"PRABHA" - A Distributed Concurrency Control Mechanism

Albert Burger and Vijay Kumar
Computer Science
University of Missouri-Kansas City
5100 Rockhill
Kansas City, MO 64110

We propose a non-preemptive, deadlock free concurrency control mechanism for distributed database systems. The algorithm uses a combination of transaction blocking and roll-back to achieve serialization. Unlike other locking mechanisms presented in the past, the algorithm proposed here uses dynamic attributes of transactions to resolve conflicts to achieve serialization. We argue that using the dynamic attributes of transactions economizes memory use and reduces conflict resolution time.

1. Introduction

Serialization of concurrent transactions is more difficult in distributed database systems (DDBS) than in centralized database systems because (a) the database is distributed among nodes, (b) message communication is necessary in transaction processing, and (c) messages may get delayed or lost during communication. For these reasons, Concurrency Control Mechanisms (CCMs) for distributed systems will have to do much more than centralized CCMs to achieve serialization. An efficient CCM for distributed systems, therefore, must have:

Independent *conflict* resolution capability. By independent conflict resolution capability we mean that every node is to be able to resolve conflict with a minimum or no help from other nodes where conflicting transactions have visited. For example, if transaction T1 conflicts with T2 at node N1 and transaction T2 conflicts with T1 at node N2, then the conflict decisions of N1 and N2 should be independent, as far as possible, of each other but must be the same (i.e. if N1 decides to *roll-back* T1 then N2 must also decide to roll-back T1). Such high degree of node independence in resolving conflicts requires that sufficient information about conflict transactions should be available to the nodes where a conflict has occurred. One of the common ways to accomplish this is using timestamps [BERN81].

***Deadlock* free.** The CCM should be deadlock free. In distributed systems, two types of deadlock may occur. One is *local* deadlock which is confined to a node and the other is *inter-node* deadlock where more than one node is involved in a deadlock. Dealing with local deadlock is not expensive but inter-node deadlock may not be that easy to detect and resolve.

Robust. The CCM should be robust, i.e., it should exhibit good performance in a lightly loaded system and acceptable performance in heavily loaded systems, and the commit operation should not be expensive (i.e., it should not require too many messages).

In the past several distributed CCMs (two-phase and non-two-phase) have been suggested [BERN81]. The main problem with these algorithms is that they either use some external ordering to pre-define the execution of concurrent transactions or they use timestamps to resolve conflicts or they are not deadlock-free. The problems with timestamping are generation and maintenance of timestamps. Since every transaction and data item are associated with unique timestamps requiring a large amount of memory to save them. Operational disadvantage is that the CCM resolves conflicts only by rolling-back transactions. The performance measurement of *basic timestamp* algorithms [KUM88a] has shown that the cost of roll-backs significantly affects the performance.

In this paper we report a distributed concurrrency control mechanism that is based on two-phase locking policy. Unlike other two-phase CCMs, our algorithm uses a subset of a set of dynamic attributes (see below) of transactions to resolve conflict. The execution order is not pre-determined, it is deadlock-free and we provide supportive arguments in favor of its usefulness and robustness.

2. Algorithm

A transaction acquires several attributes during its life time. Some relevant attributes are number of conflicts, number of entities locked by the transaction, the time the transaction waited for the desired data items etc. These attributes are related only with the underlying concurrency control mechanism and their values change continuously during its entire execution. We call these attributes *dynamic attributes* of transactions, and a subset of these attributes gives a transaction a unique status. We propose that this subset can be used to resolve conflicts and serialize transaction execution in distributed systems. The set of dynamic attributes (DA) we have selected for our algorithm is:

DA = {number of conflicts accumulated, number of locks acquired }

We selected number of conflicts since this parameter is inherent to concurrent execution of transactions and may change more frequently than other dynamic attributes. We selected number of locks acquired by a transaction since under two-phase policy the progress of a transaction depends on its successfully acquiring the desired lock. So

the number of locks gives a measure of transaction maturity (i.e., how far it has progressed in its execution) in terms of its resource utilization.

We decided to use two-phase policy for our algorithm since a number of performance studies (we list only a few) [AGA87, KUM87] show that CCMs based on this policy outperform other commonly known non-two-phase CCMs. In two-phase CCMs if the requestor (transaction making the lock request) is blocked then it may create a deadlock (as happens in general waiting algorithm [KUM87]). On the other hand if it is rolled-back then we face the problem of redundant roll-backs [KUM87]. The best approach is, therefore, to apply an "intelligent" decision to select the best possible operation (roll-back or blocking). A roll-back decision is intelligent, if it is least expensive (i.e., the roll-back restores fewer number of entities). We associate the cost of roll-back with the dynamic attribute: The number of conflicts a transaction (requestor) has accumulated so far. A larger number of conflicts means the transaction has travelled much further in its execution. It is possible that a transaction (requestor) may have accumulated a large number of conflicts but could not lock any data items. However, experience in transaction processing indicates that this happens rarely, or in a badly designed CCM we, therefore, exclude this possibility.

Algorithm "PRABHA" resolves conflicts either by blocking or by rolling-back the requestor. Taking action only on the requestor makes this algorithm deadlock-free and non-preemptive since the CPU was not being used to process the data item locked by the transaction. In order to explain the conflict resolution policy of this algorithm, we define a set called **Conflict Resolution Set** = {number of conflicts the transaction had so far, number of data items locked by the transaction}. In some cases, as we will see, these two attributes may be insufficient to resolve conflict correctly. In this situation unique identities of conflicting transactions are used. A transaction identity is different than a timestamp in the sense that the former never changes, it is an integral part of the transaction and can also be used in checkpointing.

A conflict between a requestor and a holder is resolved using their priorities. The priority of a transaction is computed using its DA set as follows. We assume

C_i = number of conflicts of a transaction T_i; i = 1, 2, ..., n and n being the number of transactions in the system.

E_i = the number of data items held by T_i; E = 1, 2, ..., D (size of the database).

P_i = priority of T_i.

```
if (Cj < Ck) then Pj < Pk
else if (Cj = Ck) then
        if (Ej < Ek) then Pj < Pk
        else if (Ej = Ek) then
                (* compare transaction identitities *)
             if (Tjid > Tkid) then Pk > Pj
```

This process guarantees to assign a unique priority to the requestor since the comparison of transaction identities will always succeed (inequality). A conflict at a node is resolved as follows:

> If the requestor's priority > the holder's priority then block the requestor otherwise roll-back the requester.

We formally present our algorithm with a set of procedures. We first describe the algorithm for a *fully* or *partially replicated* database and then we show how it works in *partitioned* (no duplication) databases. We distinguish two types of nodes for a transaction: **home node** we will also call it as **requesting node** (node a transaction originates) and **data node** (the node that holds a copy of the desired data entity). In the case of fully replicated database one node is home node and all other nodes are data nodes including the home node. At every node a status table is maintained that stores transaction identity, type of lock required by this transaction (read/write), number of conflicts, the number of data items the transaction has locked so far, and its standing (holder or requestor). The table is continuously updated and used in resolving conflict between a holder and a requestor.

3. Processing of lock requests

The processing of a transaction at its home node and at data nodes is done differently so we describe them separately.

3.1 Processing of lock requests originating from transaction's home node:

The home node sends lock request for the transaction to all data nodes. These data nodes resolve conflict using local information. The decision, grant or roll-back or block, is sent by the data nodes to the home node. The home node, after receiving these messages from data nodes, makes the final decision for the transaction which is based on a majority consensus as follows:

GRANT: If the majority of the messages received are grant message then the transaction's lock request is granted. A majority of grant message indicates that the majority of nodes support execution of this transaction.

BLOCK: If the majority of the messages received are block message then the final decision is to block the transaction.

ROLL-BACK: If the majority of the messages received are roll-back message then the final decision is to roll-back the transaction.

The final decision is sent to all data nodes which made the wrong decision and the states of conflicting transactions are updated at these nodes. A general problem, however, of majority consensus is that in some situations no transaction can get the majority of locks. For example, consider three transactions trying to lock a data item. Suppose there exist three copies of the desired data item and each transaction manages to lock one copy. In this case each transaction will get one grant message, one roll-back and/or one block message, thus no majority consensus will be present.

In the absence of majority consensus a transaction with the lowest ID (WINNER-ID) gets the lock on the data item. A requesting transaction can determine with which other transactions it is in conflict by looking at each denied-message it receives (each denied-message contains the ID of the transaction that caused it). If the requesting transaction's ID is equal to the WINNER-ID then the transaction gets the lock on the desired data item, otherwise the requesting transaction is rolled-back or blocked. The algorithm is presented in pseudo-code as follows:

```
begin
 send lock request message to all nodes which have a
 copy of the desired data item;
 wait until grant or denied messages from all data nodes
 arrived;
 if GRANT then
 begin
  allow requestor to lock the desired entity at all data
  nodes;
  send this decision via correction message (see below)
  to nodes that sent denied messages;
  add one to the number of data items locked by the
  requestor by sending update messages to those nodes
  where the requestor has visited so far;
 end
 else if   ROLLBACK then roll-back the requestor
  else     (* BLOCK*)
  begin
   block the requestor (loser);
   add one to the number of conflicts of the requestor at
   nodes where the requestor has visited by sending
   update messages;
  end;
end.
```

3.2 Processing of lock requests at Data nodes

At a data node, more than one lock requests may arrive simultaneously from several requestors.

```
A lock request arrives at a data node;
if no conflict then
begin
    record the requestor as a new holder of the requested
    data item;
    increase the number of data items locked by the
    requestor by one;
    send a grant message back to the requesting node
    (home node);
end
else (* there is a conflict *)
begin
 increase the number of conflicts of the requestor;
 compute the priority of the requestor;
 if   the requestor's priority > holder's priority  then
 begin
  block the requestor (* this may be a temporary
  blocking since this transaction may be rolled-back
  later when a roll-back message arrive from the home
  node of that transaction as explained above *);
  send block message to the requesting node (home
  node);
 end
 else
  send a roll-back message to the home node but defer
  any action on the requestor, i.e, keep the requestor
  transaction as blocked in the local table.
  (* A data node's roll-back decision may not be same
  as the final decision reached by the home node.)
end;
```

3.3 Modification of transactions' attributes at data nodes

At every node, a transaction's current status and the value of its attributes are recorded in a local table. The value of these attributes are changed by update messages. Update messages can only arrive for holder transactions, since requestor transactions in the table are blocked and cannot change their status. When an update message arrives the procedure goes as folllows:

Update the number of conflicts and number of data items of the transaction as directed by the update message from the home node. After updating transactions' attributes the conflict resolution is applied again which may change the status of a requestor transaction.

If a requestor transaction is to be rolled-back then
send roll-back message to the home node of transactions to be rolled-back. (* A requestor transaction needs to be rolled-back, if the holder transaction's priority became greater (due to the update of the attribute values) than the priority of a requestor transaction *)

3.4 Processing of a correction message at data nodes

A correction message is required to inform those data nodes that denied the lock request, that the final decision of the home is a grant. At these data nodes the holders of this data item will change their status to requestors of the data item and the transaction indicated in the update message becomes the new holder.

```
designate the correct holder in the table as indicated in
the correction message;
for the new holder
begin
 increment number of data items by one;
 decrement number of conflicts by one
end;
for all old holders
begin
 increment number of conflicts by one;
 decrement number of data items by one
end;
apply conflict resolution policy at nodes where
transactions status are changed from holder to new
requestor to decide either to roll-back or to block the
new requestors
```

for all old holders that changed their status (to requestors):
if the priority of an old holder > the priority of new holder
then
block the old holder
else roll-back the old holder;

3.5 Activation of blocked transactions

A blocked transaction is activated after it receives grant messages from the majority of data nodes. When a transaction committs, a data node selects the next requestor of the data just released (only if there are no outstanding holders of this data item) in first-come-first-served order . There may be more than one holders of the same data item in the case of read lock. In this situation nothing is done, otherwise a grant message is sent to the new holder's home node.

3.6 Restart of rolled-back transactions

A rolled-back transaction retains the past values of its attributes. These values are used to compute its priorioty when it is rescheduled for execution. This quarantees that the priority of a transaction always increases and avoids livelock (starvation).

4. Modification for partitioned databases

In a partitioned database since each node holds a unique partition, our algorithm requires comparatively less number of messages to synchronize the execution of transactions. No correction message is required to change the status of a transaction at a node and a majority of grants is not necessary to make a final decision. A transaction sends a lock request to the data node. The conflict resolution set is used to resolve a conflict if there is one. The decision (roll-back or blocking) is sent to the home node. The rest of the steps are identical to the replicated environment. Notice, there are no correction messages necessary.

5. Correctness

We establish the correctness of our algorithm by showing that every transaction eventually terminates and it is deadlock-free. It has been established [HSU87, ESW76, PAP79] that if well-formed transactions are executed in a two-phase way then the execution maintains serializability. Our algorithm is based on two-phase policy and, therefore, to establish its correctness, it is sufficient to show that no transaction will remain blocked for ever.

5.1 There is no starvation

A requestor, in our algorithm, is blocked if it has a higher priority than the holder. Every time a requestor is blocked due to a conflict, its number of conflicts is increased which improves its execution possibility. Suppose, T_j (requestor for data item X) with priority j, conflicts with T_k (holder of data item X) with priority k, and $j > k$. T_j will be blocked and its priority will be incremented by 1 to (j + 1). Now suppose T_m(requestor for X) conflicts with T_k where $m > k$, then T_m will be blocked and its priority will be incremented, If now T_k (requestor for data item Y) conflicts with T_j (holder of data item Y) T_k is rolled-back and T_j will be rescheduled. If there are several blocked transactions for the same data item FIFO scheduling should be used to avoid possible starvation. At every node a conflict will be resolved (may not be immediately due to message delay) in this way which guarantees that all blocked transactions will eventually terminate.

5.2 The algorithm is deadlock free

We show that our algorithm is deadlock-free by establishing that the condition which may cause a deadlock is never allowed. We proceed as follows:

$$T1 \rightarrow T2 \rightarrow T3 \rightarrow \cdots \rightarrow Tn\text{-}1 \rightarrow Tn \rightarrow T1$$

(T_j ---> T_k, means T_k is blocked by T_j)

Tn blocked by Tn-1 indicates Pn > Pn-1 (Priority of Tn > than priority of Tn-1)

Now if T1 conflicts with Tn then Pn > P1 must be correct. If this is the case then T1 will be rolled-back. This means that the transaction which may cause a deadlock, will always be rolled-back [KUM88b]).

Messages delay may create some side effects. One of such effects is, that there may temporarily exist two different versions of a transaction's priority at two or more different nodes. This might cause a "temporary deadlock". We do not regard this as a real deadlock, since the decision to roll-back a transaction is made on the basis of conflict resolution policy and not by using any deadlock detection algorithm. In this situation we "know" which transaction in the "temporary deadlock" will be rolled-back. Thus the "temporary deadlock" is solved as soon as the corresponding update message arrives at the node with the old priority value. Hence, there will never exist a permanent deadlock.

6. Examples

In the following we present two examples to illustrate the working of our algorithm. The algorithm uses different types of messages to achieve serialization. The structure of these messages are given below:

Request Message (RM): (requester id, home site, data item, lock mode, number of conflicts, number of data items)
Grant Message (GM): (requester id, data item, node id)
Block Message (BM): (id of blocked transaction, transaction ids of blocking transactions, node id, data item)
Rollback Message (RBM): (id of transaction to be rolled back, ids of rolling-back transactions, node id, data item)
Update Message (UM): (transaction id, number of conflicts, number of data items)
Correction Message (CM): (new holder ids, data item)

The local table at each node stores the following information about transactions visiting that node:

(transaction Id, lock mode, number of conflicts, number of data items, holder/requestor)

Example 1 (partitioned database):

T1 and T2 are two transactions, and N1, N2, N3 and N4 are four nodes. T1 originates at N4 and requests data item X, located at N3. T2 originates at N1 and requests data item Y located at N2 (Figure 1).

1. N4 sends a RM for T1, for data item X to N3; N1 sends a RM for T2, for data item Y to N2;
2. At N3, T1's request is granted: T1 is recorded as the holder of data item X with 0 conflicts and has 1 data item. A Grant-message is sent back to N4 from N3;
3. At N2, T2's request is granted: T2 is recorded as the holder of data item Y with 0 conflicts and has1 data item. A Grant-message is sent back to N1 from N2;

T1 now requests data item Y and T2 requests data item X.

4. N4 sends a RM for T1, for data item Y to N2;
5. N1 sends a RM for T2, for data item X to N3 ;
6. At N2 a conflict between T2 (holder of Y) and T1 (requestor for Y) is detected so the number of conflicts for T1 is updated by 1. The initial number of conflicts for T2 in the local table of N2 is 0. The priority of T1 is higher than priority of T2, so T1 is blocked at node N2. A BM is sent to N4 from N2.
7. At node N3 (similar to 6) T2 is blocked at N3. A BM is sent to N1 from N3 (see figure 2).
8. The arrival of the BM from N2 to N4 indicates that T1 had a conflict and hence all nodes where T1 visited earlier (in our example N3 only) need to be informed about the change of T1's priority. An UM is, therefore, sent to N3 from N4 to record the change in T1's priority, i.e., number of conflicts is updated by 1.
9. (Similar to 8). An UM is sent from N1 to N2 to inform N2 about the change of T2's priority.
10. Since the priority of T1 (holder at N3) is higher (T1's identity < T2's identity) than the priority of T2 (blocked requestor), T2 is selected to be rolled-back. A RBM is sent to N3 from N1.
11. Since the priority of T2 (blocked requestor at N3) is lower (T1's identity < T2's identity) the priority of T1 (holder at N3) no further steps need to be taken at N3.
12. When RBM from N1 arrives at N3, T2 is rolled-back

Example 2 (partially replicated database)

T1 and T2 originate at N1 and N5 respectively and request data item X. Copies of X exist at N2, N3 and N4 (see figure 3).
1. N1 sends RMs for data item X for T1 to N2, N3 and N4.
2. N5 sends RMs for data item X to N2, N3 and N4.
3. At N2 and N3 the requests of T1 (from N1) arrive before the requests of T2 (from N5). At N2 and N3, T1 is recorded as holder of data item X and GMs are sent to N1 from N2 and N3. When the RMs of T2 (from N5) arrive at N2 and N3 they conflict with T1. Number of conflicts for T2 is incremented at N2 and N3. At N2 and N3 priority of T2 is higher than the priority of T1, so BMs are sent from N2 and N3 to N5 to block T2.

At N4 the RM of T2 arrives before the one of T1. T2 is granted as holder. GM is sent from N4 to N5. When request from N1 arrives at N4, transaction T1 conflicts with T2 (recorded holder). Number of conflicts for T1 incremented by 1. T1 is blocked by T2 at N4 and a BM is sent from N4 to N1.
4. At N5, the majority of the messages received are BMs. T2 is blocked.
5. At N1, the majority of the messages received are GMs. T1 is granted access to data item X. Since N4 made a wrong decision, a Correction-message is sent from N1 to N4, to inform N4 about the actual outcome of the majority vote.
6. When the CM from N1 arrives at N4 the status table is updated accordingly, i.e., T1 is recorded as the holder and T2 becomes the requestor, and the recorded priorities are corrected.

7. Summary and Conclusion

In this paper we presented a two-phase distributed concurrency control mechanism. The algorithm is deadlock free and non-preemptive. The unique feature of this algorithm is that it utilizes transaction's inherent attributes to resolve the conflict. This scheme avoids setting any static execution ordering and additional information to achieve serialization. It is a distributed algorithm, since there is no centralized control over locking and releasing activities. The algorithm utilizes an "intelligent" scheme to resolve conflicts among conflicting transactions. It never hinders the progress of active transactions (i.e., transaction that is using CPU resources). Every node takes decision independently on a conflict, i.e., a node does not enquire other node(s) what action it should take to resolve a conflict). This individual decision may be correct but may not be the final. In the later case the final correct decision overrides the decision of the data nodes.

We would like to point out that our algorithm is designed in such a way that we can vary the priority schema without affecting the rest of the algorithm. It is part of further research work to determine an optimal function of dynamic attributes for a transaction's priority.

The algorithm does not use timestamps thus aviods expensive maintainence of the timestamps. It is possible that the algorithm may use more messages to achieve serialization, however, the message overhead may not have significant affect on the performance of the algorithm. We are in the process of measuring the performance of this algorithm and the results will be reported in our future work.

We conclude that the inherent attributes of transactions are a better set of parameters for resolving conflicts under two-phase policy. The algorithm PRABHA can be improved further by reducing the number of messages required to achieve synchronization and will also be reported in our future work.

References

[AGA87] R. Agrawal, M.J. Carey, and L.W. McVoy, "The Performance of Alternative Strategies for dealing with Deadlocks in Database Management Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 12, Dec. 1987.

[BER81] P. Bernstein, and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 1, No. 2, June 1981.

[ESW76] K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in database Systems", *Comm. ACM* Vol. 19, No. 11, Nov. 1976.

[HSU87] M. Hsu and B. Zhang, "The Mean Value Approach to Performance Evaluation of Cautious Waiting", Submitted for Publication, 1987.

[KUM88a] V. Kumar, "Performance Comparison of Database Concurrency Control Mechanisms based on Two-Phase Locking, Timestamping and Mixed Approach", *Information Sciences: An International Journal*, Vol. 47, No. 1, 1988.

[KUM88b] V. Kumar and Meichun Hsu, "A Superior Two-Phase Locking Algorithm and Its Performance", *Information Sciences: An International Journal*, Vol. 47, No. 3, 1988.

[KUM87] V. Kumar, "An Analysis of the Roll-back and Blocking Operations of Three Concurrency Control Mechanisms", *NCC*, Chicago, June, 1987.

[PAP79] C.H. Papadimitriou, "The serializability of concurrent database updates", *JACM*, 26, 4 (Oct. 1979).

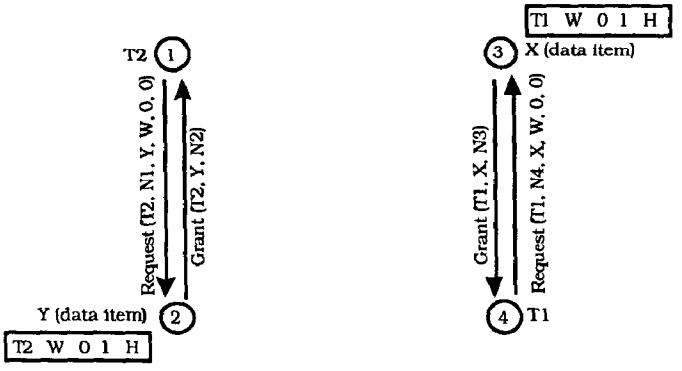

Figure 1

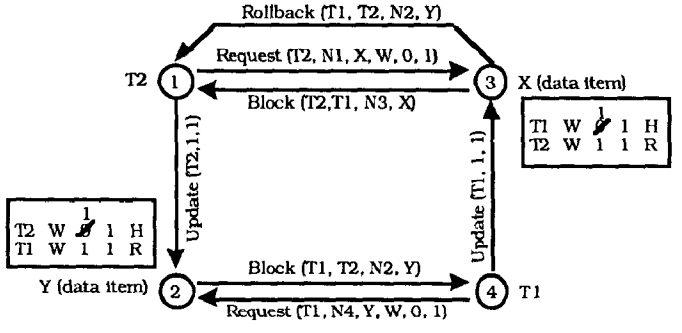

Figure 2

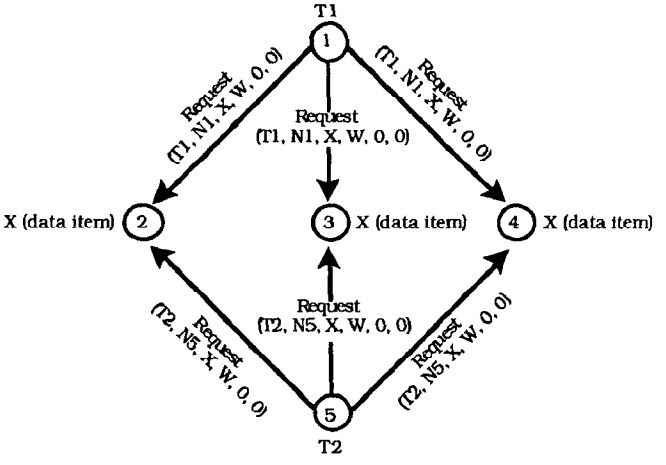

Figure 3

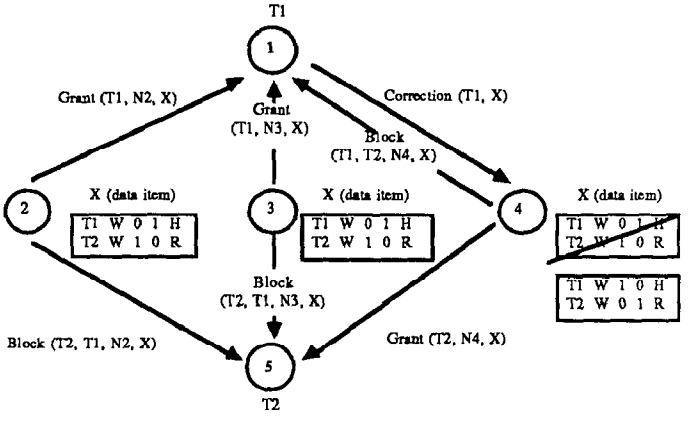

Figure 4