Timestamp Based Certification Schemes for Transactions in Distributed Database Systems

Mukul K Sinha
National Centre for Software Technology
Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005

P D. Nanedikar & S L. Mehndiratta
Indian Institute of Technology, Powai, Bombay 400 076

Abstract

Two certification schemes for transactions in distributed database systems are proposed The schemes do not construct any conflict graph but use system_wide unique timestamps to serialize certification requests of concurrent transactions. In the first scheme, the distributed certification scheme, transactions are assigned timestamp when they request for certification A transaction get certified if, at no site, its certfication request conflict with that of a transaction with higher timestamp. In the second scheme, the negotiated certification scheme, the system negotiates with participating data items, and comes up with a timestamp, if possible, with which the transaction will not face any conflict with concurrent transactions and will get certified The two_phase commit protocol can very easily be integrated with either of the two certification schemes, and it is shown that the schemes do not need any extra message cost to guarantee failure atomicity

1. Introduction

In a database system, where access conflicts (read or write access) among concurrent transactions are infrequent, it is proposed [BADA 79, KUNG 81] that accesses of transactions should be permitted tentatively with no synchronization whatsoever, and the serializability [BERN 79] of a transaction should be verified only when the transaction attempts to terminate In other words, when a transaction issues its End Transaction statement the system checks whether the access conflicts of the transaction with other concurrent transactions are serializable or not. If yes, the transaction is certified and thereby committed, otherwise it is aborted. The methods following this approach are known as certification schemes [BADA 79], or optimistic methods for concurrency control [KUNG 81]. This approach is expected to be more efficient since it does not use locks at all, and thereby it eliminates the associated delays This approach relies on transaction backup as the only control mechanism, and hence, it is attractive for every dominated database systems. Though the certification schemes have been appreciated as promising for centralized database systems, serious doubts have been raised on its applicability for distributed database systems [BERN 81] Bhargava [BHAP 82] has proposed an optimistic concurrency scheme for distributed systems where the history of committed transactions are maintained, and a dynamic conflict graph (similar to a wait-for graph) is constructed at each site to honour, or reject, the certification of a transaction In contrast, we have proposed a scheme which does not construct any conflict graph, but uses a system-wide unique timestamp to serialize certification of concurrent transactions at various sites Schlageter [SCHL 81] has tried to develop an optimistic method for concurrency control in distributed database systems which uses purely local transaction number for certification at each site, but unfortunately it does not give serializable execution sequence always The failure of Schlageter scheme is shown through an example in [SINH 85].

In a centralized certification scheme each transaction has to go through a critical section [DIJK 68] where the test of serializability of its execution sequence with respect to other concurrent transactions is done [KUNG 81]. If the test fails, the transaction comes out of the critical section and aborts. But if the serializability of the transaction is certified, the transaction commits, installs all its tentative updates (if any) permanently and leaves the critical section. Certification and installation of updates are parts of the critical section

The natural extension on this scheme for a distributed database system will be to execute a transaction as a set of processes a set of local subtransaction processes, one local subtransaction at each participating site, and a transaction manager process that coordinates the participating local subtransactions [GRAY 78] A local subtransaction does all actions of the transaction related with that site, with no synchronization whatsoever. When a transaction is through with its activities and wants to complete, the transaction manager follows two phase certification scheme to coordinate certification of all participating local subtransactions In the first phase, the certfication phase, the transaction manager asks each of its local subransactions to get certified locally. For local certification, a subtransaction enters into the local critical section of that site, where the test of serializability of the local subtransaction with respect to other local subtransactions running at that site is done After all the participating local subtransactions inform the transaction manager about the success of their local certifications, the transaction manager enters the second phase, the installation phase. In the installation phase, it asks all participating local subtransactions to install their tentative updates (if any) permanently. Since, the certification and installation of updates are integral part of the critical section, after certification local subtransactions will have to wait within their local critical sections until they get signal from the transaction manager to install (or abort) their updates. This, in turn, may lead to a situation where a group of local subtransactions (local subtransactions of different transactions) may get involved in a certification_deadlock a situation in which each member subtransaction of the group waits indefinitely to enter in its local critical section which is being visited by some member subtrasection of the group (i.e., a local subtransaction of some other transaction)

The requirement for a local subtransaction to wait inside a critical section is the primary cause for the situation of certification_deadlock to occur Waiting of a local subtransaction within a critical section can be avoided provided the installation_phase is disintegrated from the certfication_phase by pushing it out of the critical section. This mechanical separation may avoid the certification_deadlock altogether, but it introduces a problem as well even if a transaction gets local certification at all participating sites the global serializability of its execution sequence cannot be guaranteed. Local certifications of a set of conflicting transactions can get interleaved, and this interleaving robs off the guarantee of global serializability, the reason on which the applicability of certification scheme in distributed database systems was questioned in [BERN 81].

In this paper, we present certification schemes for distributed database systems where the installation_phase is not a part of the certification critical section. To guarantee global serializability among conflicting concurrent transactions, we use system-wide unique timestamps for honouring, or rejecting, the certification requests of conflicting transactions at various sites. We have decomposed the certification request into Read_Certify and Write_Certify requests. A Read_Certify (Write_Certify) request faces a conflict if it requests for certification on the data item where a Write_Certify (Read_Certify) request of some other trasaction (active or committed) has already been honoured. We characterize the conflict to be antagonistic if the Read_Certify (Write_Certify) request is having timestamp smaller than that of the Write_Certify (Read_Certify) request which is already honoured. In the distributed certification scheme based on timestamps, a certification request facing an antagonistic conflict is rejected, preventing a certification deadlock.

We have extended the distributed certification scheme based on timestamps by incorporating the concept of commutability of transaction [PAPF 82, SINH 83] and have come up with the negotiated certification scheme that salvages many transactions that would have got aborted in the distributed certification scheme. In this scheme, the system negotiates with participating data items for a timestamp with which the trasaction will not face conflict with concurrent transactions. If the negotiation succeeds, the transaction gets certified with that timestamp, otherwise it is aborted.

In section 3, we present the distributed certification scheme based on timestamps, and in section 4 we discuss the integration of the certification scheme with the two_phase commit protocol. In section 5, we present the negotiated certification scheme, and in the end we summarize the salient points of the paper

2. The Distributed Certification Scheme Based on Timestamps

We present the distributed certification scheme in the context of the abstract data model and the transaction processing model Our model has some similarity with the model presented in [EFRT 76], REED 78], but in our scheme there is no synchronization until the transaction reaches the end of its execution.

2 1 The Distributed Database Model

In a distributed database system, the information is spread across a collection of nodes (or sites) interconnected through a communication network Each node has a system-wide unique identifier (site_id), and nodes communicate through messages All messages sent arrive at their destinations in finite time, and the site-to-site communication is pipelined. Each site is equipped with a logical clock, and clocks of all sites are loosely synchronized [LAMP 78]. The system provides a primitive, Get timestamp(), which on being invoked, generates a system-wide unique timestamp with the help of its site_id and the local clock Greater than (>) and less than (<) relations for timestamps are defined as they are given in [THOM 79] While comparing two transactions, we call the transaction having smaller (greater) timestamp as the older (younger) transaction.

The model consists of two basic entities process and data item (or object) A process is an autonomous active entity that is scheduled for execution Every process has a system-wide unique name, called process id, and they communicate with each other through messages Data items are passive entities that represent some independently accessible piece of information. Each data item is associated with a process (called the data manager) that has the exclusive right to access it.

2 2 The Object Model

A data item is viewed as a <name,version> pair. The version of a data item is represented by <val,tread,twrite>, where val gives its value, tread gives the timestamp of the youngest transaction that read this data item, and twrite gives the timestamp of the transaction that created this value The twrite field is also called as the version_id of the version A data item can be in either of the two states normal state and transient state In transient state, a data item can have apart from the version, one or more tokens, i.e., temporary versions [REED 78]. A token has structure similar to a version, and to distinguish components of a version from that of a token we will use subscripts $_v$ and $_t$ respectively The set of primitives that are provided by a data manager to operate on a data item are as follows:

```
Read_Data(data_item) returns <value,version_id>

Read_Certify(data_item,version_id,timestamp)
    returns <request_certified boolean,token_id>

Write_Certify(data_item,new_value,timestamp)
    returns <request_certified·boolean,token_id>

RW_Certify(data_item,version_id,
                              new_value,timestamp)
    returns <request_certified boolean,token_id>

Install_Token(data_item,token_id)

Delete_Token(data_item,token_id).
```

Except the certify requests, Read_Certify, Write_Certify, and RW_Certify, no other request is ever rejected by a data manager. There is no concept of locking a data item, and a data manager will honour a Read_Data request at any time, and irrespective of its state. All the three certify requests carry the timestamp of the invoking transaction as one of its parameter (discussed in section 2 4) A certify request, if it gets honoured by the data manager, creates a token for the data item, and the data item makes a transition from normal state to transient state (if it is not in transient state already). The three certify requests create tokens of three different kinds, and they are Read token, Write token, and RW token respectively. Each data item is provided with two queues, Rtoken Q and Wtoken Q, where all its associated Read_tokens and Write_tokens are stored, in ascending order of their fields, $tread_t$ and $twrite_t$, respectively Since an RW_token is a union of a Read_token and a Write_token, a copy of it is attached in both the queues On being invoked a data manager processes these requests in the following ways

Read_Data(data_item)· When a data manager receives a Read_Data request it fetches the version of the

data item, and returns the value along with its version_id which is nothing but the field twrite. Later, when the transaction wants to commit, it uses version_id as a parameter of the Read_Certify request.

Read_Certify(data_item,version_id,timestamp) The requester provides the version_id of the version which it had read earlier, and wants the data manager to certify that its read request does not face any antagonistic conflict with other transactions. On successful certification, the data manager creates a Read_token which is queued in the Rtoken Q

```
if twrite_v ≠ version_id
   then request_certified := false          ---(i)
   else if Wtoken Q is empty
          then request_certified  = true
          else begin
               get the oldest Write_token, t,
                   from Wtoken Q,
               if timestamp < twrite_t
                then request_certified  = true
                else request_certified  = false
               end,                         ---(ii)

if request_certified
   then begin
        create a Read_token, t,
               where tread_t := timestamp,
               %--other fields are not used --\
        state = transient
       end,
send response to the requester accordingly,
```

Write_Certify(data_item,new_value,timestamp): The user requests the data manager to write the new_value tentatively, and provides the timestamp so that the data manager can certify that this tentative write does not face any antagonistic conflict with other transactions. On successful certification, the data manager creates a Write_token which is queued in Wtoken Q

```
if Rtoken Q is empty
   then if tread_v < timestamp
          then request_certified  = true
          else request_certified  = false
                                          ---(iii)
   else begin
        get the youngest Read_token, t,
            from its Rtoken Q,
        if tread_t < timestamp
           then request_certified  = true
           else request_certified  = false
       end,                               ---(iv)

if request_certified
   then begin
        create a Write_token, t,
               where val_t  = new_value,
                     tread_t  = tread_v,
                     twrite_t := timestamp,
        state = transient
       end,

send response to the requester accordingly,
```

RW_Certify(data_item,version_id,new_value,timestamp): This call is nothing but the combination of a Read_Certify and a Write_Certify calls On successful certification the data manager creates only one token, an RW_token (say, it is t), a union of a Read_token and a Write_token, i.e., tread_t as well as twrite_t is set to timestamp A copy of the RW_token is queued in Rtoken Q as well as Wtoken Q

As shown above, a certify request can face failure on four conditions Condition(i) and Condition(ii), in case of Read_Certify, and Condition(iii) and Condition(iv), in case of Write_Certify. Again, the failures on Condition(i) and Condition(iii) are due to committed transaction, but that on Condition(ii) and Condition(iv) are due to the existence of tokens. It is possible that the transaction which created the token (causing the abortion of the certify request) may get aborted. And hence, rejecting a certification request due to the existence of a token is too harsh a decision This issue is discussed in section 4

A user whose certify request has been honoured by the data manager must send either an Install_Token request or a Delete_Token request following that. A data manager never honours an Install_Token or a Delete_Token request otherwise

Delete_Token(data_item,token_id): On receiving this request the data manager removes the token from the system, whether it is in Rtoken Q, or in Wtoken Q, or in both

```
if token_id exists in Rtoken Q
   then delete it from Rtoken Q;

if token_id exists in Wtoken Q
   then delete it from Wtoken Q,
if no token is left then state = normal,
```

Install_Token(data_item,token_id): In honouring this request a data manager follows the the Thomas Write Rule [THOM 79]

```
if the token_id (say, t) exists in Rtoker Q
   then if it is not an outdated token
          then begin
               %--a version_field is modified\
               tread_v  = tread_t,
               mark all Read_tokens ahead of t
                    in Rtoken Q as outdated,
               remove t from Rtoken Q
               end
          else remove t from Rtoken Q;

if the token_id (say, t) exists in Wtoken Q
   then if it is not an outdated token
          then begin
               %-a field of token is modified\
               tread_t  = max(tread_v, tread_t);
               delete the version and make t
                the version of the data item;
               mark all Write_tokens ahead of
                    t in Wtoken Q as outdated
               end
          else remove t from Wtoken Q,

if no token is left then state = normal,
send acknowledgement to the requester,
```

2.3 The Distributed Transaction Processing Model

A user operates in an interactive mode with the system, and the following five operations are defined at the user-system interface level [BERN 81].

```
Begin_Transaction,
Read(data_item),
Write(data_item,new_value),
Abort_Transaction,
End_Transaction.
```

A user's transaction is a sequence of Read and Write operations on participating data items bracketed between Begin_Transaction and Abort_Transaction or End_Transaction operations. A Begin_Transaction call results in the creation of a transaction manager at the user's site which processes all user's requests following it. The transaction manager initializes its subtransaction table that keeps information about the set of local subtransactions participating in this transaction. To start with the table is empty When it receives a Read(data_item)/Write(data_item, new_value) request, it checks whether or not a local subtransaction has been created at the site of the requested data item. If it exists, it passes on the request to the local subtransaction, and returns the reply received to the user.

But, if a local subtransaction does not exist at that site it creates a local subtransaction there, and to which it passes on the request It also updates its subtransaction_table Since the model follows an optimistic concurrency control scheme a user's Read and Write requests never fail.

When a local subtransaction gets created, it is in started state It initializes its activity table that functions as a buffer to keep information about data items accessed by it. For each data item entry in the activity_table, there are following fields· (i) the name of the data item, (ii) its current_value, (iii) the version_id (only for those data items that have been read by the user, and is useful for read certfication in the end), (iv) the token_id (filled after certification), (v) a read flag, set if the data item is read, and (vi) a write flag, indicating whether the user has updated the data item or not An entry can have some fields set to null

When a subtransaction receives a Read(data_item) request from its transaction manager, it checks whether or not the data item exists in the activity table. If it exists, its current_value is returned to the user. If it does not exist, it issues a Read_Data(data_item) request to the data manager. After it receives the response from the data manager, an entry in the activity_table is created where the name, the current_value (set to the value of the data item returned), and the vesion_id fields are filled, and the read_flag is set When a subtransaction receives a Write(data_item,new_value) request from its transaction manager, it again checks the activity_table for an entry of the data_item If it exists, it overwrites the current_value in the table by the new_value supplied, and sets the write_flag. If it does not find the data item in the table, it creates an entry in the activity table and fills accordingly. Please note that it does not issue any request to the data manager.

When a transaction manager receives an Abort_Transaction command from the user, it sends abort signal to all its local subtransactions (by going through its subtransaction_table), and aborts itself When a local subtransaction in started state receives an abort_signal from its transaction manager it discards its activity_table and terminates But when a transaction manager receives the End_Transaction command from the user, it acquires a system-wide unique timestamp by issuing the Get_Timestamp() call, and follows the distributed certification scheme

2.4 The Distributed Certification Scheme

The distributed certification scheme follows two phase certification protocol. In the first phase, i.e., the certificetion_phase, the transaction manager sends a certify signal to all participating subtransactions, and waits either for a certification_success, or for a certification failure signal from each one of them. If it receives a cerification_failure signal it enters the abort phase, sends abort signal to all participating subtransactions, and terminates. But, if it recives a certification_success signal from each of its particpating subtransactions it commits, and enters the commit phase. In the commit_phase, it writes commit_flag in a stable place [LAMP 76], sends commit signal to all participating subtransactions, and waits for their acknowledgements. After receiving acknowledgement of its commit signal from every participants it terminates.

Similar to the transaction manager, a local subtransaction also goes through two phases. When a subtransaction receives a certify signal from its transaction manager it records its started state in a stable place and enters the local certification critical section, the first phase of the two_phase certification protocol. Each site has its own local certification critical section which is as follows·

```
< for each entry in its activity_table
  do begin
      if read_flag and write_flag
         then send RW_Certify request
                   to the data manager
         else if Read_flag
                 then send Read_Certify request
                           to the data manager
                 else send Write_Certify request
                           to the data manager,
      % -- data manager responds with
           <request_certified boolean,token_id>\
      if not request_certified
         then loopexit
         else save token_id in the activity_table
     end;
  if request_certified
     then signal certification_success
                 to the transaction manager
                 and set state = certified
     else signal certification_failure
                 to the transaction manager
                 and terminate,
>
```

< > denotes the critical section

On successful certification, the local subtransaction comes out of the certification critical section, and it is in certified state. It waits for an commit/abort signal from its transaction manager, and on receiving the signal, it executes the following

```
for each entry in its activity_table
    do fetch the token_id of the entry (say, t)
       if it is a commit signal
          then send Install_Token(data_item,t)
                    to the data manager
          else send Delete_Token(data_item,t)
                    to the data manager;

send acknowledgement to the transaction manager
     and terminate;
```

Please note that unlike the centralized certification scheme [KUNG 81] this portion of the code is not part of a critical section, since an Install_Token request or a Delete_Token request is related to a specific token of the data item, and it is bound to get honoured irrespective of its arrival time.

It is important to realize at this point that the presence of a token may forbid a certification request of some other subtransaction getting honoured, e.g., the subtransactions facing Condition(ii) and Condition(iv). Hence, to inhance the success of certification requests, the local scheduler should give priority to the processing of abort signal received by a subtransaction in certified state.

3. Failure Atomicity and the Distributed Certification Scheme

While presenting the certification protocol, we had assumed a reliable distributed system. But, a distributed system is prone to failures, and to ensure failure atomicity a transaction manager, on receiving the End_Transaction request, must follow the two phase commit protocol [GRAY 78] in communicating with its local subtransactions. The two_phase certification protocol and the two_phase commit protocol can be nicely integrated since the logical point in the execution of the transaction manager (and local subtransactions), on which the phase does change is same for both the protocols. Hence, a certify request can be accompanied by a pre commit request [GRAY 78], and a commit/abort signal can be treated as a signal for both the protocols, i.e., no extra message cost is paid to guarantee failure atomicity.

4 A Distributed Certification Scheme Based on Negotiations

In the distributed certification scheme based on timestamp, the certify requests of concurrent conflicting transactions are ordered according to their timestamps which is assigned to them, by the system, prior to the occurance of the conflict. A certify request that reaches a site out of order gets aborted, resulting in the restart of entire transaction. In place of assigning a timestamp prior to the conflict, if it is assigned after the occurance of the conflict it is possible to salvage the transaction which otherwise would have got aborted [SINH 83]. We have incorporated the concept of assigning a timestamp after a conflict has occurred in our certification scheme based on timestamp, and have come up with a new scheme, the negotiated certification scheme. A transaction that would have got aborted, in the distributed certification scheme, on Condition(ii) and Condition(iv) can be salvaged in this new scheme.

The negotiated_certification scheme distinguishes itself from the distributed certfication scheme based on timestamps in the following ways.

(i) Contrary to the distributed certification scheme, where a transaction manager accuires a unique timestamp from the system independently, in this scheme, the transaction manager tries to get a unique timestamp, that will guarantee its serializability, by going through negotiation with participating data items.

(ii) The transaction manager negotiates directly with the participating data managers, and it does not create any local subtransaction All tasks, earlier done by a local subtransaction (e.g., maintaining activity_table, interacting with data managers, etc.), are done by the transaction manager itself

(iii) When the negotiation is going on among the transaction manager and all its participating data managers, all data items participating in the transaction are put under negotiation locks And hence, no process has to go through any critical section.

(iv) The negotiated_certification scheme has three phases: Negotiation phase, Certification phase, and Installation phase

4.1 The Object Model for the Negotiated Certification Scheme

For negotiated certification, the basic object model needs modification, and the modified object model provides four more primitives The first three primitives are negotiation calls (corresponding to the three certification calls described in section 2 2), and the fourth primitive permits a transaction manager to abandon its negotiation with participating data managers It also gives different semantics to the three certify primitives described earlier in section 2 2.

```
Read_Negotiation(data_item,version_id)
      returns <negotiation_feasible boolean,
                              range of timestamps>

Write_Negotiation(data_item)
      returns range of timestamps

RW_Negotiation(data_item,version_id)
      returns <negotiation_feasible boolean,
                              range of timestamps>

Withdraw_Negotiation(data_item).
```

In the modified object model, a data item offers two separate negotiation_locks which are independent of each other read_negotiation lock and write_negotiation lock We mention rw_negotiation lock when the read_negotiation lock as well as the write_negotiation lock of the data item is accuired by the same process When a data item is under a negotiation_lock, the data manager keeps the negotiation requests of the same type pending, and rejects a certify request of the corresponding type, other than that received from the owner of the lock. Processing of all other requests (Read_Data, Install_Token, and Delete_Token) does not get affected by the presence of a negotiation_lock For the owner of the negotiation_lock, only two types of requests are valid, a certify request of corresponding type, or the Withdraw_Negotiation request. Any of the two requests, on completion, will remove the negotiation_lock A negotiation_lock should be distinguished from other locks since (i) it forbids only negotiation request of its type and (ii) the negotiation_lock is for extremely short period, and it is withdrawn by the very next request of the owner. Again, to make it resilient to failure, each negotiation lock is associated with a timeout.

Read_Negotiation(data_item,version_id): The requester provides the version_id of the version that it had read earlier, and it wants to negotiate for a timestamp (asking a range of timestamps), if possible, with which it will not face any antagonistic conflict with those concurrent transactions that have accessed this data item. The data manager keeps the request pending if the data item is under a read_negotiation lock.

```
if twrite_v ≠ version_id
   then negotiation_feasible = false
   else begin
        negotiation_feasible = true;
        if Wtoken Q is empty
           then range = <twrite_v .∞>
           else begin
                get the youngest Write_token,
                    t, of Wtoken Q,
                range = <twrite_v..twrite_t>
                end
        end;

if negotiation_feasible
   then put read_negotiation lock
           on the data item;
send response to the requester accordingly;
```

Write_Negotiation(data_item): The data manager keeps the request pending if the data item is under a write_negotiation lock. A Write_Negotiation request never fails.

```
if Rtoken Q is empty
   then range := <tread_v..∞>
   else begin
        get the youngest Read_token, t,
            of Rtoken Q;
        range = <tread_t.. ∞ >
        end;

put write_negotiation lock on the data item;
send range as the response to the requester;
```

RW_Negotiation(data_item,version_id): It is a combination of a Read_Negotiation and a Write_Negotiation requests. A user who has read as well as written the data item will issue this request, negotiationg for a timestamp (or a range of timestamps) with which it will not face any antagonistic conflict with those concurrent transactions that have accessed this data item. The data manager keeps the request pending if the data item is either under a read_negotiation lock, or under a write_negotiation lock.

```
if twrite_v ≠ version_id
   then negotiation_feasible = false
   else begin
        negotiation_feasible = true,
        if Wtoken Q is empty
           then range_r = <twrite_v ∞>
           else begin
                get the youngest Write_token,
                    t, of Wtoken Q;
                range_r = <twrite_v..twrite_t>
                end
        end;

if negotiation_feasible
   then begin
        if Rtoken Q is empty
           then range_w = <tread_v..∞>
           else begin
                get the youngest Read_token,
                    t, of Rtoken Q;
                range_w := <tread_t.. ∞>
                end;
        range := range_r ⋂ range_w;
        if range is empty
           then negotiation_feasible = false
           else put rw_negotiation lock
                   on the data item;
        end,
send response to the requester accordingly;
```

Withdraw_Negotiation(data_item): On receiving this request, the data manager checks the identity of the owner of the negotiation_lock. If the requester is the owner, the negotiation_lock is withdrawn, and a pending negotiation request is chosen from the queue (if any). If the data item is not under negotiation_lock, the request is ignored.

Certify Requests: A certify request of a process gets accepted by the data manager only in the case the data item is under negotiation_lock owned by the same process, not otherwise. Since a certify request carries the negotiated timestamp, the certification is bound to succeed, provided the negotiation_lock is not broken by timeout. A certify request, once honoured, creates a token of corresponding type (as discussed in section 2.2) and then, it releases the negotiation lock.

4.2 The Transaction Processing Model for Negotiated Certification

The user interface to the system remains unchanged, but the activity of the transaction manager changes qualitatively.

There is no concept of 'ocal subtransaction, and the transactior manager communicates directly with the participating data managers.

When a user issues a Begin_Transaction request, it results in the creation of a transaction manager at the user site. The transaction manager initiates its activity_table (same as that of local subtransactions in the distributed certification scheme), and is in started state. When it receives any of the requests, Read(data_item), Write(data_item, new_value), and Abort_Transaction from the user, it operates in the same way as a local subtransaction does in the distributed certification scheme, i.e., issuing appropriate calls to the data managers, updating the activity_table, etc. But when it receives an End_Transaction command from user, it follows the negotiated certification scheme.

4.3 The Negotiated Certification Scheme

The negotiated_certification scheme follows the three phase negotiated certification protocol In the first phase, the negotiation phase, the transaction manager scans through its activity_table, and sends appropriate negotiation requests to the participating data managers. If the negotiation requests get granted by the data managers (i.e., all participating data items are put under either read, or write, or rw_negotiation lock), the transaction manager receives a range of timestamps from each of the participating data managers. It computes the intersection of the ranges of timestamps received If the intersection is empty (or any of its negotiation requests get rejected) it means that the tranaction manager is unable to acquire, through negotiation, a unique timestamp which would guarantee the global serializability of the transaction In this case, it sends Withdraw_Negotiation to data managers that honoured its negotiation request, and terminates.

If the intersection of the ranges of timestamps received is non empty, the negotiation is successful, and the transaction manager chooses the lowest timestamp from the intersection, called the negotiated timestamp. It sends certify requests to all its participating data managers, along with the negotiated timestamp, and enters the certification phase. Since all participating data items are put under a negotiation_lock, a transaction manager will receive certification_success signals from its participating data managers, unless some locks got broken due to timeout. A successful certification creates a token for the data item. If the transaction manager receives one or more certification_failure signals, it sends Delete_Token request to all data managers that have honoured its certification request, and terminates. On the other hand, if the transaction manager receives certification_success signals from all its participating data managers, it writes commit_flag in a stable place, enters the commit phase and sends Install_Token requests to all participating data managers. On receiving acknowledgements of all its Install_Token requests, the transaction manager terminates.

4.4 Failure Atomicity and the Negotiated Certification Scheme

To guarantee failure atomicity, the transaction manager follows two_phase commit protocol while communicating with the participating data managers. Similar to the case of distributed certification scheme (discussed in section 3), the two_phase commit protocol can very easily be integrated with the latter two phases of the three_phase negotiated_certification protocol.

5. Summary

(i) We have proposed a timestamp based certification scheme for transactions in distributed database systems. A transaction may fail to get certified if its certificatior request reaches a site later than that of a conflicting younger transaction.

(ii) We have proposed a negotiated_certification scheme for transactions where the system negotiates with participating data items for a timestamp with which the transaction will not face conflict with concurrent transactions, and will get certified.

(iii) Either of the two certification schemes can be integrated with the two_phase commit protocol so appropriately that the integrated scheme does not pay any extra message cost for guaranteeing failure atomicity.

Acknowledgement

The authors would like to thank Ron Obermarck for his discussion with Mukul Sinha when the latter visited IBM Research Laboratory, San Jose in May 1984. The comments of Ron Obermarck on the concept of commutable transactions contributed to the idea of the negotiated certification scheme.

References

[BADA 79] D Z. Badal, "Correctness of Concurrency Control and Implications in distributed databases," Proc COMPSAC 79 Conference, Chicago, November 1979.

[BAYE 82] R Bayer, K Elhardt, J. Heigert, and A Reiser, "Dynamic Timestamp Allocation for Transaction in Database Systems," Distributed Databases, H.J Schneider (Ed), North-Holland, pp. 9-20, 1982

[BERN 79] P A. Bernstein, D.W. Shipman, and W.S Wong, "Formal aspects of serializability in database concurrency control," IEEE Tran Software Engineering, Vol SE-5, No 3, pp. 203-216, May 1979.

[BERN 81] P.A Bernstein, N Goodman, "Concurrency control in distributed database systems," ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp 185-221.

[BHAR 82] B Bhargava, "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems," Proc Second Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, June 1982, pp. 19-32

[DIJK 68] Dijkstra, E W. "Co-operating sequential processes," in Programming Languages, F. Genuys (Ed.), Academic Press, New York, 1968

[GRAY 78] Gray, J N "Notes on database operating systems," in Operating Systems An Advanced Course, vol. 60, Lecture Notes in Computer Science, Springer-Verlag, 1978, pp 393-481

[KUNG 81] Kung, H T , and Robinson, J.T. "On optimistic methods for concurrency control," ACM Trans , Database Syst , Vol 6, No. 2, June 1981

[LAMP 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communication of ACM, Vol 21, No 7, July 1978.

[LAMP 76] B. Lampson, and H. Sturgis, "Crash Recovery in a Distributed Data Storege System," Xerox Palo Alto Research Centre, 1976

[REED 78] D.P Reed, "Naming and Synchronization in a Decentralised System," Ph D Thesis, Laboratory for Computer Science, M I.T , Cambridge, September 1978

[SCHL 81] Schlageter, B. "Optimistic Methods for concurrency control in distributed database systems," Proc 1981 Int Conf Very Large Data Bases (IEEE), Cannes, Sept. 1981, pp. 125-130

[SINH 83] M K. Sinha, "Synchronization of Transactions in Distributed Database Systems," Ph D Thesis, National Centre for Software Development & Computing Techniques, TIFR, Bombay, March 1983

[SINH 85] M K. Sinha, P D Nanedikar, and S L. Mehndiratta, "Timestamp Based Optimistic Schemes for Concurrency Control in Distributed Database Systems," Technical Report No 100, NCSDCT, TIFR, Bombay, January 1985

[THOM 79] R H Thomas, "A Solution to the Update Problem for Multiple Copy Database Which Uses Distributed Control," ACM Trans Database Syst , Vol. 4, No. 2, pp. 180-209, June 1979.