulers.

Residence time: This is the total time a transaction spends in the system, i.e., the time from when a transaction arrives to the time when it commits. This is a very important measure from the point of view of the user (or transaction).

Number of aborts: This metric is considered to check the number of unnecessarily aborted global transactions verses the number of committed transactions. It indicates the amount of re-execution of global transactions that results due to aggressive concurrent submission of global transactions in the Ticket Method. NGSS-1 and NGSS-2 will have no aborts of global transaction because of their pessimistic submission policies.

Utilization: This measures the percentage of time the database was busy. The percentage of time the objects are locked is a measure of utilization of objects. Locks are not released until the last operation of a LT completes in a database in order to enforce serializability at the local level. In the Ticket Method locks are held for an even longer time (until the execution of the last GST or the abort of the global transaction, whichever comes first) to enforce global serializability. Therefore, the time when the database is accessed by transactions is usually shorter than the lock time. We will report both the information on utilization, i.e., lock time and the database access time. Each object in a database can have different utilization levels and so an average value is reported. It should be noted that the GST processed through LDBS but aborted later on by the MDBS, will also be included in the calculation.

Computation Time: The MDB simulator is generic and the transaction manager module can load one of the three schedulers. The CPU time required to run the simulator, using a specific scheduler can be used to indicate the computational time of the scheduler. We report computational time for 100 time units (seconds) of database simulation.

5.1.3 Parameter Setting for Local Database

The local database simulator simulates a database with 6 relations varying in size from 75 to 11000 bytes with average relation size of 3400 bytes. The database processing speed is set at 10000 bytes/sec. Obviously, these numbers are too small to be realistic in today's environment but can be scaled appropriately.

There are ten different queries (local transactions) presented to the database. They are divided into five read and five write queries. The probability of a read query is 0.8 with the remainder being write queries. Each query has an associated probability. The probabilities of all read queries sum to 1, as do the probabilities of all write queries. The average number of bytes accessed over all queries is approximately 10500 bytes. Thus, the average time to execute a query should be 1.05 sec.

5.1.4 Experimental Variables

The following variables were considered in designing the simulation experiments.

- **Load Related Variables:** The database load is affected by processing speed of the databases and the inter-arrival time of global and local transactions as described below.

  (i) **Processing Speed:** If the processing speed of the database is very low (as compared to arrival rates), the database gets overloaded because the queue of waiting transaction will be too long and the service rate will exceed processing power.

  (ii) **Inter-Arrival Time of transaction:** If the inter-arrival time of global and local transactions decreases for a given database the system load increases.

  Thus, the values for the three variables (i) processing speed of local databases, (ii) inter-arrival time of local transactions and (iii) inter-arrival time of global transac-

tions need to be set for each experiment. These should be taken in such a way that the waiting queue (or subsequently the transaction residence time) should not be infinitely long.

Setting Processing Speed: The value of processing speed of LDBS is set at 10000 bytes/second as noted in Section 5.1.3.

Setting Inter-Arrival Time at the LDBS: LDBS processes LT and GSTs submitted through MDBS. In order to set an appropriate level of load at LDBS, the local database simulator was run 10 times for 1000 seconds at processing speed of 10000 bytes/second. Each run has no run-up period and is terminated with a hard-close after 1000 seconds. Statistics were gathered on transaction residence time. As Figure

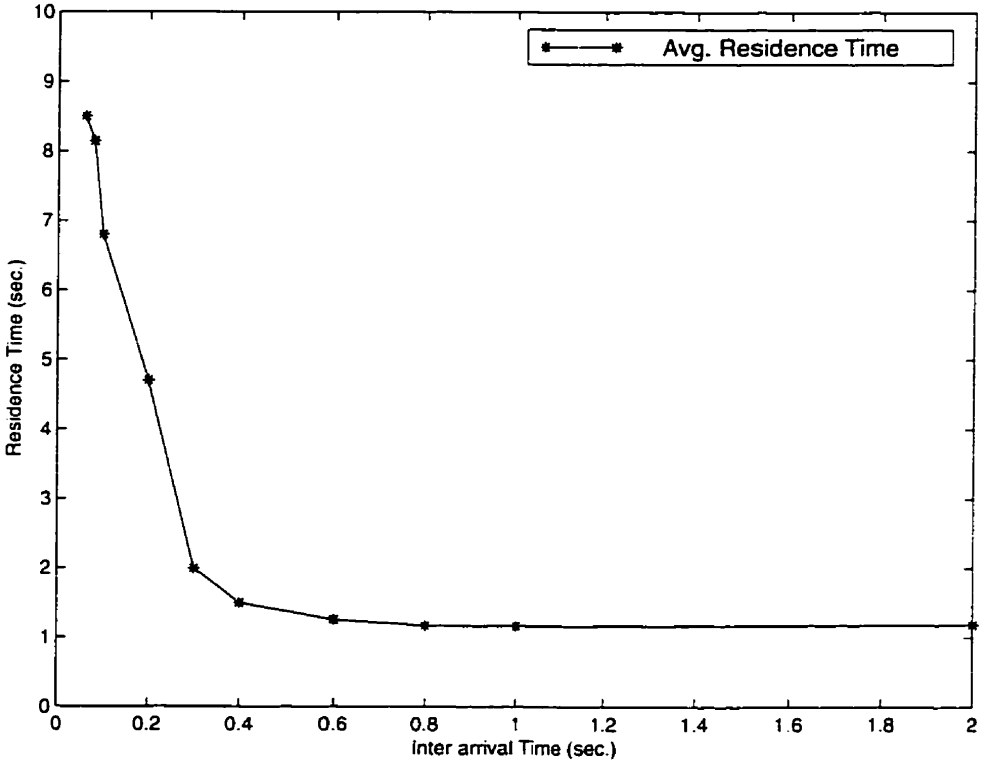


Figure 5.1: Transaction Residence Time Vs. Inter-Arrival Time for LTs

5.1 shows, the average transaction residence time increases rapidly as the inter-arrival time increases beyond 0.3 transactions/sec. Note that the system does not get over-

Inter-Arrival Time (sec.)		Processing Speed (bytes/sec.)
Global Transaction	Local Transaction	
2.0	1.5	10000
2.5	1.5	10000
3.0	1.5	10000
4.5	1.5	10000
7.5	1.5	10000
15.0	1.5	10000
30.0	1.5	10000

Table 5.1: Setting values for load related variables

loaded at an inter-arrival time of approximately 1 second. Although the arrival rate is greater than the service rate, the increased parallelism allowed by executing multiple read transactions simultaneously allows for a higher arrival rate than could normally be achieved.

Now we need to break up this load into equivalent global and local transactions. We need to set the inter-arrival time for local transactions at higher than 1 second to accommodate global transactions while avoiding overloading. Global inter-arrival time, local inter-arrival time and processing speed of LDBSs to be used in the simulation experiment are summarized in Table 5.1.

- **Number of Sub-transactions in Global Transaction:** Global transactions are broken into global sub-transactions to be submitted to individual LDBS. If the number of local databases are more in a multidatabase system, parallelism will be more (more GST can be submitted concurrently).

  In our experiment, we consider three and five LDBSs. The lengths of GTs were chosen using probabilities shown in the Table 5.2.

No. of LDBSs	No. of LDBSs required by a GT	No. of LDBS combinations possible (Variation of GTs defined)	Frequency of each variation of GT
3	1	3	0.067
	2	3	0.200
	3	1	0.200
5	1	5	0.010
	2	10	0.020
	3	10	0.050
	4	5	0.040
	5	1	0.050

Table 5.2: Frequency distribution of different lengths and variations of global transaction.

5.1.5 Parameter Setting for Ticket Method

The Ticket Method GTM has two parameters critical to its performance. They are the timeout value assigned to a global transaction (g_timeout) and the time for global transaction re-submission after abort (g_resubmit). The algorithm is very sensitive to these parameters. If the g_timeout is set too low, global transactions may abort when they are not in global deadlock. If g_timeout is too high, the system suffers from lower concurrency as the time to recognize global deadlock is high. Further, when g_timeout is too high, local database overloading is possible as global transactions hold local resources from local transactions which queue up waiting for the resources. Since the global transactions tend to access about the same number of bytes in each database, g_timeout is made constant over all transactions.

Defining the global transaction resubmit time is even more complex. If g_resubmit is zero, the GTM may overload local databases by continually resubmitting global transactions which cannot complete. Furthermore, besides taking resources and impeding local transactions, these resubmitted global transactions are more likely to continually abort as the delay times at the local databases increase due to overloading. G_resubmit has been de-

fined to be the square of the number of times the transaction has aborted times a constant factor α. Even the choice of this constant factor is very sensitive. In testing for a constant value of 10, the average GT residence time was 4 seconds. However, with constant values of 5 and 20, the average GT residence times were 131 seconds and 32 seconds, respectively.

We experimented on the values of g_timeout and the constant α as (2*g_timeout) in g_resubmit = α*num_aborts^2 for each run and the best found was chosen. It turns out to be better not to limit the growth of g_resubmit (say at some constant 100). This tuning is specific to the MDBS configuration. It is highly unlikely that this tuning can be performed in a general, dynamic MDBS. We ran the simulator using different values of g_timeout and α until 100 GTs were processed. The combination of g_timeout and α which gave the lowest global residence time, were selected as shown in Table 5.3 (for 3 databases) and Table 5.4 (for 5 databases).

5.2 Results

The MDBS simulator was run 10 times (once for 5 databases using ticket method as computation time was excessively high) for 100 global transactions. A global transaction must commit before it is allowed to leave the system, so it may be restarted many times until it commits. After 100 global transactions have been generated, the global transaction generator no longer submits global transactions although the local database transaction generators continually submit local transactions. This is done to see how the different GTMs handle the same load. The set of 100 GTs generated will be exactly the same for three GTMs.

Statistics are gathered on residence time, the number of global transaction aborts, utilization of LDBS and computational time of the scheduler.

Inter-Arrival Time (sec.)		g_timeout (sec.)	α (sec.)	Residence Time (sec.)
Global Transaction	Local Transaction			
2.0	1.5	2	4	354.281
		3*	6*	204.631
		4	8	406.234
2.5	1.5	2	4	495.752
		3	6	146.331
		4	8	348.236
		5	10	780.435
		6*	12*	37.775
		9	18	357.675
3.0	1.5	2	4	376.074
		3	6	9.498
		4	8	34.532
		5	10	118.399
		6*	12*	5.055
		7	14	55.217
4.5	1.5	2	4	108.292
		3	6	2.293
		4*	8*	2.003
7.5	1.5	2	4	2.114
		3	6	9.672
		4	8	3.827
		5*	10*	3.260
		6	12	90.736
15.0	1.5	2	4	7.371
		3	6	1.846
		4	8	1.854
		5*	10*	1.844
		6	12	1.931
		7	14	1.981
30.0	1.5	2	4	2.253
		3	6	2.026
		4	8	1.716
		5*	10*	1.684
		6	12	1.684

* Value selected

Table 5.3: Setting parameter in Ticket Method for three databases.

Inter-Arrival Time (sec.)		g_timeout (sec.)	α (sec.)	Residence Time (sec.)
Global Transaction	Local Transaction			
2.0	1.5	2	4	567.590
		3*	6*	291.247
		4	8	699.022
		5	10	2057.373
		6	12	1173.480
2.5	1.5	2	4	410.717
		3*	6*	292.141
		4	8	846.862
		5	10	2011.043
3.0	1.5	2	4	345.219
		3	6	258.630
		4	8	2.470
		5*	10*	2.448
		6	12	2.448
4.5	1.5	2	4	175.390
		3*	6*	53.722
		4	8	157.769
		5	10	302.741
		6	12	206.014
7.5	1.5	2	4	2.197
		3	6	1.991
		4*	8*	1.952
		5	10	1.952
		6	12	1.952
15.0	1.5	2	4	2.115
		3*	6*	1.890
		4	8	1.890
		5	10	1.890
		6	12	1.890
30.0	1.5	2	4	3.489
		3	6	2.798
		4	8	2.022
		5*	10*	1.944
		6	12	1.944

* selected value

Table 5.4: Setting parameter in Ticket Method for five databases.

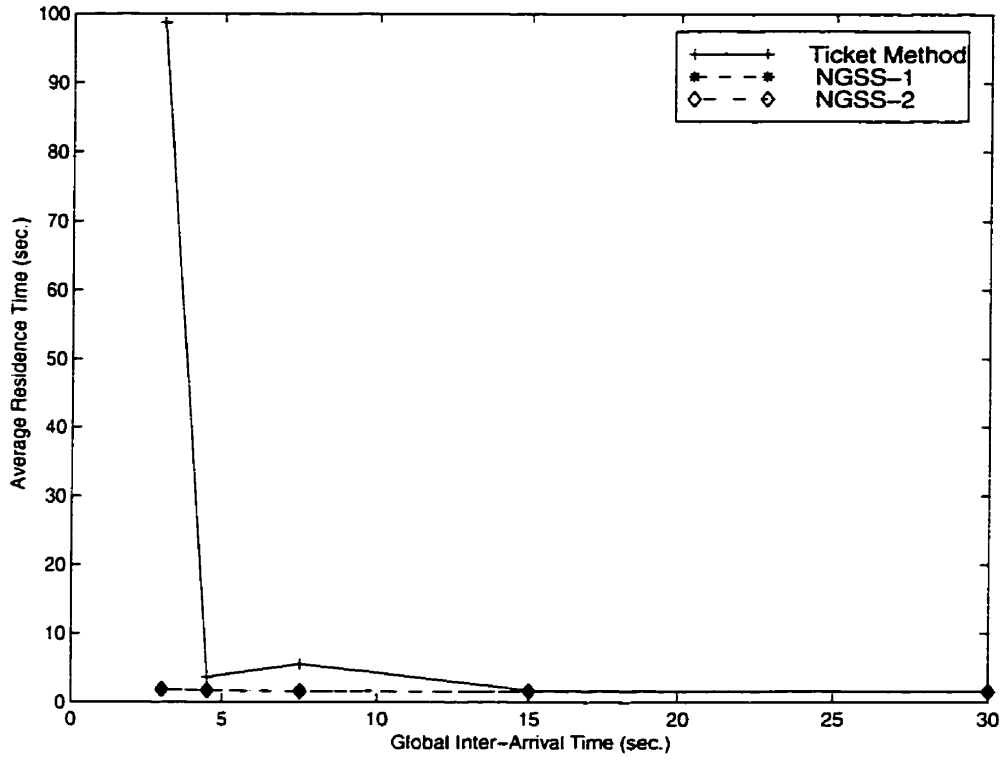


Figure 5.2: Global Transaction Residence Time Vs. Inter-Arrival Time

5.2.1 Residence Time

Global transaction's average residence time and SD (standard deviation of mean values obtained over 10 replications of simulation) for Ticket method, NGSS-1 and NGSS-2 are shown in Table 5.5 and in Figure 5.2. Comparison of residence time of local transactions and global sub-transactions together in the first database are shown in Table 5.6 and in Figure 5.3. It can be observed that as the load increases, the residence time of transactions increases with all the three schedulers. Global residence time is very sensitive to load while local residence time is not too sensitive to global arrival rate. However, Ticket Method has a higher residence time than the two algorithms. Further, at high load, performance of ticket method is poor. As the load decreases (inter-arrival time of GTs increases), its performance is improved as the transactions are processed serially. In NGSS-1 and NGSS-2, the residence time for GTs is consistent because the LDBS does not get overloaded. At

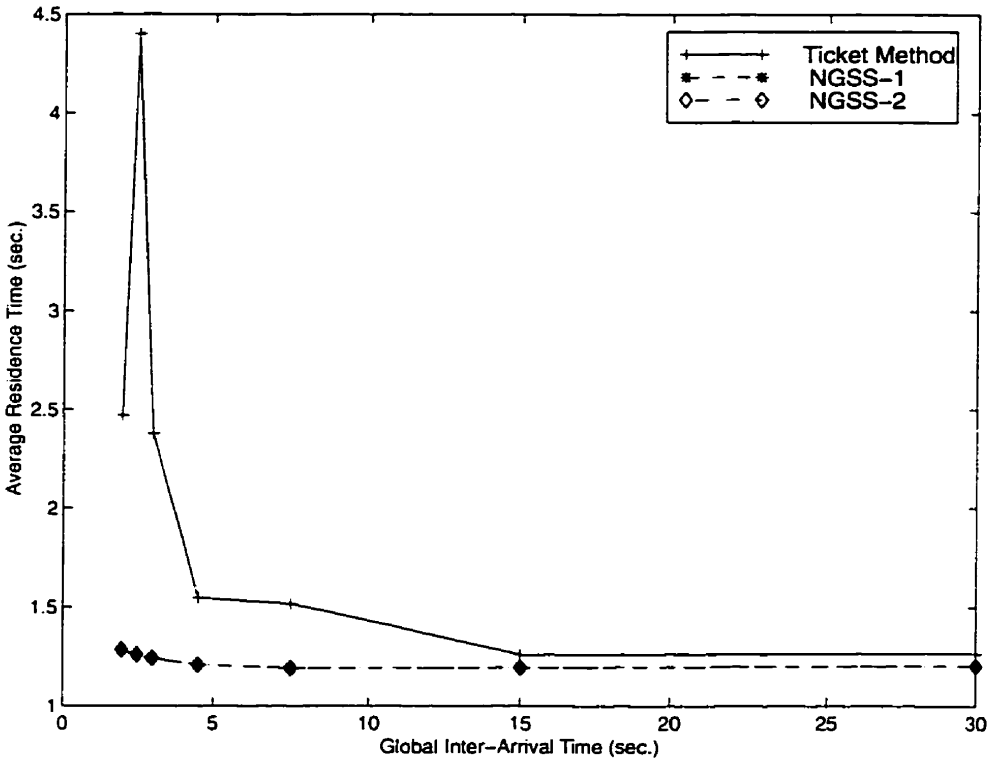


Figure 5.3: Transaction Residence Time Vs. Inter-Arrival Time in LDBS1

much lower load all the schedulers give very similar results.

Between NGSS-1 and NGSS-2, neither gives better results consistently. Although NGSS-2 submits GSTs more aggressively than NGSS-1, the submission test is more rigorous in NGSS-2 than in NGSS-1. Hence, in some cases NGSS-1 is superior while it is inferior in some other cases. However, the difference is not significant since the load is similar in 3 and 5 databases. Further, the residence time is more with 5 databases than in 3 databases for the same load due to the increased processing involved.

5.2.2 Number of Aborts

The overloading of the databases with the Ticket Method causes more global transactions to timeout and abort. The abort rate with the global inter-arrival time is shown in Table 5.7 and in Figure 5.4.

As the load in the database increases or the number of database increases, the abort

No. of Data-base	Inter-Arrival Time (sec.)		Residence Time (sec.)					
	Global Transactions	Local Transactions	Ticket Method		NGSS-1		NGSS-2	
			mean	SD	mean	SD	mean	SD
3	1.5	1.5	*	*	2.634	0.870	2.510	0.831
	2.0	1.5	223.410	154.302	2.114	0.323	2.143	0.347
	2.5	1.5	248.462	202.574	1.919	0.239	1.941	0.238
	3.0	1.5	98.742	201.931	1.819	0.160	1.832	0.165
	4.5	1.5	3.647	5.009	1.687	0.136	1.690	0.134
	7.5	1.5	5.518	6.146	1.571	0.085	1.570	0.084
	15.0	1.5	1.679	0.209	1.534	0.083	1.533	0.084
	30.0	1.5	1.560	0.095	1.534	0.100	1.534	0.100
5	1.5	1.5	*	*	5.252	0.524	5.373	0.647
	2.0	1.5	2357.165	**	3.888	0.639	4.149	0.903
	2.5	1.5	1734.044	**	3.062	0.179	3.081	0.246
	3.0	1.5	130.047	**	2.819	0.330	2.747	0.327
	4.5	1.5	478.657	**	2.428	0.104	2.403	0.118
	7.5	1.5	4.172	**	2.189	0.370	2.171	0.053
	15.0	1.5	2.038	**	2.063	0.048	2.053	0.044
	30.0	1.5	2.062	**	2.037	0.048	2.034	0.047

* Execution was aborted as run could not be completed in few hours at this load.

** Ticket method was run for only one replication due to excessive computation time.

Table 5.5: Comparison of Global Residence Time

No. of Databases	Inter-Arrival Time (Sec.)		Average Residence Time (sec.)					
	Global Transactions	Local Transactions	Ticket Method		NGSS-1		NGSS-2	
			mean	SD	mean	SD	mean	SD
3	1.5	1.5	*	*	1.252	0.063	1.241	0.061
	2.0	1.5	2.470	0.315	1.284	0.072	1.285	0.074
	2.5	1.5	4.401	2.787	1.256	0.086	1.260	0.086
	3.0	1.5	2.378	1.015	1.241	0.040	1.241	0.040
	4.5	1.5	1.547	0.251	1.209	0.034	1.209	0.034
	7.5	1.5	1.517	0.356	1.192	0.027	1.192	0.027
	15.0	1.5	1.260	0.033	1.195	0.016	1.195	0.016
	30.0	1.5	1.266	0.017	1.203	0.016	1.203	0.016
5	1.5	1.5	*	*	1.502	0.059	1.486	0.046
	2.0	1.5	1.829	**	1.414	0.035	1.524	0.123
	2.5	1.5	1.843	**	1.413	0.064	1.411	0.049
	3.0	1.5	1.930	**	1.388	0.055	1.381	0.032
	4.5	1.5	1.943	**	1.345	0.045	1.349	0.053
	7.5	1.5	1.409	**	1.270	0.030	1.269	0.029
	15.0	1.5	1.262	**	1.226	0.009	1.224	0.010
	30.0	1.5	1.216	**	1.218	0.015	1.217	0.015

* Execution was aborted as run could not be completed in few hours at this load.

** Ticket method was run for only one replication due to excessive computation time.

Table 5.6: Comparison of Local Residence Time (combined LTs and GSTs) in the first database.

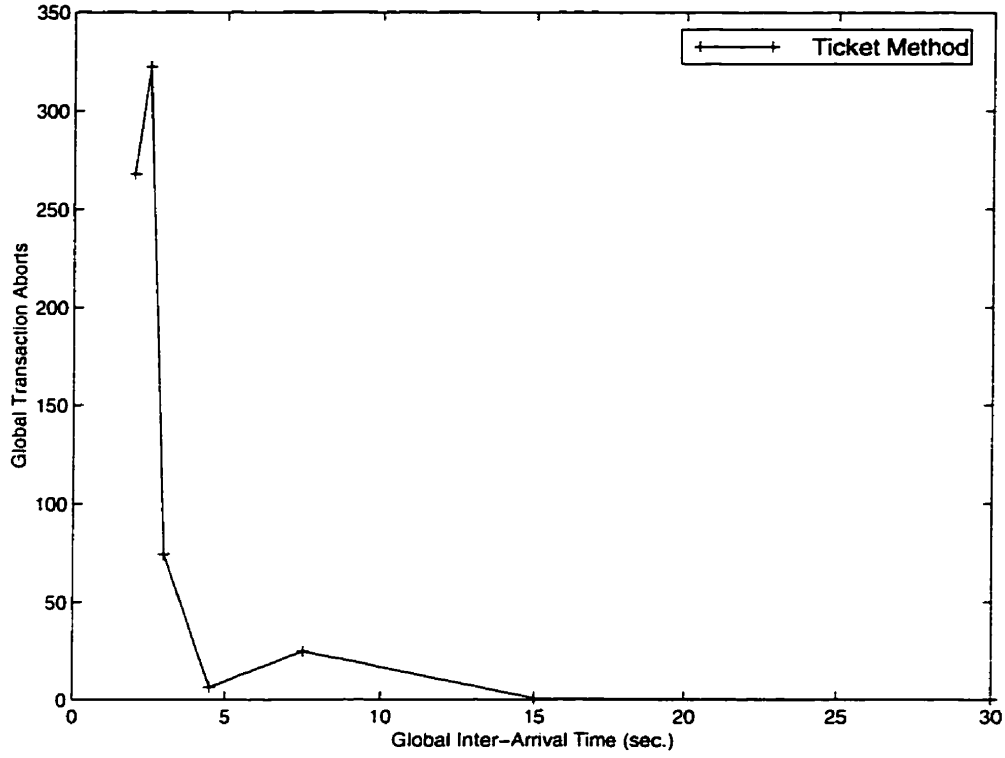


Figure 5.4: No. of Aborts Vs. Inter-Arrival Time in Ticket Method

rate of transactions also increases. The long wait and processing time causes more global transactions to timeout and hence abort.

5.2.3 Utilization

Table 5.8 and Figure 5.5 shows the comparison of utilization of objects for the three schedulers in the first database (selected arbitrarily). The Ticket Method shows a high percentage of lock time minus access time as compared to the other two schedulers. Locks on objects are held (for deadlock detection) until all the sub-transactions are completed in the Ticket Method. On the other hand there is no cycle check in the two serial schedulers and locks are released as a sub-transaction completes. Hence, lock time is less in serial schedulers as compared to Ticket Method. At reduced load all the three algorithms give similar utilization values due to the serial execution of global transactions.

Inter-Arrival Time (sec.)		Average number of Aborts in 3 database		in 5 database
Global Transaction	Local Transaction	mean	SD	mean
2.0	1.5	268.0	91.75	721.0
2.5	1.5	322.4	249.62	635.0
3.0	1.5	74.5	131.66	84.0
4.5	1.5	6.4	10.62	371.0
7.5	1.5	24.8	24.12	11.0
15.0	1.5	0.7	1.16	0.0
30.0	1.5	0.1	.32	0.0

Table 5.7: Average global transaction aborts in Ticket Method

No. of Databases	Inter-Arrival Time (Sec.)		Lock time (database access time) (%)		
	Global Transaction	Local Transaction	Ticket Method	NGSS-1	NGSS-2
3	1.5	1.5	*	39.93 (18.36)	40.03 (18.40)
	2.0	1.5	41.80 (13.82)	33.73 (16.48)	33.73 (16.48)
	2.5	1.5	41.51 (13.52)	33.92 (16.30)	33.92 (16.30)
	3.0	1.5	46.93 (15.30)	30.70 (14.92)	30.70 (14.92)
	4.5	1.5	34.40 (14.08)	28.13 (14.05)	28.13 (14.05)
	7.5	1.5	30.95 (13.34)	25.90 (13.06)	25.90 (13.06)
	15.0	1.5	26.78 (12.82)	24.75 (12.79)	24.75 (12.79)
	30.0	1.5	24.77 (12.37)	23.88 (12.37)	23.88 (12.37)
5	1.5	1.5	*	38.88 (18.32)	38.00 (18.37)
	2.0	1.5	29.67 (12.40)	34.12 (16.76)	34.25 (16.71)
	2.5	1.5	30.05 (12.44)	33.35 (16.59)	33.93 (16.58)
	3.0	1.5	31.78 (12.53)	30.73 (15.06)	30.51 (15.06)
	4.5	1.5	32.02 (12.54)	28.88 (14.15)	28.86 (14.15)
	7.5	1.5	33.06 (13.16)	25.23 (13.12)	26.81 (13.12)
	15	1.5	26.71 (12.82)	24.80 (12.81)	24.80 (12.81)
	30	1.5	24.83 (12.38)	23.88 (12.38)	23.88 (12.38)

* Execution was aborted as run could not be completed in few hours at this load

Table 5.8: Comparison of Utilization

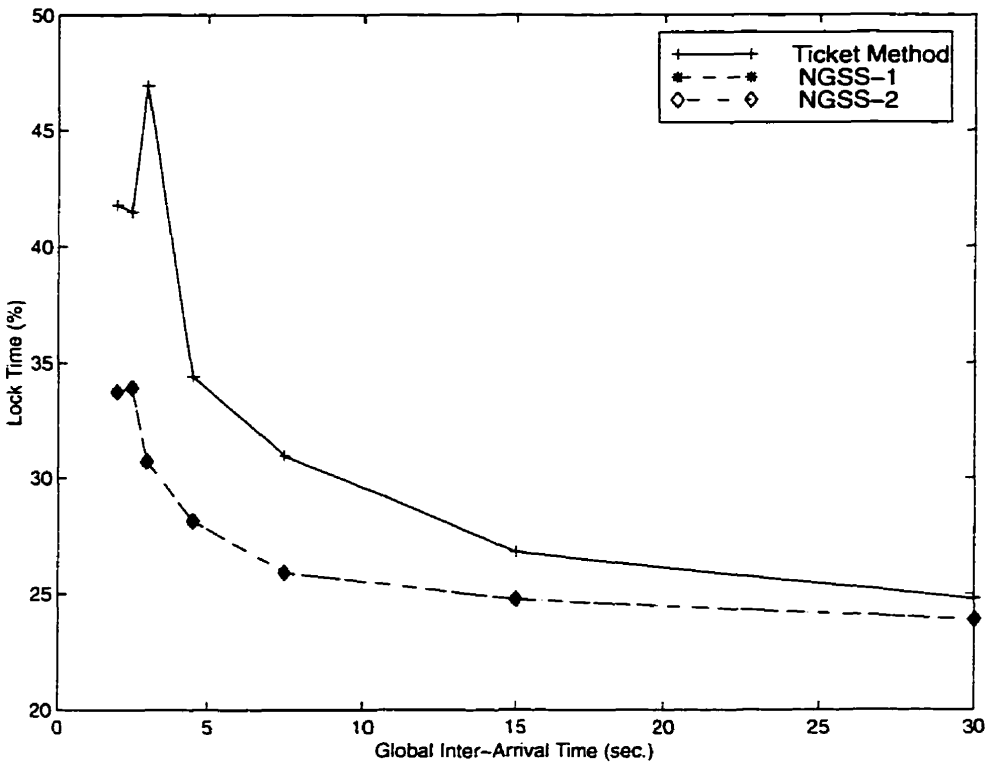


Figure 5.5: Utilization Vs. Inter-Arrival Time

5.2.4 Computational Time

The ticket method processes more transactions because there is no global flow control which leads to a large number of aborts thereby reducing actual processing of transaction.

The computational time is small at extremely low load as shown in Figure 5.6 and Table 5.9. At higher load, there appears to be no definite trend. The computational time is generally more for 5 databases than in 3 databases. Based on the computation time the schedulers can be ranked (increasing order) as the Ticket Method, NGSS-2, NGSS-1. Computation time of serial schedulers are more than with the Ticket Method because it requires more rigorous tests than NGSS-1 and NGSS-2.

No. of Databases	Inter-Arrival Time (Sec.)		CPU Time (sec.)		
	Global Trans.	Local Trans.	Ticket Method	NGSS-1	NGSS-2
3	1.5	1.5	*	4.3385	4.3462
	2.0	1.5	2.0536	3.2698	2.8026
	2.5	1.5	2.1696	4.8671	4.4927
	3.0	1.5	3.1117	4.6677	4.3565
	4.5	1.5	2.2955	1.8782	1.6695
	7.5	1.5	4.1389	4.3898	4.1390
	15.0	1.5	4.458	1.3814	1.3814
	30.0	1.5	1.3822	1.256	1.2251
5	1.5	1.5	*	8.9013	9.5233
	2.0	1.5	4.0626	5.5178	5.8920
	2.5	1.5	4.8244	5.5409	4.7894
	3.0	1.5	6.9323	4.0453	4.0453
	4.5	1.5	2.3379	3.5503	5.8476
	7.5	1.5	3.0114	2.7605	2.5095
	15.0	1.5	2.4492	2.2609	2.2609
	30.0	1.5	2.1364	2.1362	2.1362

* Execution was aborted as run could not be completed in few hours at this load

Table 5.9: Comparison of Computational (CPU)Time for 100 seconds of Simulation

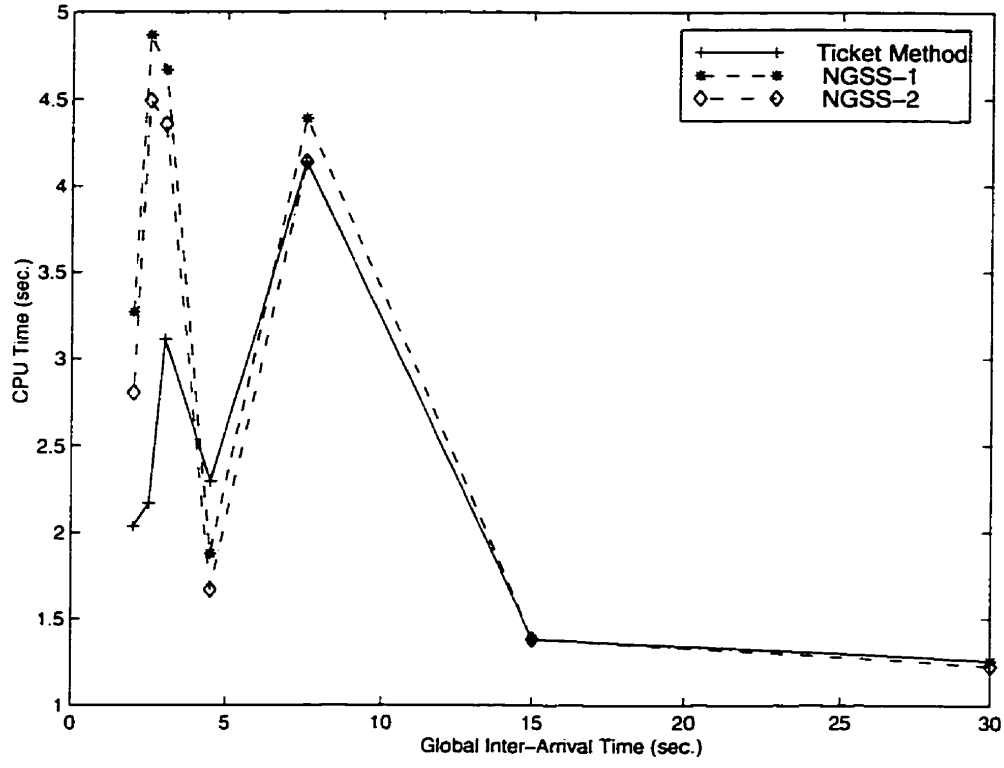


Figure 5.6: CPU Time Vs. Inter-Arrival Time

5.3 Discussion

The ticket method allows all the global transactions to be submitted as soon as a global transaction arrives. The ticket in each LDBS and the GSG check insures global serializability, but there is no global flow control. In other words, all the sub-transactions are submitted regardless of the load on the LDBSs. This often results in lower concurrency as the LDBSs become overloaded.

It is noticeable that even at moderate LDBS loads, the residence times for global transactions are high and highly variable. This is because sub-transactions of global transactions are constantly competing with each other for resources, especially the ticket resource. Also, the constant re-submission after aborts often overloads the LDBSs which further exasperates the problem.

The tuning of the Ticket Method as described previously is fairly difficult and has a

great effect on its performance. Unfortunately, this tuning is not very robust and can easily fall apart as LDBSs loads increase/decrease, global transaction inter-arrival time change, or the query mix changes. Thus, the Ticket Method is not very robust and suffers from poor performance in the general case. Although, the algorithm is simple and offers the potential for higher concurrency, the lack of global-level flow control often overwhelms the LDBSs causing many global transaction conflicts and aborts leading to an overall weaker performance.

The NGSS-1 and NGSS-2 are deadlock-free and do not allow global transaction aborts. This allows the average and maximum GT residence times to be fairly consistent. At the local level, these algorithms do not cause local aborts, and the global residence time at an LDBS does not depend on the global inter-arrival time except a little at very low inter-arrival time.

Thus, the total residence time of all LDBS transactions is not effected by the global inter-arrival time (except a little at very low inter-arrival time). The stability of the residence time at the LDBSs arises because both NGSS-1 and NGSS-2 algorithms implement both concurrency control and flow-control at the global level. Global sub-transactions are only submitted when no conflicts can arise which limits the number of global sub-transactions active at any LDBS. Consequently this prevents the MDBS from overloading a LDBS with global sub-transactions. Although performance may be limited slightly by executing some GSTs serially, this performance is more than made up for by limiting the burden placed on the LDBSs by global transactions. Thus, global sub-transactions that are submitted to a LDBS can execute faster than if they were competing for the same resources with other global sub-transactions. Global-level flow-control is especially important when the global transaction inter-arrival time is low and when one or more LDBSs are heavily loaded.

In terms of performance, there is no comparison between the Ticket Method and serial schedulers (NGSS-1 and NGSS-2). The possible higher concurrency, for which the Ticket

Method algorithm was designed to allow, ends up being a determinant to its performance. It suffers from frequent transaction aborts, local database overloading, and performance loss through global deadlocks. All these factors significantly reduce the concurrency and performance.

The NGSS-1 and NGSS-2 have better performance because they control the flow of global transactions entering the LDBSs. Although this may reduce concurrency in some situations, it does not cause global deadlock, LDBS overloading, or global transaction aborts.

In terms of implementation, the NGSS-1 and NGSS-2 are also much easier to build and configure. They are highly robust and only slightly effected by increases in LDBS load, query mix changes, or varying global transaction submission rates. The Ticket Method algorithm is highly susceptible to performance concerns if the deadlock detection time "timeout" and the restart time "resubmit time" are not properly configured. Unfortunately, the performance varies wildly even within the same configuration in multiple runs. The Ticket Method supports a visual prepared-to-commit state. This violates autonomy of local databases [Bar94]. Further, it is not easy to properly configure the system to handle changing MDBS conditions. For example, the NGSS-1 and NGSS-2 can easily handle local transaction inter-arrival time of less than one second for the given MDBS configuration. The Ticket Method algorithm does not even complete the simulation for such values as it gets stuck in long cycles of global aborts/restarts.

Both the NGSS-1 and NGSS-2, exhibit similar performance although NGSS-2 submits transactions more aggressively than NGSS-1. However, the computation time required by NGSS-1 is generally higher than NGSS-2. This is because in NGSS-2 submission criterion is more rigorous than in NGSS-1. NGSS-1 submits all the sub-transactions of a transaction or none which holds all the sub-transactions of a transaction until all the sub-transactions can be submitted. NGSS-2 submits each sub-transaction of a transaction individually so

the sub-transactions do not have to wait for all the sub-transactions to submit. In this way, NGSS-2 should be faster than the NGSS-1. However, the criterion for submission in NGSS-2 is more rigorous and some of the sub-transaction of a transaction have to wait for long time for submission or the completion of other sub-transactions.

Chapter 6

Conclusions

In a multidatabase system, transaction management is a major issue because it is very difficult to have a good concurrency control algorithm without violating local autonomy. Some scheduling algorithms have been proposed to handle concurrency control, but no independent simulation or testing has been reported to evaluate their performance.

In this thesis, schedulers that employ two distinct approaches are considered. In one approach, transactions are submitted concurrently, but, to ensure the consistency (serializability), transactions may have to be aborted and restarted several times. The other follows a much less concurrent (almost serial) submission policy but guarantees that such submission will not lead to serializability problems and hence no aborts. We considered the Ticket Method [GRS94] in the first category and developed two serial schedulers NGSS-1 and NGSS-2 following the lines of GSS developed by Barker [Bar90]. We have developed a simulation model for multidatabase system to study the performance of these scheduling algorithms. Simulation results indicate the following:

- Residence time of transactions increases as load increases (inter-arrival time decreases) and length of global transaction increases with all the schedulers. However, the residence time of local transactions and global sub-transactions are less sensitive to these factors compared to that of global transactions. The number of aborts resulting using the Ticket Method also increases with load and transaction length.

- Performance of serial schedulers (NGSS-1 and NGSS-2) is much better than the Ticket Method, especially at higher load. At very low load they all give similar results as the transactions are processed as they come, i.e., serially.

- Difference in performance (residence time) of serial schedulers NGSS-1 and NGSS-2 is not very significant. However, NGSS-2 requires lower CPU time than NGSS-1.

- In terms of implementations, serial schedulers are much easier to build and configure. The Ticket Method is highly susceptible to tuning parameters such as "timeout" and "resubmit time".

The Ticket method implementation requires a visual prepared-to-commit state. This violates autonomy. Serial schedulers do not have the above requirement, thereby allowing more autonomy.

6.1 Future Research

Some possible extensions of this thesis could be the following. Since results show the transaction manager which is based on concurrent transaction submission has poor performance, it would be interesting to determine if other global transaction scheduling algorithms which submit transactions concurrently have the same performance liabilities as the Ticket Method GTM.

The findings of this research suggest that it is worthwhile to concentrate on developing schedulers that submit transactions to guarantee that they will not be aborted at the cost of less concurrency.

As indicated in the thesis, parameter setting in Ticket Method for the GSG check for deadlock detection is difficult. A method for deadlock detection suggested in [Tri97] seems to give better results when integrated with Ticket method. A detailed simulation study can be conducted to determine the performance of this method.

Currently, the FCFS rule has been used in processing the queue of NGSS-1 and NGSS-2. Other queue processing techniques such as longest job first, shortest job first could be tested to see if they further improve performance.

Currently, Strict 2PL TM is implemented at local databases to evaluate all GTM (schedulers). Performance of the system when LDBS uses other concurrency control methods such as Time stamp Ordering or Optimistic Concurrency Control can be studied.

Bibliography

[ADW92] D. Adler, B. Dageville, and K. F. Wong. A C-based Simulation Package. Technical report, ECRC, Munich, 1992.

[Bar90] K. Barker. *Transaction Management on Multidatabase Systems.* PhD thesis. University of Alberta, 1990.

[Bar94] K. Barker. Quantification of Atonomy on Multidatabase Systems. *Journal of Systems Integration*, 4:151–169, 1994.

[BE95] D Bolier and A. Eliens. Simulation Modeling Support for Discrete Event Simulation in C++. Technical report, Vrije Universiteit, 1995.

[BGMS95] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Transaction Management in Multidatabase Systems. *Modern Database Systems*, pages 573–591, 1995.

[BHG87] P. Bernstein, V. Hadzilacos, and N Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., 1987.

[BHP92] M. W. Bright, A.R. Hurson, and S. H. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *IEEE*, 1992.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill Book Company, 1984.

[DE89] W. Du and A.K. Elmagarmid. Quasi Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 347–355, Amsterdam, August 1989.

[GRS91] D. Georgakopolous, R. Rusinkiewicz, and A. Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 286–293, April 1991.

[GRS94] D. Georgakopolous, M. Rusinkiewicz, and A. Sheth. Using Tickets to Enforce the Serializability of Multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):1–15, 1994.

[HFNL96] L. Henschen, C. Fernandes, T. Neild, and W.S. Li. Modeling Data Correspondence in Multidatabase Systems. In *Proceedings of the International Conference on Intelligent Information Management Systems (IIMS96)*, pages 19–23. Washington, D.C., USA, June 5-7 1996.

[LZ88] W. Litwin and A. Zeroual. Advances in Multidatabase Systems. In *Proc. European Teleinformatics Conference-Euteco 88, Research into Networks and Distributed Applications*, pages 1137–1151, North-Holland, Amsterdam, 1988.

[MRB+92] S. Mehrotra, R. Rastogi, Y. Breitbart, H. F. Korth, and A. Silberschatz. The Concurrency Control Problem in Multidatabases: Characteristics and Solutions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 288–297, San Diega, California, June 1992.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice hall, 1991.

[SLSV95] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3):325–363, September 1995.

[SO93] J. Shillington and M.T. Ozsu. Permeable Transactions and Semantics-Based Concurrency Control for Multidatabases. In *Proc. of the 3rd International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 245–248, Vienna, Austria, April 1993.

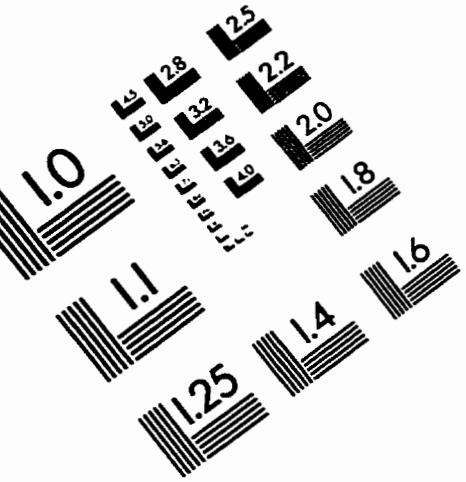
[SSM96] H. Swaminathan, T. Spracklen, and J. J. Mathieu. Aspects of Model Design and Development through Simulation Software. In *Proceedings of the 1996 Simulation Multiconference*, pages 146–151, April 8-11 1996.

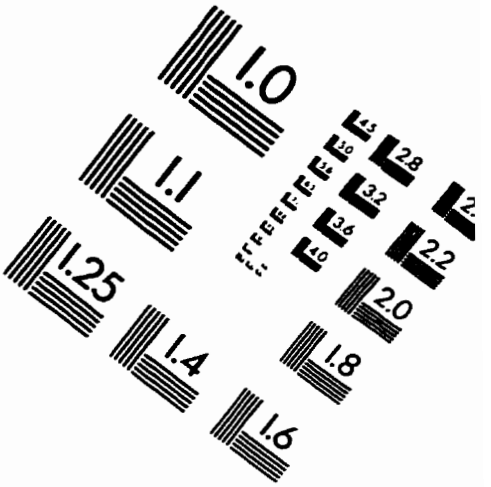
[Tri97] P. Triantafillou. An Approach to Deadlock Detection in Multidatabases. *Information Systems*, 22(1):39–55, 1997.

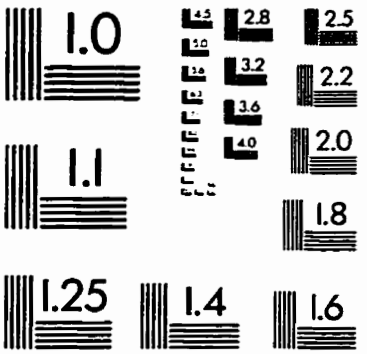
[ZE93] A. Zhang and A. K. Elmagarmid. on Global Transaction Scheduling Criteria in Multidatabase Systems. *VLDB Journal*, 2(3):331–360, 1993.

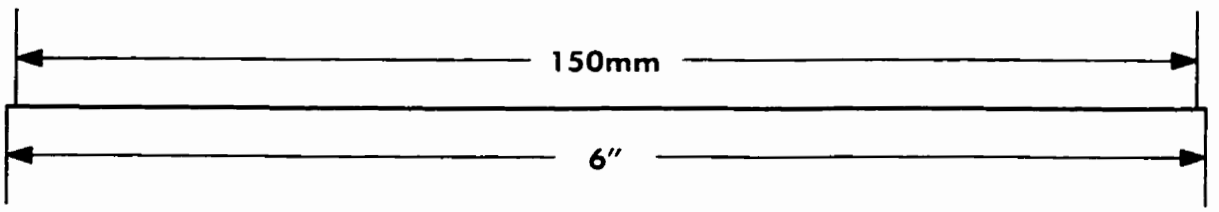
[ZYL95] A. Zaslavsky, L. H. Yeo, and S. J. Lai. Transaction Processing Simulation in Mobile Computing Environment Using Petri Nets. In *IASTED International Conference on Modeling and Simulation*, pages 373–375, Pittsburgh, USA, April 1995.

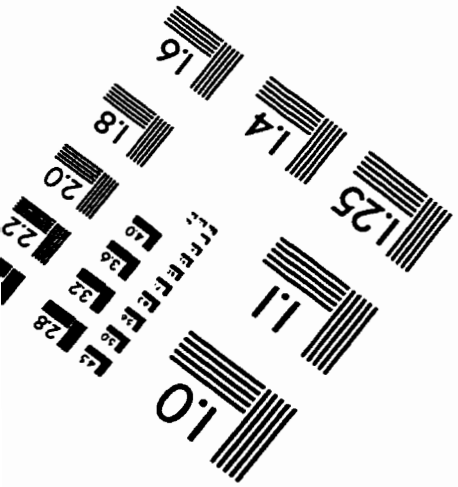
IMAGE EVALUATION TEST TARGET (QA-3)

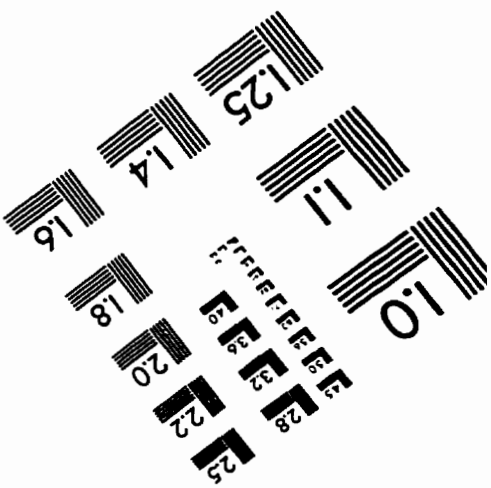












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989