Deadlock Detection in Multidatabase Systems: a Performance Analysis

Roberto Baldoni, Silvio Salza

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Deadlock Detection in Multidatabase Systems: a Performance Analysis

Roberto Baldoni, Silvio Salza

N° 2668

Septembre 1995

PROGRAMME 1

ISSN 0249-6399

Deadlock Detection in Multidatabase Systems: a Performance Analysis *

Roberto Baldoni **, Silvio Salza ***

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projets Adp

Rapport de recherche n ° 2668 — Septembre 1995 — 21 pages

Abstract: Deadlock detection is an interesting problem in MultiDataBase Systems (MDBS), since if all local transaction managements are blocking and force direct conflicts between global transactions, the problem of ensuring the global serializability in the MDBS is reduced to detecting and resolving global deadlocks.Unfortunately the autonomy of the local systems precludes the visibility of the state of local transactions and the contention on items, and therefore the classical approaches proposed for homogeneous distributed database systems, and based on necessary and sufficient conditions, cannot be extended to the MDBS case. A few specific methods have been proposed in the literature that exploit weaker necessary conditions to detect *potential* global deadlocks, that not necessarily correspond to real ones. In this paper we present a comparative performance study of several global deadlock detection methods. The results of the analysis have suggested a new *Hybrid Deadlock Detection* method, that we present in the paper and that is very well suited for a distributed implementation and has a performance that , according to our experiments, compares favorably with all the other methods in a variety of workload conditions.

Key-words: Distributed Deadlock Detection, Concurrency Control, Multidatabase Systems, Performance Evaluation.

(Résumé : tsvp)

*Work partially supported by the the *Consiglio Nazionale delle Ricerche* under contract No.93.02294.CT12 and by the following Basic Research Action Programs of the European Community: the HCM project under contract No.3702 "CABERNET" and the ESPRIT project under contract No. 6360 "BROADCAST".

**IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France.

***Dipartimento di Informatica e Sistemistica, University of Rome "La Sapienza", via Salaria 113, I-00198 Rome, Italy.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Détection de l'interblocage dans une base de données hétérogène: une évaluation des performances

Résumé : Dans un contexte des bases de données hétérogènes (MDBS), la détection de l'interblocage est un problème intéressant puisque si tous les traitements des transactions locales sont bloquants et créent des conflits entre les transactions globales, le problème de sérialisabilité dans les MDBS se réduit à la détection et à la résolution des interblocages globaux. Cependant, l'autonomie des systèmes locaux empêche de voir l'état des transactions locales et les conflits d'accès et rend par la même inopérantes les approches classiques proposées pour les systèmes de base de données homogènes. On trouve dans la littérature quelques méthodes spécifiques exploitant les conditions faibles nécessaires à la détection d'interblocages globaux potentiels mais ceux-ci ne correspondent pas nécessairement aux interblocages réels. Dans cet article, nous présentons une étude comparative des performances de plusieurs méthodes de détection d'interblocages globaux. Les résultats de cette comparaison ont suggéré une méthode hybride bien adaptée à une mise en oeuvre distribuée et qui présente, selon nos expérimentations, des performances soutenant largement la comparaison avec les autres méthodes.

Mots-clé : Détection répartie d'interblocages, contrôle de la concurrence, bases de données hétérogènes, évaluation de performances.

1 Introduction

A *MultiDataBase System (MDBS)* allows access and manipulation of data stored on several pre-existing, autonomous and (possibly) heterogeneous *Local DataBase Systems (LDBS)*. Transactions in a MDBS may be either local or global. Local transactions are directly submitted to individual LDBS and access only local data, while global transactions access data from several LDBS by running local *subtransactions* under the control of the MDBS. The main problem in MDBS transaction management is to deal with the autonomy of local systems [Vei90]: *design autonomy* since the structure of each LDBS cannot be modified, and *execution autonomy* since each LDBS has complete control and does not give visibility on the state of local transactions. As a consequence of this a hierarchic structure has to be adopted for global transaction management, with *Local Transaction Managers (LTM)* controlling the execution of local transactions and ensuring local consistency and a *Global Transaction Manager (GTM)* controlling the execution of global transactions to ensure the correctness and consistency of their concurrent execution.

Even though several alternative consistency criteria have been proposed in the literature, such as *quasi-serializability* [DE89], *two-level serializability* [MRKS91] and *RS-correctness* [MRKS92], we consider in this paper *global serializability*, as it is the most widely used criterion for multidatabase consistency. According to this criterion the GTM validates a global schedule only if it is globally serializable, i.e. if the relative serialization order of all conflicting subtransactions of any two global transactions is the same at each LDBS where they execute [BS88]. Otherwise the schedule is not validated and some of the global transactions are aborted.

Hence, to ensure global serializability, the GTM should actually know the local serialization order of all subtransactions. However, these orders, because of LDBS autonomy, are out of the control and the visibility of the GTM which can only know the relative execution orders of subtransactions at each local site. But it has been proved that the serialization order corresponds to the execution order of transactions in an LDBS, under the restrictive condition that the LTM produces *rigorous* schedules, since it then forces all conflicts (write/write and read/write) between uncommitted transactions to be direct conflicts [BGRS91]. Under the weaker assumption of LTM producing *strict* schedules [BHG87], the serialization order and the execution order of subtransactions may still differ due to possible *indirect* (transitive) conflicts between global transactions caused by local transactions. In this case the problem may be solved by forcing each subtransaction to perform a write access to a special item, the *ticket*, thus forcing a direct conflict between any two subtransactions running at the same site [GRS94]. An especially interesting case, the one we actually consider in this paper, is when all LTMs are blocking either rigorous or strict (with ticket access forced by the GTM), and resolve directly local deadlock. In this situation the problem of ensuring the global serializability in the MDBS is reduced to detecting and resolving *global deadlocks*, i.e. deadlocks involving subtransactions submitted at several LDBSs, since in this case any schedule that is not globally serializable will result in a global deadlock.

Unfortunately the classical approaches proposed for homogeneous distributed database systems [Kna87] and, in general, for distributed systems [Sin89] cannot be extended to

MDBS, since the autonomy of local system precludes access to the information (i.e., item contention) needed to maintain a global state, e.g. a *Waits-For Graph (WFG)* of the entire multidatabase, and to detect possible cycles. Hence one has to rely on some weaker necessary condition which, not being a sufficient condition as well, will detect *potential* deadlocks that not necessarily correspond to *real* deadlock. Two approaches have been proposed in this direction up to now in the literature. The first one is to set a *global timeout* for every global transaction, and to detect a potential deadlock if the timeout expires. The second one is based on a directed graph called *Potential Conflict Graph (PCG)*, where each local system is considered as a single item. and a cycle corresponds to a potential deadlock [BLS90].

The first purpose of our study has been to analyze the performance of the proposed multidatabase deadlock detection methods in a variety of workload situations, and to compare them to an *ideal* method (not feasible in an MDBS) based on a WFG. A quite interesting preliminary result of the analysis has been to show that with PCG detection the distribution of the deadlock cycle length tends to be always very skewed, with most cycles being of length two (about 90% in typical situations). On the contrary with WFG detection the distribution tends to be uniform, especially with heavy workload. According to this remark we also propose a new *hybrid* approach that consists in directly detecting all the potential global deadlocks of length two, and detecting the remaining ones through a global timeout. This method, that we shall call *Hybrid Deadlock Detection (HDD)*, is actually considerably simpler and more suited for a distributed implementation than PCG and more stable than global timeout. Moreover the performance analysis we present in the paper clearly shows that HDD can be easily tuned to match the performance of PCG in a large variety of workload situations.

The paper is organized as follows. In Section 2 the reference MDBS architecture and the transaction model is defined. In Section 3 the different multidatabase deadlock detection methods proposed in the literature are discussed. Section 4 introduces the hybrid deadlock detection approach we propose, and also gives a distributed implementation of it. Next in Section 5 we present the simulation model used for the performance analysis, and we discuss the workload model and parameters. In Section 6 the results of the performance evaluation are presented and discussed, and, finally, conclusions are given in Section 7.

2 The Multidatabase Model

The MDBS architecture we consider in this paper is sketched in Figure 1. At each site S_i there is a *local agent* of the global transaction manager GTM_i, a set of active *servers* W_i^j and a LDBS which includes the local transactions manager LTM_i to handle local transactions. The LTM ensures local serializability by means of a blocking protocol (e.g. two-phase-locking (2PL)) producing rigorous deadlock-free schedules. Equivalently strict schedules can be produced with a write access to a ticket enforced for all subtransactions. In this environment, as pointed out in the Introduction, the task of the GTM to ensure global consistency is reduced to the detection and resolution of global deadlocks.

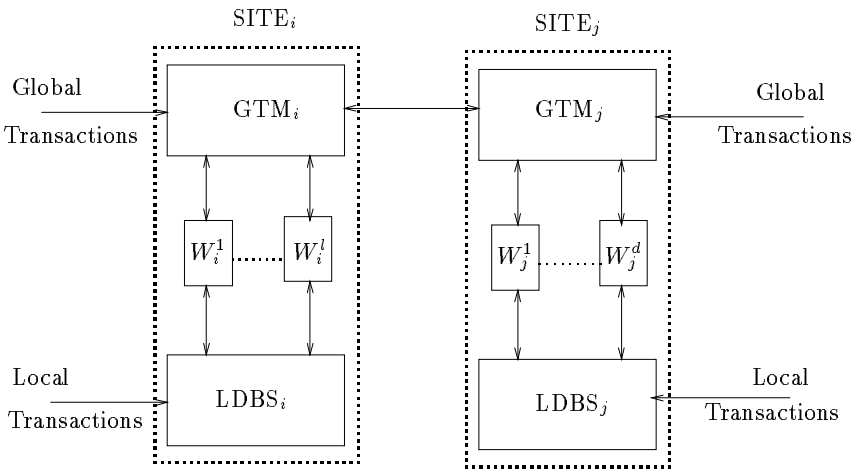

Figure 1: The multidatabase system architecture

Each transaction performs a sequence of read and write operations on a set of items. Local transactions only access items from a single site and are directly managed by the local LTM. Global transactions instead access items from several sites, and are managed by the GTM$_i$ agent of the site S_i where they are submitted, which acts as a *coordinator* [1]. For each site S_k where a global transaction T_j submitted in site S_i accesses an item, the GTM allocates a server W_k^j that will be actually performing all the operations the global transaction has to execute at site S_k. The set of all these operations is called a *subtransaction*, and its execution is seen by the LDBS as a local transaction. When a global transaction has performed all its accesses and all servers have completed their operations, the transaction enters the global commit phase, all subtransactions are committed and all servers deallocated. All servers of a given transaction are deallocated as well if the transaction is aborted because of a global deadlock detected by the GTM, or of a local deadlock that causes the abort of a subtransaction. We also assume for simplicity that transactions cannot abort for other reasons than deadlock. In what follows we consider a one-resource model (each transaction can have outstanding at most one request at a time).

3 Deadlock Detection Methods

3.1 Waits-For Graph

As discussed in the Introduction, the *Waits-For Graph (WFG)* is not a feasible approach for global deadlock detection in a MDBS, since it requires the complete knowledge of item

[1] The same GTM can be the coordinator of multiple concurrent global transactions.

contention, which is not globally available because of the LDBS autonomy. However, we may consider it as an ideal reference method since it allows to detect and resolve a global deadlock as soon as it is formed. Formally a WFG [Kna87, BHG87] is a directed graph where the vertices $\{T_i\}$ represent the active transactions and there is an edge from T_j to T_i if transaction T_j requests an item locked by transaction T_i in some site S_k. In this case we say that T_j is a *waiting* transaction. As we consider a multidatabase system, items are on distinct LDBSs, so the WFG is the union of the local WFGs one for each LDBS.

A WFG can be seen as a forest of trees. The root of each tree represents a non-waiting. i.e. *running*, transaction. All the other transactions in the tree are waiting directly or transitively on one of the items locked by the root transaction, so each path represents a distinct waiting chain. A cycle in the WFG represents a deadlock, in particular a local deadlock if all the items in the circular waiting chain belong to the same LDBS, otherwise a global deadlock.

The operations performed by a global transaction modify the structure of the WFG in the following way:

- if a running transaction T_i gets waiting on an item locked by T_j, then the tree A_i rooted in T_i becomes a subtree of the tree to which T_j belongs; this increases the average length of the paths in the WFG and reduces the number of trees;
- if a running transaction T_i is committed or aborted (because the request of an item forms a cycle in the WFG), then all items locked by T_i are released, and the tree A_i rooted in T_i is split. This creates possibly one running transaction and one distinct tree, with depth lesser than A_i, for each item previously locked by T_i. Altogether this modification decreases the average length of the paths in the WFG, and increases the number of the trees.

3.2 Global Timeout

The *Global Timeout (GT)* method simply associates a fixed timeout t_G to each global transaction. The transaction is assumed to be in a potential deadlock, and is then aborted as soon as the timeout expires. It is trivial to show that this method ensures global deadlock detection, since timeout expiration (before the transaction is committed) is an evident necessary condition to deadlock. From the viewpoint of the implementation, this method is inherently distributed, since all controls can be performed locally by the coordinating GTM_i agent and no information is needed about the state of the other transactions.

However the selection of an appropriate value for the timeout is very critical, and may considerably affect the performance. Small timeout values (compared with the transaction response time) lead to abort many transactions that are simply not finished, but actually not engaged in any deadlock, thus wasting system resources. On the other hand, large timeout values result in long detection delays of real deadlocks, and originate large transaction jams that severely affect the performance. Since a reliable estimate of transaction response time is usually not available, selecting the right timeout value may become a critical problem for the designer.

3.3 Potential Conflict Graph

A *Potential Conflict Graph (PCG)* [BLS90] is a directed acyclic graph where the vertices $\{T_i\}$ represents the global transactions, and there is an edge from T_j to T_i if there is a site S_k such that T_j is waiting on some item in S_k, and T_i detains at least a lock in S_k and is not waiting on any item in S_k. As for the WFG, the global PCG results from the union of the local PCGs one for each LDBS when considering the local PCG_k be the part of the PCG due to site S_k. In [BLS90] it has been formally proved that if there is a global deadlock in the waits-for graph, then there is a cycle in the PCG. The reverse is not true.

The main problem with PCG detection is that, especially if the number of items is large compared to the number of sites, and each transaction accesses many items, there is a large number of *apparent* deadlocks (i.e. potential deadlocks which do not correspond to actual ones). And this may affect the performance by causing many restarts. Therefore, in order to reduce such false alarms, when a global transaction gets waiting, the detection of cycles in the PCG is postponed by a time t_L, called *local timeout*. But setting up the local timeout is a delicate task and a wrong choice may affect the performance, though not so severely as in the case of global timeout.

Moreover the construction of the PCG requires to piece together all local PCGs. This gives problems both in a centralized and in a distributed implementation. Drawbacks of a centralized implementation are: the single point of failure implied by a centralized approach and the heavy communication to maintain the PCG up-to-date that risks to saturate links near the control site. In the case of a distributed implementation, typical deadlock detection methods proposed for distributed systems based on tokens [BHRS95], probes [MM84, CM82], consistent global snapshots [BT87, KS94] lead to an high complexity in terms of number of messages needed to detect potential deadlocks and, worse, do not take into account the restrictions connected to local autonomy [Vei90].

4 Hybrid Deadlock Detection

The main problem with PCG detection is that a cycle may have any length up to the number of sites in the MDBS. But, as we will discuss later in Section 6, a very interesting result of our performance analysis has been to show that, with PCG detection, the deadlock cycle length distribution is actually very skewed, with a large majority of cycles having length two. This has suggested us to propose a new method, that we shall call *Hybrid Deadlock Detection (HDD)* that consists in:

- directly detecting all potential global deadlocks that involve only two global transactions, i.e. corresponding to cycles of length two in the PCG;
- using a global timeout t_G to detect the remaining deadlocks (i.e. the one involving more than two global transactions).

The HDD approach has two main advantages. First building the PCG is no longer necessary, since only a partial knowledge of the PCG is needed to detect deadlocks involving

only two transactions; in particular, a cycle of length two can be detected just by piecing together a distinct pair of local PCGs. Therefore it is simpler than PCG and more suitable to a distributed implementation. Second, since the global timeout is used to detect only global deadlocks involving more than two transactions (i.e. a strict minority), the selection of the timeout value is far less critical than in the plain GT approach.

To show how the method works we give in the following subsection a sketch of a distributed implementation.

4.1 HDD Distributed Implementation

In the algorithm we assume that the GTM$_i$ agents communicate by exchanging messages over reliable, asynchronous and buffered channels and that transmission delays of messages are unpredictable but finite. There is neither a shared-memory nor a common clock.

For clarity of exposition and without loss of generality we assume that each GTM$_i$ can be the coordinator of one global transaction at a time. All the identifiers of global transactions are assumed to be unique. This can be achieved by using a timestamp technique [Lam78], e.g. utilizing as identifier of each global transaction the virtual time in which the transaction was submitted. All messages are supposed to be labeled with the identifier of the global transaction. Unread messages of committed/aborted transactions are removed from the buffer by a background garbage process. Receive commands allow to screen out messages in the buffer by specifying restrictions on the contents of the message [Geh84]. For example **receive** $deadlock\ (y) \wedge y = item$ ignores all messages in the buffer until one of type *deadlock* is received with its information component y equal to *item*.

Pseudo code sketching the behavior of GTM$_i$ is shown in Figure 3.

Information maintained by GTM$_i$. Each GTM$_i$ agent maintains the following information on the global transactions that detain a lock on some item or are waiting for some item at site S_i, and on each global transaction that was submitted at site S_i, and is therefore managed by GTM$_i$:

> $\mathcal{A}_i = \{T_1, .., T_n\}$: the set of all global transactions that are *active* at site S_i, i.e. lock at least one item at that site;
>
> $\mathcal{B}_i = \{T_1, .., T_m\}$: the set of all global transaction that are waiting on some item at site S_i;
>
> $\mathcal{S}(T_k) = \{S_1^k, .., S_q^k\}$: the set of all the sites S_j^k where the global transaction T_k, coordinated by GTM$_i$, is currently active.

For sake of simplicity in the following we omit the actions the GTM$_i$ has to do to atomically update the sets $\mathcal{A}_i$, $\mathcal{B}_i$ and $\mathcal{S}(T_k)$.

Actions of GTM$_i$. When a global transaction T_g is submitted to GTM$_i$, which will then be its coordinator, a timer is started, and if later the global timeout t_G is reached before the

transaction is committed, then the transaction is aborted. Each time T_g requests an item on a different site, say S_j, GTM$_i$ calls the procedure REQUEST (see Figure 3) which sends the *request* message to agent GTM$_j$ (line 1). If the reply is the message *lock* (line 27), that means the item was locked, and then the transaction may proceed (line 4). If the reply is the message *abort* (line 29) that means that the subtransaction at site S_j was aborted due to a local deadlock , and hence T_g is aborted as well (line 6). Finally if the reply is the message *wait* then a *local timeout* t_L is set (line 11), that has the same function than in PCG. If t_L expires before a *lock* message arrives (line 9) then the procedure to detect a potential deadlock cycle of length is started (line 13).

Detection of potential deadlock cycles of length two. According to the definition of PCG, and the one given above for $\mathcal{A}_i$, $\mathcal{B}_i$ and $\mathcal{S}(T_k)$, the condition for a potential global deadlock of length two can be reformulated as follows: a transaction T_g blocked in site S_j is setting a potential deadlock of length two, if there exists another transaction T_x such that T_x is blocked on some site where T_g is active, and active on site S_j (as shown in Figure 2). So the procedure works as follows: when T_g coordinated by GTM$_i$ requests an item at site S_j and the item is locked, then the set $\mathcal{A}_j$ is piggibacked on the reply message *wait* sent by S_j (line 30). Then GTM$_i$ checks the predicate $(T_g \in \mathcal{A}_i) \wedge (\mathcal{B}_i \cap \mathcal{A}_j \neq \emptyset)$, and if it is true a potential deadlock involving S_i and S_j is detected (lines 13-14). Otherwise there could still be a deadlock involving S_j and a site $S_k \in \mathcal{S}(T_g) - \{i\}$, therefore a message *check-deadlock* with $\mathcal{A}_j$ piggibacking is sent to every GTM$_k$ with $S_k \in \mathcal{S}(T_g) - \{i\}$ (line 16). Each GTM$_k$ then checks the predicate $(\mathcal{B}_k \cap \mathcal{A}_j \neq \emptyset)$, and if it is true sends a *deadlock* message to GTM$_i$ (line 24). On the other hand, if while waiting a *deadlock* message, the *lock* message is received, the transaction may proceed (lines 18-19). For simplicity, in the algorithm the deadlock resolution technique employed is the following: the transaction that closes the cycle is aborted. More efficient deadlock resolution techniques could be envisaged that, for example, abort the youngest transaction involved in the cycle by using its identifier (timestamp).

Messages exchanged per waiting state and detection delay. The main advantage of this distributed algorithm is that to maintain the structures $\mathcal{A}_i$, $\mathcal{B}_i$ and $\mathcal{S}(T_k)$ each GTM$_i$ only needs information which is locally available, and hence no additional messages are needed. Messages specifically sent to check for deadlock are sent only when a transaction enters a waiting state, i.e. gets blocked (and the local timeout expires). So the detection cost is in some way proportional to the degree of congestion, and to the probability of deadlock. In the best case, when the site coordinating the global transaction is one of the two sites involved in the potential deadlock, the number of messages exchanged per waiting state is zero. In the worst case, it is $2(n-2)$ where n is the number of sites in which the global transaction is active, and assuming that a potential deadlock is detected in each site. As for the detection delay in terms of number sequential messages exchanged, the algorithms detects immediately a potential deadlock in the best case, and in $2T$ in the worst case, with T being the average transmission delay.

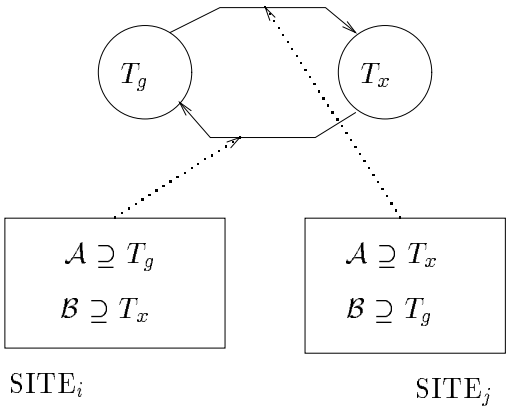

Figure 2: Potential deadlock of length two

5 The Simulation Model

In this and in the following section we present the results of a modeling study that we have performed to compare the different global deadlock detection methods for multidatabase systems discussed in the previous sections, i.e. the GT, PCG, and the new HDD we propose in this paper. Moreover we also compare the performance of the above methods to the *ideal* WFG method that we take as a reference since it allows instant detection and recovery from deadlocks, and detects only real deadlocks.

In our simulation model the MDBS is represented as a set of N_{site} identical LDBSs, each managing the same number N_{item} of items, and with the same transaction workload. More precisely we have assumed a closed transaction workload, i.e. there are at each site two fixed populations of N_G and N_L customers submitting respectively local and global transactions. Each customer continuously generates transactions, waiting a *think time* t_{think}, between the commit of a transaction and the submission of the next one. A closed workload model has indeed clear advantages in a performance study, since the number of active transactions is always limited by the number of customers, and therefore the system never gets to saturation. The intensity of the workload can then be readily modulated by varying the number of customers and/or the think time.

As for the transaction model, we have assumed that each global transaction accesses a total of n^g_{lock} items, from a subset of the sites in the multidatabase, and that the execution of the transaction evolves as follows:

- a *server* is started on each site where the transaction requests a lock;
- locks are requested sequentially, by submitting each request to the corresponding server, only after the previous lock has been granted;

```
   procedure REQUEST(item,GTM_j): % item to lock, GTM_j in which item is stored %
   begin
1.      send request(item) to GTM_j;
2.      select
3.            receive lock
4.                  return(item_locked);
5.            receive abort
6.                  return(abort); % local deadlock %
7.            receive wait(A_j)
8.                  select
9.                        receive lock
10.                             return(item_locked);
11.                       delay t_L;
12.                 end
13.                 if (T_g ∈ A_i) ∧ (B_i ∩ A_j ≠ ∅) ∧ (i ≠ j); % deadlock detection procedure %
14.                 then return(abort); % potential global deadlock %
15.                 else
16.                       for each w ∈ S(T_g) − {j,i} send check_deadlock(item,A_j) to GTM_w;
17.                       select
18.                             receive lock % false global deadlock %
19.                                   return(item_locked);
20.                             receive deadlock(y) ∧ y = item
21.                                   return(abort); % potential global deadlock %
22.                       end
23.     end
   end.

   when check_deadlock(item,A_j) arrives at GTM_i from GTM_k:
   begin
24.     if (B_i ∩ A_j ≠ ∅) then send deadlock(item) to GTM_k;
   end.

   when request(item) arrives at GTM_i from GTM_k:
   begin
25.     forward the request to the server of the transaction whose coordinator is GTM_k;
26.     if the item has been locked;
27.     then send lock to GTM_k;
28.     else if the transaction has been aborted by the LDBS;
29.          then send abort to GTM_k;
30.          else send wait(A_i) to GTM_k;
   end.
```

Figure 3: The HDD algorithm code (GTM_i side)

- each lock is a write lock with probability p_w;
- when a lock is acquired, a time $t^g_{I/O}$ is spent for disk access (on the server site) and transmission, and a time t_{CPU} for processing (on the transaction site);
- if either a local or global (potential) deadlock is detected the global transaction is aborted and restarted after a time t_{rest};
- if the transaction completes successfully a processing time of t^g_{comm} is spent on the transaction site to complete the global commit phase.

Local transactions have a similar structure, but request their n^l_{lock} locks directly from their LTM, and spend all their disk and processing time on their site.

In designing the simulator we adopted a fairly simple structure, since the main purpose was not to provide a *realistic* representation of the MDBS, but instead to investigate on deadlock and to compare different detection methods. Therefore the resources in the sites were very schematically represented: all times were assumed to have exponential distribution, and resource contention was explicitly modeled only for CPUs by assuming Processor Sharing discipline. Disk and transmission times were all modeled as pure exponential delays.

6 Results of the experiments

A preliminary set of experiments was run to select appropriate values (intervals) for the workload parameters to be used in the comparative analysis. A main outcome of this preliminary analysis was that considering local transaction workload was not necessary, since it merely produces a *background noise*, and is not interesting for our specific purpose of performance comparison. The values of workload parameters used in the main set of experiments discussed below are shown in Table 1. The number of customers submitting global transactions per site N_G has been used as a running parameter in some experiments to represent the workload intensity. In the other experiments three values were considered to represent different intensities: N_G=6 for *light workload*, N_G=8 for *medium workload* and N_G=10 for *heavy workload*. As stated before, a symmetrical structure of the MDBS was considered, in the sense that the same values of the parameters were used for all the sites in all the experiments.

As for the performance metrics we considered two main indices:

- T_R: *transaction response time*, i.e. expected time between submission and commit of a global transaction.
- λ: *throughput*, i.e. expected number of global transactions committed at each site per unit of time.

The two indices represent respectively a measure of the *quality* of the service and the *productivity* of the system.

Parameter	*Meaning*	*Value*
N_{site}	Number of sites	10
N_{item}	Number of items per site	200
N_G	Number of global customers per site	6 - 10
n^g_{lock}	Number of items locked by a transaction	15
p_w	Probability of a write lock	50
t_θ	Think time	10 sec
t^g_{CPU}	Processing time per item	35 ms
$t^g_{I/O}$	Disk and transmission time per item	40 ms
t^g_{comm}	Global commit time	100 ms
t^g_{rest}	Global restart time	1 sec

Table 1: Workload Parameters

6.1 Distribution of deadlock cycle length

A first series of interesting results concerns the distribution of the length l of deadlock cycles, i.e. the number of items involved in a deadlock. This was analyzed for different workload intensities and for different deadlock detection methods. The distributions are shown in Figure 4 and 5 for the WFG and PCG methods respectively.

The behavior is quite different in the two cases. With WFG detection cycles tend to be short for light workload (low deadlock probability), but as the intensity (and the deadlock probability) increases the distribution tends to become uniform. Instead with PCG detection the distribution appears to be largely insensitive to the workload intensity, and is very skewed, with high probability for cycles of length two (about 90% for the workload profile we considered). This suggests indeed that deadlock detection can be efficiently performed, concentrating on potential cycles of length two, as actually is done by the HDD method we propose.

This behavior can be explained as follows. Referring to the discussion in Section 3.1, in the WFG the cycle length depends on the depth of the trees that form the graph. This in turn depends on two different factors: the average utilization of the items which tends to increase the depth of trees, and the deadlock probability which tends to prune the graph since it leads to transaction abortions, and then to tree decomposition. For low item utilizations (light workload) few transactions are waiting, and then the trees in the WFG are shallow, and this is why the distribution is skewed on the left. When the workload intensity increases the experiments showed that the item utilization increases much faster than the deadlock probability and hence becomes the dominant factor, leading to deeper trees and to a flatter distribution. In the case of light workload, an explanation to the predominance of short cycles has been given in [BHG87] while an empirical confirm can be found in [Gray et al. 81].

In the PCG case the generation of the paths in the graph has a substantially similar structure, but in this case the potential deadlock probability is the dominant factor, since

Figure 5: Deadlock cycle length distribution with PCG detection

Figure 6: Global transaction throughput versus workload intensity

for the same workload it is considerably higher than in the WFG case, as in the PCG the contention is on the sites (a few units) and not on the items (several thousands). Increasing the workload intensity does not change the behavior, since the very high potential deadlock probability limits the number of active transactions (not in think or restart state), and therefore sets a limit to the item utilization as well.

6.2 Response time and throughput

Results on throughput, shown in Figure 6, are in full agreement with the remarks we made at the end of last section. That means increasing the workload intensity N_G increases the number of active transactions and then item utilization and the deadlock probability, and finally the transaction response time. This brings the system toward saturation, but however it can never get to it since we are considering a closed workload. Nevertheless, when N_G goes above a given threshold the throughput starts decreasing, with a behavior not surprisingly similar to thrashing in paging systems.

The interesting remark is that both the PCG and the HDD methods show a more stable behavior and get their maximum throughput for a higher workload intensity. This can be readily explained by considering that the detection of potential deadlocks instead of real ones leads to a larger number of restarts, and consequently for heavy load sets some kind of control on the number of active transactions. It should be remarked that, as far as throughput is concerned, both PCG and HDD perform better than the reference method.

For transaction response time we have run a set of experiments to compare in different situations all four detection methods i.e. WFG, PCG, HDD and Global Timeout (GT). The

Figure 7: Transaction response time and local timeout (medium workload)

problem has been mainly to determine the influence of two relevant parameters that appear in the definition of the algorithms i.e. the local timeout t_L and the global timeout t_G (cfr. Sect. 3 and 4).

Figures 7 and 8 show the influence of local timeout for medium and high workload intensity. The pictures clearly show that small values of t_L give the best performance. Therefore, since the minimum is hard to determine for the system designer, a choice of $t_L = 0$ is always safe, and indeed we have assumed a null value for t_L in all the other experiments. Another interesting remark is that a *reasonably* low value of the global timeout in the HDD method reduces the negative effect of high values of t_L. This can be clearly seen in Figure 8 by comparing graphs a and b.

The main results of the performance study are summarized in Figures 9, 10 and 11, where the transaction response time T_R is plotted, for all detection methods, against the value of the global timeout t_G, for different workload intensities.

The pictures clearly show the main problem connected with GT detection, i.e. selecting an appropriate value for t_G. In fact the performance of the GT method dramatically depends on this parameter. Small values of t_G cause too many restarts, due to apparent deadlocks, and large values increase congestion since deadlocked transactions get stuck for a long time before being restarted. Therefore the selection of values for t_G either too small or too large may lead to a considerable degradation in the response time. So there is no safe side to be taken by the designer, and this makes the GT method definitely very critical to use, though quite easy to implement.

Both WFG and PCG are of course independent from t_G, and for light and medium workloads have a very similar performance. Instead the performance of HDD is somewhat in

Figure 9: Transaction response time and global timeout (light workload)

Figure 10: Transaction response time and global timeout (medium workload)

between PCG and GT: for small values of t_G it has almost the same behavior than GT, but for larger values of t_G it behaves pretty much as PCG. That means HDD is still sensitive to the value of t_G but, differently from the GT case, there is a *safe side* since it is sufficient to select t_G large enough. As a rule of thumb from our experiments it was determined that t_G can be safely selected in a range between two and six times the expected transaction response time.

As for the influence of the workload intensity, the remarks made when discussing the throughput are confirmed. That is for light and medium workload (Figures 9 and 10) PCG and HDD (with appropriate value of t_G) have pretty much the same performance than WFG, but for heavy workload (Figure 11) the former performs considerably better. As for GT, with heavy workload this method can only approach the performance of PCG and HDD, and only for the optimum value of t_G, which, as we pointed out earlier, may be very hard to select.

7 Conclusions

The aim of our study has been to gain a better understanding of the structure of the global deadlock detection problem in multidatabase systems, and to compare, in a large variety of workload situations, the performance of the global deadlock detection algorithms proposed in the literature. A first relevant result of the study has been to show that if potential deadlocks are detected, and regardless to the workload intensity, the distribution of the deadlock cycle length becomes very skewed, with a very large majority of the cycles involving only two sites. Therefore we propose a new approach, the *Hybrid Deadlock Detection (HDD)* that concentrates on detecting such minimal cycles. This method is very suitable for a distributed

Figure 11: Transaction response time and global timeout (heavy workload)

implementation, and actually a sketch of a distributed algorithm is presented in the paper. Moreover, according to the results of our performance analysis, HDD compares quite well with the other methods we considered, i.e. *Global Timeout (GT)*, *Potential Conflict Graph (PCG)*, and *Waits-For Graph*, the latter being only an *ideal* reference method, since it cannot be implemented in a MDBS.

The results of the performance analysis can be summarized as follows:

- the WFG method performs well especially for light workload, but as the workload intensity increases it is outperformed by the methods based on weaker conditions; this is because their higher rate of restart (due to *apparent* deadlocks) limits the number of active transactions, thus postponing the *thrashing* effect due to saturation, as our analysis of the throughput clearly shows; this behavior may seem surprising, as one would expect that causing *inappropriate* restarts should degrade the performance;

- the GT method can be very easily implemented, but may be impractical to use since the selection of the global timeout t_G value is very critical, with an optimal value that, for heavy workload, corresponds to a very deep minimum; values both smaller and larger than the optimal value produce a severe performance degradation, therefore there is no *safe* side to be taken for the designer;

- the PCG method has quite a good performance, and an appropriate selection of the local timeout t_L is not critical and allows to limit the number of restarts due to apparent deadlocks; on the other hand it is quite complex to implement since it requires to build a global state of the multidatabase, in order to trace cycles of any length;

- the HDD method removes most of the implementation problems of PCG, because only simpler structures and a limited number of messages are needed to trace cycles of length two; moreover the selection of t_G is far less critical than in GT since there is a *safe* side; and for the performance our analysis shows that, for a reasonable choice of t_G, HDD has practically the same performance than PCG on a wide range of workload conditions.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publ. Co., Reading, Massachussetts, 1987.

[BT87] G. Bracha, and S. Toueg. Distributed deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[BGRS91] Y. Breitbart, D. Georgakopolous, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991.

[BHS92] Y. Breitbart, Garcia-Molina H., and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–240, 1992.

[BLS90] Y. Breitbart, W. Litwin, and A. Silberschatz. Deadlock problems in a multidatabase environment. In *Proc. of the IEEE COMPCON spring 91, San Francisco, USA*, IEEE press, 145–151, 1991.

[BS88] Y. Breitbart and A. Silberschatz. Multidatabase update issues. In *Proc. of the ACM SIGMOD International Conf. on Management of Data, Chicago, USA*, ACM press, 135–142, 1988.

[BST87] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. *IEEE Data Engineering Bulletin*, 10(3), 1987.

[BHRS95] J. Brezinski, J.M. Helary, M. Raynal and M. Singhal. Deadlock models and general algorithms for distributed deadlock detection. *To appear on Journal of Parallel and Distributed Computing.*

[CM82] K.M. Chandy, and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. of the ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, ACM press, 290–294, 1894.

[DE89] W. Du and A. Elmagarmid. Quasi serializability: A correctness criterion for global concurrency control in interbase*. In *Proc. of the 14th International Conference on Very Large Data Bases, Amsterdam, The Netherlands*, 347–355, 1989.

[Geh84] N.H. Gehani. Broadcast Sequential Processes. *IEEE Transactions on Software Engineering*, SE-10(4):343–351, 1984.

[Gray et al. 81] J.N. Gray, P. Homan, H.F. Korth and R.L. Obermark. A strew man analysis of the probability of waiting and deadlock in a database system. *Tech, Rep. RJ 3066*, IBM Research Laboratory, San Jose, 1981.

[GRS94] D. Georgakopoulos, M. Rusinkiewicz, and A. Shet. Using tickets to enforce the serializability of multidatabase transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):166–180, 1994.

[Kna87] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–327, 1987.

[KS94] A.D. Kshemkalyani, and M. Singhal. Efficient detection and resolution of generalized distributed deadlocks. *IEEE Transactions on Software Engineering*, 20(1):43–54, 1994.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):331–360, 1978.

[MRKS91] S. Mehrotra, R. Rastogi, H.F. Korth, and A. Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *First International Conference on Parallel and Distributed Information Systems*, 245–252, 1991.

[MRKS92] S. Mehrotra, R. Rastogi, H.F. Korth, and A. Silberschatz. Relaxing serializability in multidatabase systems. In *Proc. of the 2nd International Workshop on Research Issues on Data Engineering: Transaction and Query Processing, Tempe, USA*, 205–212, 1992.

[MM84] D.P. Mitchel, and M.J. Merrit. A distributed algorithm for deadlock detection and resolution. In *Proc. of the ACM Symposium on Principles of Distributed Computing, Montreal, Canada*, ACM press, 290–294, 1894.

[Sin89] M. Singhal. Deadlock detection in distributed systems. *IEEE Computers*, 37–47, 1989.

[Vei90] J. Veijalainen. *Transaction Concepts in Autonomous Database Environments.* R. Oldenbourg Verlag, Germany, 1990.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399