A Distributed Transaction Management Scheme for Multidatabase Systems*

Xinfeng Ye, John A. Keane,
Department of Computer Science, Department of Computation,
University of Auckland, UMIST,
New Zealand. Manchester, UK.

Abstract
Transaction management in a multidatabase system must ensure global serializability. Local serializable execution is, by itself, not sufficient to ensure global serializability, since local serialisation orders of subtransactions of global transactions must be the same at all systems. In this paper a distributed transaction management scheme is introduced. The scheme maintains autonomy of the local database systems. It is free from global deadlock, and guarantees fairness in the execution of the transactions in the system.

1 Introduction
A multidatabase system (MDBS) is a collection of pre-existing autonomous, and possibly heterogeneous, local database systems (LDBSs). Transactions in an MDBS are of two types:
- Local transactions: Those transactions that only access data managed by a single LDBS.
- Global transactions: Those transactions that access data managed by more than one LDBS.

Transaction management in the MDBS is hierarchical. Each LDBS controls the local transactions and the subtransactions of the global transactions at its site, and assures serializable execution at that site. The MDBS software controls the global transactions, and assures global serializability.

Global serializability guarantees the correct concurrent execution of global transactions [4]. Global serializability requires that the local serialisation orders of the subtransactions of global transactions be the same at all LDBSs where the global transactions execute. It is difficult to achieve this because the LDBSs do not release information concerning their local serialisation orders.

Transaction management in the MDBS also has to deal with the possibility of global deadlocks. Deadlock detection in an MDBS is difficult and complicated [1]. Therefore, it is highly desirable to incorporate methods assuring freedom from global deadlocks into a transaction management scheme [1].

Most schemes that ensure global serializability in an MDBS use a centralised global transaction manager (GTM) to coordinate execution of the global transactions [3, 5, 8, 10]. A centralised scheme is prone to bottleneck problem in distributed systems [9]. In [1] a distributed scheme is introduced. However, the scheme is not fair in the sense that a global transaction can be continuously aborted if it conflicts with some other transactions. Therefore, a global transaction may never have a chance to be executed.

In this paper a distributed transaction management scheme is introduced. The scheme is free from global deadlock and does not require knowledge of the transaction management methods of the LDBSs. In the scheme each LDBS has a GTM. The GTM on each site commits or aborts the global transactions based on its local information. The scheme is more flexible than the one in [1], and guarantees fairness in the execution of the transactions.

2 Transaction Management in MDBS
The MDBS software is a layer above the LDBSs. It coordinates the global transactions. Local transactions are submitted to the LDBSs directly. Therefore, the MDBS software is unaware of the existence of the local transactions. Global transactions are submitted to the MDBS. They are decomposed into a set of subtransactions. Each subtransaction is executed at one LDBS, and is treated as if it were a local transaction. A global transaction commits when all the subtransactions of the global transaction have completed their operations successfully. If any of the subtransactions aborts, then the global transaction aborts. Therefore, the operation of a subtransaction can only be committed when the global transaction commits. In the following a transaction refers to either a local transaction or a subtransaction.

Transactions can be modelled as a sequence of read and write operations, denoted as r(x) and w(x) respectively, where x is the data item to be accessed. A local execution consists of all the operations performed at an LDBS. Two operations conflict if both operate on the same data item and one of them is a write. Two transactions conflict directly if they contain conflicting operations. For two transactions, T1 and T2 that conflict directly, T1 precedes T2 (denoted as T1 → T2) in the execution order if T1 is executed before T2 in terms of the conflicting operations. The precedence relation is transitive. A local execution is serializable (also referred to as local serializability) if all the conflicting transactions
have the same precedence order on all the conflicting operations. The serialisation order refers to the precedence relation between the transactions in a serializable execution.

Let s₁ and s₂ be two subtransactions at an LDBS. s₁ and s₂ indirectly conflict if they do not directly conflict and there are some transactions, t₁, ..., tₙ, such that s₁ → t₁ → ... → tₙ → s₂. Two transactions are said to be conflict with each other if they conflict either directly or indirectly.

Example 1: Assume an MDBS consists of two sites, 1 and 2. Data items a and b are stored at site 1 and data items c and d are stored at site 2. All local transactions are waiting for each other, and none of transactions can make progress. A global deadlock involves more than one LDBS, and the MDBS is unaware of the local transactions. In order to solve the problem, each LDBS has some mechanisms to prevent and resolve local deadlock. This, however, does not ensure that global deadlock will not occur.

Example 2: Consider an MDBS where data items a and b are at site i and c and d are at site j. The following global transactions are submitted:

Let L₃ and L₄ be two local transactions that are submitted at sites i and j respectively:

L₃ : r₃(a)r₃(b) r₃(c) L₄ : r₄(a)r₄(b)

Let e₁ and e₂ be the executions at sites i and j respectively:

e₁ = r₃(a), w₁(a), w₂(a), r₃(b) e₂ = r₄(c), w₃(c), w₄(d), r₄(d)

A global deadlock may occur as follows: At site i, G₁₃ waits for data item a locked by L₃. L₃ waits for the data item b locked by G₂₄. Therefore, G₁₃ is waiting indirectly for G₂₄ at site i. Similarly, G₂₄ is waiting indirectly for G₁₃ at site j. G₁₃ waits for data item a locked by L₃. L₃ waits for data item b locked by G₂₄. Thus, G₁₃ and G₂₄ are indirectly conflict at site 2; and the order is L₃ → G₁₃ → G₂₄ → L₄. Therefore, G₁₃ and G₂₄ indirectly conflict at site 2; and the order is G₁₃ → G₂₄.

In Example 1, although at both sites the executions are locally serializable, the executions are not globally serializable. This is because the serialisation order between the subtransactions of G₁ and G₂ are different at the two sites. The problem is caused by the execution at site 2, where the order of performing the operations is different from the serialisation order between G₁₂ and G₂₂. The difference is due to L₃ which introduces indirect conflict between G₁₂ and G₂₂ and forces G₂₂ to be serialised preceding G₁₂.

The problem described above cannot be solved by limiting the concurrent execution of the global transactions. This is because the MDBS is unaware of the existence of local transactions, thus it cannot detect indirect conflicts between the subtransactions that are caused by the local transactions. In order to solve the problem, [6] proposed a scheme which forces the subtransactions to conflict directly. In the scheme an artificial data item is inserted to each LDBS. This data item is maintained by the LDBS as a regular data item. Each subtransaction is extended to include a write access to that data item at the corresponding LDBS. This access forces the subtransactions to conflict directly.

If the LDBSs use some sort of locking mechanism, then global deadlock could arise. Global deadlock refers to the situation that more than one global transaction are waiting for each other, and none of them can make progress. A global deadlock involves more than one LDBS.

Each LDBS has some mechanisms to prevent and resolve local deadlock. This, however, does not ensure that global deadlock will not occur.

3 The MDBS Model

The completely distributed system architecture in [1] is used here. The MDBS software at each site consists of a GTM and a set of servers. The GTMs at different sites can communicate with each other. Global transactions can be submitted to any of the GTMs. The GTM to which a global transaction is submitted becomes the coordinator for this transaction. Each global transaction is assigned a timestamp. The timestamps are unique system-wide, and define a total order among the global transactions. The timestamps generated at each site are in monotonically increasing order. The timestamp of a transaction G is denoted as time(G).

The coordinator of a global transaction decomposes the transaction into a set of subtransactions, each of which is sent to a participating site where it can be executed. All subtransactions of the global transaction carry its timestamp. The coordinator controls the commit or the abort of the global transaction. If a global transaction is aborted due to the violation of global serializability, it will be restarted with the same timestamp.

When a GTM receives a subtransaction, it creates a server. A server is a process assigned to a subtransaction by a GTM to act as an agent for the global transaction. A server submits the operations of the subtransaction to the LDBS, monitors its execution and interacts with the local GTM. If a server and the coordinator of the global transaction reside at different sites, then the local GTM of the server controls communication between the server and the coordinator. In this paper, server and subtransaction are used interchangeably. The coordinator of a global transaction is also called as the coordinator of the subtransactions (servers) of the global transaction.

Interaction between the servers and the LDBSs occurs (a) when the servers send the operations of the subtransactions to the LDBSs and (b) when the LDBS reports the completion of the operations. A server can instruct a LDBS to commit or abort the operations. The following assumptions are made:

1. Each LDBS guarantees local serializability.
2. All local executions are deadlock-free. That is, either deadlock does not occur in the LDBSs, or if deadlock occurs then it is detected and resolved locally.
3. All local executions are fair. That is, the opera-
tions submitted to a LDBS will be completed
eventually.
4. Once an LDBS informs a server that the opera-
tions of the subtransaction have been completed,
the LDBS will not abort the operations unilater-
ally.
5. The LDBSs do not inform the MDBS software of
the local serialisation orders.

4 The Scheme
In a similar way to [6], the scheme in this paper
enforces the direct conflict between the subtrans-
actions on a LDBS. In contrast to [6], the global trans-
actions are not subject to global deadlock.
To enforce the direct conflict, each LDBS main-
tains a data item dummy. A write to dummy is
added to each subtransaction. This forces the sub-
transactions on a LDBS to conflict directly. The
conflict makes it possible for the GTM to know the
serialisation order of the subtransactions at a site.
The main ideas of the scheme are as follows:
1. When the operations of a subtransaction are com-
pleted, the coordinator is informed. When all the
subtransactions have completed successfully, the
coordinator commits the global transaction. If one of
the subtransactions aborts at a site, then the coor-
dinator aborts all other subtransactions.
2. Each server must request the LDBS perform a
write to dummy first. After the LDBS completes
this write, the server submits the operations of the
subtransaction to the LDBS.
3. Only one server is allowed to submit w(dummy) to
an LDBS at any time. The order of committing the
operations of the subtransaction must be the same
as the order of submitting w(dummy) to the LDBS.
4. When the coordinator of global transaction G is
informed that one of its subtransactions has com-
pleted, the coordinator requests the other subtrans-
actions of G be executed immediately. When the
request is received by the GTM at a site, the GTM
aborts all the subtransactions which precede G and
whose timestamps are greater than time(G). Abort-
ing transactions prevents global deadlock.
The details of the scheme are explained now.
Servers are in one of several states. A GTM creates
a server when the GTM receives a subtransaction.
The created server is in the dormant state. A GTM
activates a server by changing its state to register
when no other servers are in the register state.
When a server enters the register state, it sub-
mits w(dummy) to the LDBS. When the operation is
completed by the LDBS, the server informs the
GTM and enters the active state. A server in the
active state submits the operations of the subtrans-
action to the LDBS. When the operations of a sub-
transaction are completed by the LDBS, the server
sends a request-to-commit message to its coordina-
tor and enters the prepared-to-commit state. The
request-to-commit message informs the coordinator
that the subtransaction has completed. When the
servers of a global transaction have all entered the
prepared-to-commit state, the coordinator instructs
the servers to enter the ready-to-commit state.
A server sends a ready message to the coordinator
when it enters the ready-to-commit state. A server in
the ready-to-commit state can only be aborted by the
coordinator of the server. The ready-to-commit state
is necessary because some of the servers may have
been aborted due to the local deadlock prevention
measures after they entered the prepared-to-commit
state. Therefore, the ready-to-commit state allows
the coordinator to test whether all the servers are
still available to commit. If the coordinator receives
ready message from all its servers, then the coordina-
tor asks the servers to commit. If a server is aborted
before it enters the ready-to-commit state, then the
coordinator of the server will receive an abort mes-
sage. In this case the coordinator will abort all the
subtransactions.
At any state, a server aborts itself if it receives an
abort message from the GTM. If a server is aborted
to prevent local deadlock, then an abort message is
sent to the coordinator of the server. When a coor-
dinator receives an abort message, the coordinator
asks all its servers to abort.
The operations of the coordinators and the servers
are described below in an event driven style.

Coordinator of a Global Transaction:
1. when received a global transaction:
2. assign timestamp to the transaction;
3. decompose the transaction;
4. send each subtransaction to corresponding
   LDBS;
5. when received abort message:
6. send abort message to all LDBSs with
   subtransactions;
7. when received first request-to-commit message:
8. send immediate-execution request to all
   LDBSs except one from which request-to-
   commit message is received;
9. when request-to-commit messages received from
   all servers:
10. send continue message to all LDBSs where
    servers reside;
11. when ready received from all servers:
12. send commit message to all LDBSs where
    servers reside;
13. when change to register state:
14. submit w(dummy) to the LDBS;
15. when w(dummy) is completed by the LDBS:
16. send finish-register message to local GTM;
17. submit operations of subtransaction to the
    LDBS;
18. when submitted operations are completed by
    the LDBS:
19. send request-to-commit message to local
    GTM;
20. when continue message is received:
21. send ready message to local GTM;
22. when commit message received:
23. request the LDBS to commit the operations;
24. terminate;
25. when abort message received:
26. request the LDBS to abort the operations;
27. terminate;
A GTM has two tasks. One to act as coordinator
of global transactions being submitted to the GTM.
The other to create and control the servers at a site. To guarantee global serializability, a GTM has to know the serialisation order of the subtransactions. In this paper a GTM only activates a server when no servers are in the register state (i.e. when no w(dummy) is waiting to be completed). Thus, only one server can submit w(dummy) to the LDBS. As a result, dummy is accessed in the same order as the subtransactions are activated; and the order is the serialisation order of the subtransactions.

In order to achieve global serializability, a GTM only allows a subtransaction a to send a request-to-commit message to a’s coordinator when all subtransactions which precede a have committed or aborted. This measure ensures that, if the subtransactions of some global transactions have different local serialisation orders, then the global transactions will not be committed.

Global deadlock must be prevented. In the scheme, a transaction with a larger timestamp is aborted if the transaction might be blocking a transaction with a smaller timestamp to commit. Aborting a transaction with a larger timestamp ensures that all the transactions have a chance to be executed eventually.

**Global Transaction Manager:**
26. when subtransaction st is received:
27. create a server for st;
28. if there are no servers in the register state
   then change st to register state;
29. else set st to dormant;
30. when received finish-register message:
31. activate a server which is dormant;
32. when received request-to-commit message from server st:
33. if all subtransactions which precede st have committed or aborted
   then forward the message to coordinator of st;
34. else hold the message;
35. when received ready-message from a server:
36. forward message to the coordinator of server;
37. when received one of continue/commit/abort from coordinator of server:
38. forward the message to the server;
39. when received immediate-execution from coordinator of st:
40. for each server which precedes st and whose timestamp is larger than time(st) do
   if server is not ready-to-commit
   then send abort message to server and the coordinator of the server respectively;
41. if st is dormant then activate st;
42. when server st has committed or aborted:
43. check if received request-to-commit from server st which follows st in the serialisation order;
44. if such a message has been received
   then forward message to coordinator of st;

Although each GTM forces the subtransactions to be committed in their serialisation order, the scheme allows the concurrent execution of the subtransactions of different global transactions at a site. The concurrent execution of the subtransactions is only restricted when they access the dummy data item in the LDBS. This restriction allows the GTM to know the serialisation order of the subtransactions without violating the autonomy of the LDBS.

All subtransactions can be ordered according to the precedence relation. However, since some of the precedence relations between the transactions do not affect the operations of the transactions [2]. More than one server may be in the prepared-to-commit state if the operations of the servers are not influenced by each other. Hence, sometimes it is necessary for the GTM to delay the request-to-commit message (lines 33-34).

5 Correctness of the Scheme
Correctness is proved by showing that the scheme satisfies three properties: (a) global serializability, (b) deadlock-free, and (c) fairness.

**Theorem 1:** Global serializability is guaranteed.

**Proof:** Assume global serializability is not satisfied. This means that the subtransactions of some committed global transactions have different local serialisation orders at some LDBSs. Without lose of generality, the following assumptions can be made:
1. Two global transactions G1 and G2 such that time(G1) < time(G2), have committed.
2. The subtransactions of G1, G1.1, and G1.2, are executed at sites 1 and 2 respectively.
3. The subtransactions of G2, G2.1, and G2.2, are executed at sites 1 and 2 respectively.
4. The serialisation orders at sites 1 and 2 are G1.1 → G2.1 and G2.2 → G1.2 respectively.

Since G1.1 → G2.1 holds at site 1, G2.1 cannot send a request-to-commit message before the operations of G1.1 are committed (lines 33-34). The coordinator of a global transaction tries to commit the transaction when it receives the request-to-commit message from all its subtransactions. This means that G2 cannot commit until G1 commits. Since G2 cannot send a request-to-commit message to its coordinator before G1 commits, no subtransactions of G2 can be in ready-to-commit state before G1 commits (lines 9-10 and 20-21). Applying the same argument at site 2, G1 cannot commit until G2 commits. From lines 18-19, when the operations of G1.1 are complete at site 1, G1.1 sends a request-to-commit message to its coordinator. In turn, the coordinator will send an immediate-execution request to site 2 (lines 7-8). Since time(G1) < time(G2), G2.1 is aborted at site 2 (lines 39-40). Therefore, G2.2 → G1.2 must not hold at site 2. This means that it is impossible to find a pair of global transactions which violate global serializability. Hence, the assumption that the global serializability is not satisfied is wrong. Thus, the theorem holds.

**Theorem 2:** The scheme is deadlock-free.

**Proof:** When a deadlock occurs, a cycle can be found in the wait-for graph [4, 7]. Without loss of generality, it can be assumed that a cycle is found in the wait-for graph (see Figure-1).

In the cycle G4i is the subtransaction of global transaction Gj at site j, and time(G4i) < time(Gj). Two cases involving transactions waiting for each other need to be considered:
1. A transaction in the cycle is ready-to-commit.
   A subtransaction can only be in the ready-to-commit state for a limited period. This is because a
In this paper a transaction management scheme for MDBS has been proposed. The scheme is distributed. Each site manages the global transactions based on locally available information. Therefore, it has the potential to provide a higher degree of fault tolerance, and allows incremental growth of the system. The scheme preserves autonomy and heterogeneity of the LDBSs. It is fair and free from global deadlock. The global consistency is enforced through (a) additional operations on a dummy data item stored in the LDBSs, (b) forbidding accessing the dummy data item concurrently by different subtransactions, and (c) forcing the subtransactions to be committed in the order that they access the dummy data item. Freedom from deadlock is achieved by aborting the transactions which might block the execution of the transactions that have smaller timestamps. The aborting only occurs when the transactions with smaller timestamps request the execution of all their subtransactions to be carried out immediately. Therefore, the scheme in this paper is more flexible than [1], where the transaction must be committed in strict timestamp order. The fairness is achieved by assigning different execution priority to the transactions. The priority is based on the timestamps of the transactions. The priority of a transaction with a small timestamp is always higher than a transaction with a large timestamp. Therefore, the transaction with the smallest timestamp can always be executed.

6 Conclusions

In this paper a transaction management scheme for MDBS has been proposed. The scheme is distributed. Each site manages the global transactions based on locally available information. Therefore, it has the potential to provide a higher degree of fault tolerance, and allows incremental growth of the system. The scheme preserves autonomy and heterogeneity of the LDBSs. It is fair and free from global deadlock. The global consistency is enforced through (a) additional operations on a dummy data item stored in the LDBSs, (b) forbidding accessing the dummy data item concurrently by different subtransactions, and (c) forcing the subtransactions to be committed in the order that they access the dummy data item. Freedom from deadlock is achieved by aborting the transactions which might block the execution of the transactions that have smaller timestamps. The aborting only occurs when the transactions with smaller timestamps request the execution of all their subtransactions to be carried out immediately. Therefore, the scheme in this paper is more flexible than [1], where the transaction must be committed in strict timestamp order. The fairness is achieved by assigning different execution priority to the transactions. The priority is based on the timestamps of the transactions. The priority of a transaction with a small timestamp is always higher than a transaction with a large timestamp. Therefore, the transaction with the smallest timestamp can always be executed.

References


