BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M11

"The Concurrency Control Mechanism of the Mongrel System Design and Implementation"

by
Sergio A. Nakai

The Concurrency Control Mechanism of the Mongrel System Design and Implementation

Sergio A. Nakai
Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of
Master of Science in the Department of Computer Science at Brown University

May 1993

This research project by Sergio A. Nakai is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degree of Master of Science.

Stan Zdonik

Professor Stan B. Zdonik
Advisor

5/19/93

Date

The Concurrency Control Mechanism of the Mongrel System
Design and Implementation

Sergio A. Nakai
Department of Computer Science
Brown University

May 1993

Abstract

A multidatabase is a distributed database system which creates the illusion of logical integration of heterogeneous local databases without requiring physical database integration. The local database systems may have different logical models and data definition and manipulation languages, and furthermore, may differ in their concurrency control and transaction processing mechanisms [KS91].

The Mongrel system is a multidatabase system based on the Interactions model (described in detail in [No91] and [No92]). This work describes and analyzes the concurrency control mechanism of the Mongrel system. It includes design and implementation problems encountered and the solutions chosen, as well as the reasoning behind the decisions made.

A brief description of the more important aspects of the Interactions model and the architecture of Mongrel are first presented. This is followed by the detailed analysis of the concurrency control mechanism. The appendix includes the design documentation of the relevant modules (all were written in C++).

1. Introduction

The main objective of the concurrency control scheme of a database is to ensure the correct serialization of transactions. The most widely used concurrency control protocols are locking, timestamp, and graph based. In a multidatabase (MDBS), concurrency control is becomes much more complex than in an isolated database management system (DBMS). The protocol of a MDBS has to be sufficiently sophisticated and flexible to be able to synchronize the concurrency control schemes of all the DBMSs that compose the MDBS.

The implementation of the Mongrel system, a multidatabase system designed based on the Interaction model [No91], is currently under way. In this work, the concurrency control mechanism of the Mongrel system is presented and analyzed. Moreover, the design and implementation issues are described in detail.

2. Background

Assuming knowledge of the multidatabase model, this section will only present an overview of the aspects of the Interaction model that are relevant to this study. For a more detailed description of the Interaction model, please refer to the work [No91]. Similarly, the discussion on the architecture of Mongrel focuses on those components of Mongrel that are part of or are related to its concurrency control mechanism.

2.1 Interactions and Transactions

In the Interactions model, an interaction is the top-most unit of work and the task to be accomplished. In comparison to a DBMS, an interaction is analogous to a long-term transaction. For instance, if the Interaction model is applied to the implementation of a travel agency's multidatabase system--where the databases that compose the MDBS are airline, hotel, and car rental companies'--an interaction would represent a trip plan.

Consider a customer's request for a round-trip ticket, hotel, and car reservations. Clearly, this requires accessing at least three databases, and therefore, it needs at least three transactions. In this example, the interaction is the set of the transactions required to completely and successfully service the request of the customer.

Transaction are categorized into two groups, global and local transactions. Local transactions are executed in the a single database and are viewed and treated by the DBMS alike a transaction that is not controlled by the MDBS. Global transactions are sequences of one or more local transactions that are executed in different DBMSs. In the example above, the three reservations requested are handled by three different global transactions. When reserving a car, a global transaction may check several car rental companies' databases for availability and prices. For each database accessed, a local transaction is created to execute the transaction at its particular DBMS. Therefore, an interaction is composed of one or more global transactions, which in turn are composed of one or more local transactions.

In the following figure (fig. 1), the transaction diagram of a common interaction of a travel agency MDBS is shown. The interaction is represented by the trip plan of customer John Doe. This interaction is divided into three global transactions, namely plane, car, and hotel reservations. Each of these global transactions is in turn broken up into a collection of one or more local transactions.

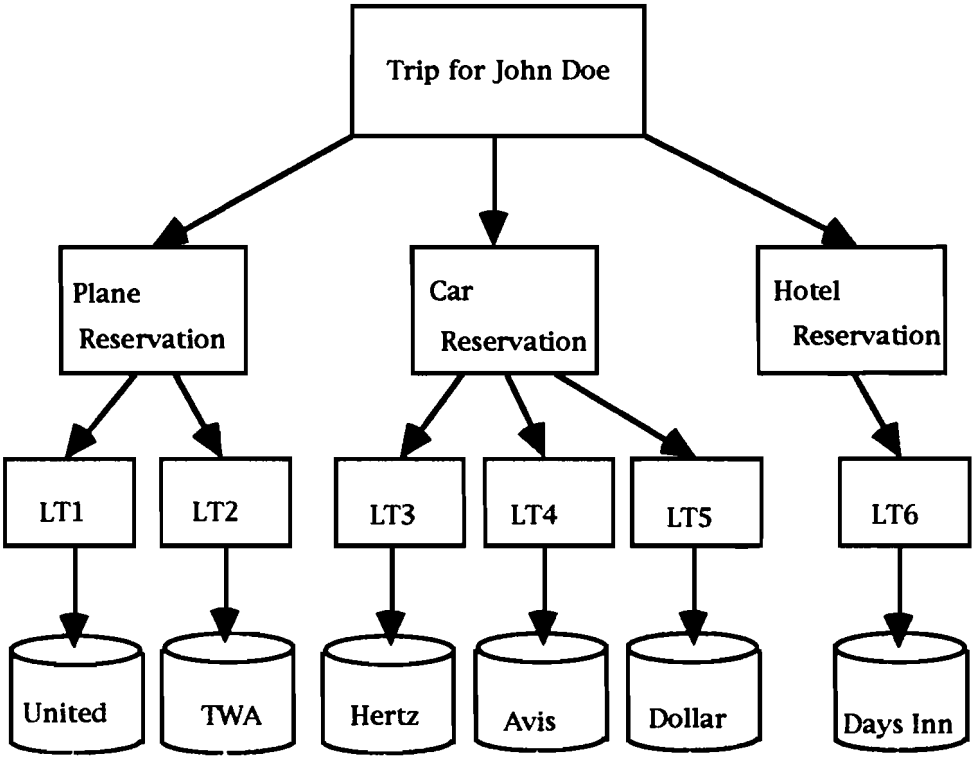


fig 1. Transaction Diagram of a typical Interaction

2.2 Architecture of the Mongrel System

The Mongrel System is divided into three main components: (1) TaSL, the graphical user interface, (2) the Interaction Manager (IM), the core module of the system, and (3) the Agents, the modules (one for each DBMS that forms part of the Mongrel MDBS) that interface the IM with the different DBMSs.

The IM is the core of Mongrel, its central system. It is responsible, among other things, for processing the information entered through TaSL, which usually requires logging data for recovery reasons, updating the data it stores to keep track of the global transactions in execution and the local transactions each of them creates, and naturally, communicating with the proper local databases requesting some service or executing a

local transaction. Clearly, this is a very simplified description of the basic functionalities of the IM. To this study, the most important element of the IM is the Concurrency Control Manager (CCM), which is in charge of enforcing the proper serialization of global transactions.

The main responsibility of the Agents, the interfaces between the IM and the databases, is to manage the local transactions executing at the DBMS associated with it. This involves, among other things, spawning new processes to run the local transactions (a local transaction is handled by one and only one process), keeping track of the serialization order of the local transactions for concurrency control reasons (explained in the next section), and logging information needed to perform recovery procedures. The Serialization Enforcer (SE) is the component of the Agent that performs the concurrency control tasks. The CCM works together with the SEs to ensure proper serialization of the Interactions. The following figures, fig. 2 and fig. 3, illustrate the architecture of the Mongrel MDBS system. The first one depicts the overall system, whereas the latter one shows a detailed description of the Agent.

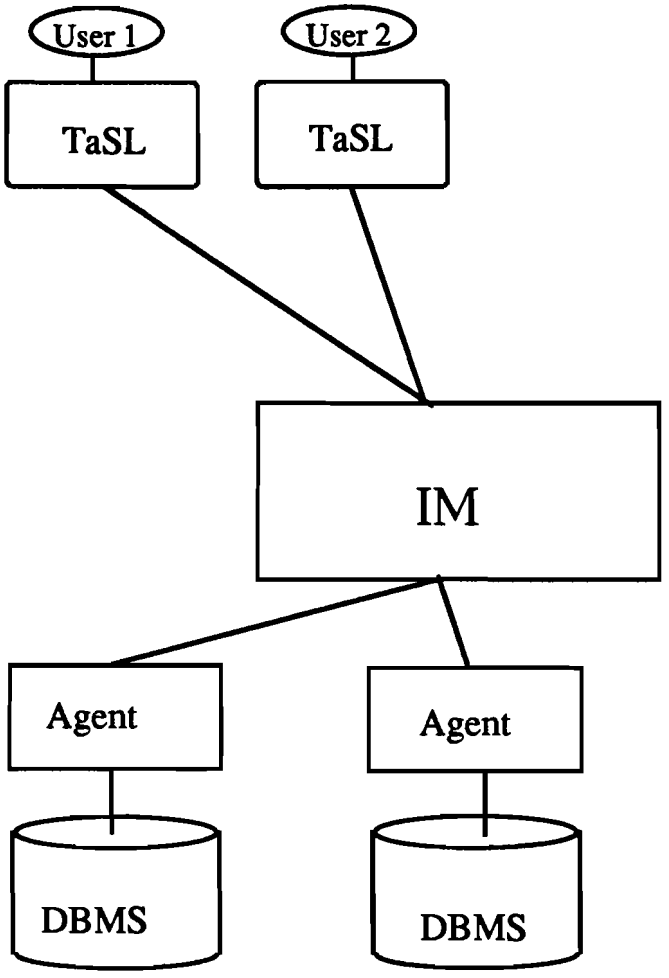


fig. 2: Main Componenets of the Mongrel System

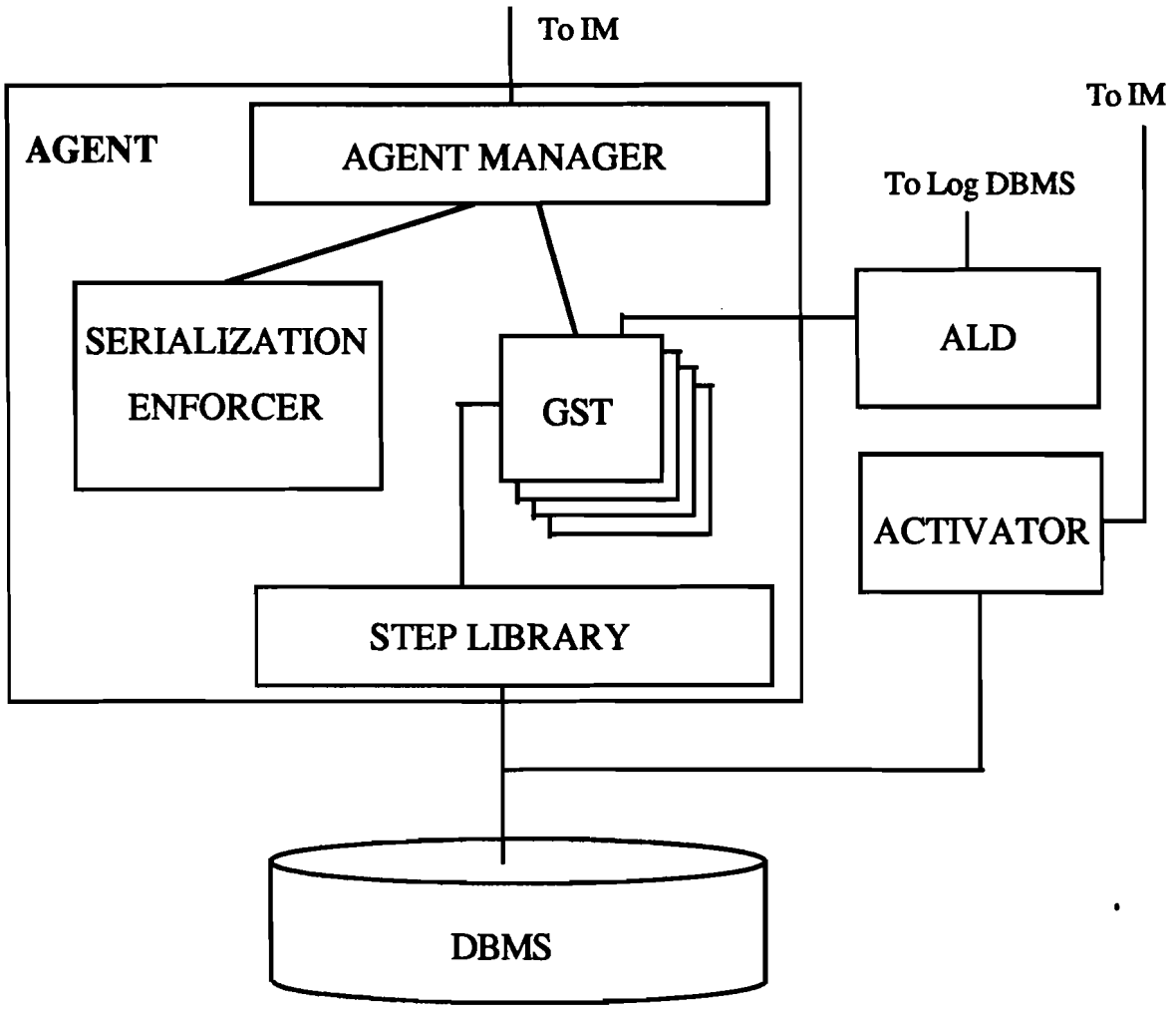


fig. 3: Architecture of the Agent

In fig. 3, all the components that run at each DBMS in the Mongrel system are shown: the Agent and its two side modules, the Agent Logger Daemon (ALD) and the Activator. Both of these sub components are elements of the Recovery and Rollback Mechanism and will not be discussed further in this work. The Agent, on the other hand is the most important piece of the Concurrency Control Mechanism of the Mongrel system. As depicted, it is composed of a Agent Manager (AM), a Serialization Enforcer (SE), a Step Library, and a set of Global Subtransaction objects (GSTs).

The Agent Manager is the interface of the Agent. All messages are first received by the AM which then redirects it to the proper recipient. In addition, it locks and unlocks the Agent (necessary when committing, as explained later) and keeps track of the transactions executing at the associated DBMS. The SE was briefly described above, and will be studied in detail in the next sections. The Step Library contains a collection of functions or steps that perform a specific task in the database. For instance, an airline Agent's Step Library might contain steps such as make_reservation or delete_reservation. The GST objects are processes that handle the local transactions. Each local transaction requires a dedicated process to execute it, thus one GST object is necessary for each transaction running a database.

3. The Concurrency Control Mechanism

In the Interactions model, the basis of the Mongrel system, interactions are non-atomic transactions, whereas global transactions are atomic. To illustrate these characteristics of the interactions, consider two interactions I_a and I_b, where I_a is composed of global transactions GT_{a1} and GT_{a2} and I_b is composed of GT_{b1} and GT_{b2}. Since the interactions are non-atomic, it is valid to interleave the global transactions of I_a and I_b without having to serialize them. Thus, the schedule $<GT_{a1}, GT_{b1}, GT_{b2}, GT_{a2}>$ is correct, even though I_a is not executed atomically. However, since global transactions are atomic, it is not valid to interleave the local transactions of any global transactions. Hence, if GT_{a1} is composed of local transactions LT_{a11} and LT_{a12}, and GT_{b1} is composed of a single local transaction LT_{b11}, the schedule $<LT_{a11}, LT_{b11}, LT_{a12}>$ is not valid since GT_{a1} is not executed atomically.

In order to enforce a correct serialization of global transactions, Theorem 4.1 introduced in [No91] shows that it is sufficient to enforce the serialization of the local transactions according to the serialization order of the global transactions, which we will

call the Global Serialization Order (GSO). In other words, if for any global transactions GT_a and GT_b, where GT_a is before GT_b in the GSO, Theorem 4.1 affirms that if for any local transactions LT_a and LT_b, where LT_a is a local transaction of GT_a and similarly LT_b is of GT_b and LT_a and LT_b are executed in the same local database, then if LT_a is serialized before LT_b in this local database, then GT_a is executed atomically and is serialized before GT_b.

Therefore, Theorem 4.1 tells us that it is enough to enforce the GSO on all the local databases, to ensure the correct serialization of global transactions. Thus, the main objective of the concurrency control mechanism of the Mongrel system is to enforce the GSO in all the databases that form part of the MDBS, which involves serializing all the local transactions according to one unique serialization order, the GSO.

Naturally, the component responsible for computing the GSO is the IM. The IM is aware of all the interactions and their global transactions executing in the system at any point in time, since all Mongrel users have to communicate only with the IM (using TaSL) to initiate and execute interactions and transactions. The CCM and the Agents then work together to enforce the GSO in all the local databases.

The following section discusses the implementation of four different concurrency control protocols. The next section analyzes the three most widely used concurrency control protocols for DBMSs, namely the timestamp, the two-phase lock, and the serialization graph test protocols. The serialization point of a transaction is the point in time when a transaction is serialized (against all the other transactions being executed in the same database). The three schemes are examined to find when each of them serializes a transaction, or in other words, what their serialization points are. The important issue of how commit is handled in the Mongrel system is investigated next. And, finally, a description of how all the components of the concurrency control mechanism work together to accomplish the enforcement of the GSO is presented.

3.1 Enforcing A Global Serialization Order - Four Approaches

The concurrency control mechanism of the Mongrel system did not only require a careful design, where all possible cases had to be thoroughly examined, but also involved some performance considerations. A major performance problem we foresaw is that in some cases, cascading aborts could not be avoided. Four different approaches to enforce a global serialization order were proposed. Two of them were based on the certifier scheme. In [No91], it is maintained that a "certifiers do not check whether or not a transaction conflicts until the transaction commits. Then, using a procedure depending on the type of certifier, [the certifier] decides whether or not the transaction conflicted with other active transactions". In Mongrel, the clear choice of entity that acts as certifier is the IM (i.e. the core system), making its decisions on the information provided by the Agents. The second two approaches are based on less centralized methods. The Agents not only keep information about the active transactions, but have the power to enforce the GSO themselves. Whenever an Agent detects a conflicting action is about to happen or has just happened, it performs a procedure to ensure that the GSO is maintained (e.g. aborts a transaction, reports the problem to the IM, etc.).

Both of the two groups described, certifiers and non-certifiers, are divided depending on when the scheme serializes the transactions. In particular, it can either serialize the transactions when they begin or when they commit. In other words, this sub division is based on whether the GSO is enforced to be the same as the order in which transactions begin or the order in which they commit. Hence, the four cases considered were named Non-Certifier Begin Order, Non-Certifier Commit Order, Certifier Begin Order, and Certifier Commit Order.

One might think that the certifier cases should have worse performance results because in these cases, an entire transaction has to be performed before the IM can decide whether it can commit or not, whereas in the non-certifier cases, problems can be detected earlier on and thus, no time needs to be wasted running a transaction that is not

going to commit in any case. However, this was not the only consideration. It is difficult to predict which of the four approaches would produce the least aborts, which is perhaps the main performance issue. It was decided that the four schemes had to be implemented and then tested against each other.

3.2 Understanding the Concurrency Control Protocols - Analysis of the Timestamp, Two-Phase Lock, and Serialization Graph Testing Schemes

An important characteristic of multidatabase systems is the fact that the databases that are part of it may be completely different from each other. This includes the concurrency control mechanisms of the various databases. In the Mongrel system, the three most widely used protocols, namely timestamp, two-phase lock, and graph-based, were taken into consideration. These mechanisms ensure proper serialization of transactions in very different ways. An important objective of the concurrency control mechanism of the Mongrel system is to synchronize the concurrency control mechanisms of the DBMSs that form part of it.

A correct serialization in an isolated DBMS implies that although the transactions were executed concurrently, the results obtained are the same as if they had been executed one after the other. Proper concurrent execution of transactions, and thus correct serialization, can be achieved if the concurrency control mechanism ensures the transactions are executed atomically.

Concurrency control in MDBS becomes a more complex problem. The MDBS does not have the complete control of the execution of transactions that an isolated DBMS has. Moreover, as was stated above, the concurrency control of Mongrel has to be able to control the order in which the different databases serialize the transactions. And since there is unique GSO that has to be enforced in all databases, the Mongrel's concurrency control mechanism has to synchronize the serialization order in database systems using different concurrency control protocols.

In order to enforce a unique GSO, an analysis of the three concurrency control protocols named above is necessary. In particular, we need to know the exact point in the lifetime of a transaction in which the concurrency control scheme serializes a transaction.

The Timestamp protocol is the perhaps the simplest to analyze. A transaction's serialization order is determined by its timestamp, which is gotten by the transaction when it starts executing. Consider any two transactions T_1 and T_2, with timestamps TO_1 and TO_2 respectively. It is important to note that we are only concerned with transactions that conflict in at least one data item. If the transactions do not conflict, then any schedule is correct, and there was no problem to start with. If T_1 starts before T_2 (i.e. $TO_1 < TO_2$), then if T_2 commits before T_1, we can conclude that T_1 has aborted and rolled back. Therefore, the concurrency control mechanism of the Mongrel system can rely on the timestamp mechanism of a DBMS to enforce the GSO. In order to have a transaction T_1 be serialized before T_2, it is enough to make T_1 get a lower timestamp than T_2. If T_1 commits, then we can be sure that T_2 has not yet committed. If T_1 aborts for any reason, the IM can be notified, and either cascading aborts may occur or a reorganization of the GSO can be performed so that T_1 can be executed again at a later time.

In the Two-Phase Lock protocol, it is not quite as simple to determine the order in which transactions are serialized. In fact, the serialization order of a transaction cannot be determined until it commits. Consider two transactions T1 and T2 which conflict in at least two data items x and y. Let us assume T1 gets a write lock for x, and thus, T2 cannot use x until T1 releases the lock. By the same token, assume T2 get the write lock for y, and thus T1 cannot access y until T2 releases the lock, thus creating deadlock. The deadlock detection mechanism then runs a procedure in which either T1 or T2 is aborted and the other is allowed to continue execution. Since no order list is kept by the concurrency control mechanism, it is impossible direct it to abort one or the other based on some pre-defined ordering. Therefore, we can say that the transactions are serialized

only at commit time; that is, if T1 commits before T2, then T1 is serialized before T2, or vice versa.

In the Timestamp case, a method was found by which the concurrency control mechanism of the DBMS could be used to enforce the GSO. It should be obvious that for the Two-Phase lock protocol, such a method cannot be found so readily. For the Commit Order approaches, both Certifier and Non-Certifier, we can rely on the DBMS's concurrency control mechanism, since it serializes transactions at commit time. Transactions are allowed to executed freely until one of them is ready to commit (i.e. all its instructions have been sent from the IM to the Agent, and from the Agent to database and this transaction is the next in the GSO). If the transaction commits, then it is serialized properly. If it does not commit, we can conclude that it must have been queued waiting for some lock, and thus, it can be aborted. Aborting a local transaction implies that the global transaction it forms part of must be aborted as well as all its other local transactions. Then the aborted global transaction can be moved down in the GSO list and executed again at a later time (aborting a global transaction may involve aborting other global transaction which must be serialized after the aborted transaction).

However, for the Begin Order approaches, the solution is not quite as simple. In the certifier case, a local serialization order list is kept by the Agents. At commit time, the IM requests all the local serialization order lists from the Agents and then compares the unique the GSO to these lists. If any discrepancies are found, the IM can act accordingly. In the non-certifier case, two different algorithms were examined. The first one also involves keeping a local serialization order list. A transaction is inserted into this list when it starts and is allowed to commit if and only if it is the first item in the local list. When a transaction commits or aborts, it is taken off the local list. Since a transaction is added to the list when it starts and the transactions are started by the IM in the order indicated by the GSO, the local list keeps the transactions in the same order as they appear in the GSO. The second approach is based on the forced-local conflict

algorithm and is described in detail in the analysis of the graph-based protocols which follows.

Finally, in the graph-based protocols, the serialization order of transactions is also determined only at commit time. In this protocol, a directed graph is kept, where the "nodes represent transactions and the edges represent conflicting operations in those transactions [connected by the edge], specifically form the earlier operation in the [schedule] to the later operation" [No91]. If at any time, a cycle is formed, the resulting schedule would not be serializable, and thus, the transaction is aborted. The schedule of execution of a database cannot be predicted, and thus, the serialization order of a transaction cannot be determined until it commits.

The graph-based protocols is the most challenging of the three studied, in terms of finding an algorithm to enforce the GSO that takes advantage of the characteristics of the graph-based protocol. The solution chosen is based on the forced-local conflict method introduced in [GRS91]. In this work, a "ticket" algorithm is described. This algorithm enforces a serialization order on transactions by making them conflict on a single data item called the "ticket".

In the Begin Order cases, certifier and non-certifier, transactions are started according to the GSO. The first instruction a transaction attempts to executes is to "take the ticket", which basically means, "touching" the "ticket" data item in the database. This action causes edges to be formed in the graph kept by the concurrency control mechanism--one edge for every node in the graph, since all active transactions must have "taken the ticket". Since the transaction is just starting, a node has to be created first, and since no edge can yet be directed into the new node, no cycle can yet be formed. However, by having all the transactions access this "ticket", we are effectively forcing all transactions to conflict with each other in at least this data item. If no other edges are formed between two nodes, then the two transactions represented by these nodes were non-conflicting transactions. However, if two transactions are conflicting ones, then all

other edges must also go from the node that represents the transaction that "got the ticket" first to the other transaction's node, or otherwise a cycle would be formed. Thus, the GSO is enforced.

For the Commit Order approaches, a very similar procedure is followed. The main difference is the time in which transactions attempt to "take the ticket". In this case, a transaction accesses the "ticket" after all of its instructions have been executed, but before starting the commit process. "Taking the ticket" causes edges to be created only if there are other transactions that have already "taken the ticket", but have not yet finished (i.e. committed or aborted).

In the description of the Two-Phase Lock protocol, it was stated that it used the "ticket" algorithm. This approach uses this algorithm very much alike the graph-based Non-Certifier Begin Order case. When a "ticket is taken", the transaction gets a write-lock on the "ticket" data item. Thus, if a transaction T1 "gets the ticket" before a transaction T2, then either T1 commits before T2 or T1 aborts. A major performance problem with this approach is that concurrency is effectively eliminated, since a transaction's first step is to "get the ticket", and thus, all transactions that do not "get the ticket" will be queued up waiting for the transaction that has the write-lock for the "ticket" data item to release it.

In this section, methods to enforce a serialization order in databases with Timestamp, Two-Phase Lock, and Graph-Based Protocols were presented. Moreover, a description of the "ticket" algorithm, which is based on forced conflicts, was presented.

3.3 The Commit Protocol

When a global transaction commits, to ensure atomicity, all the local transactions that formed part of it must also commit. If at least one does not commit, then all must be aborted and thus, the global transaction also aborts. Two approaches for committing

transactions in distributed databases were considered--global commit before local commit and local commit before global commit. In the former one, the global transaction is first committed and then all of its local transactions attempt to commit. If at least one does not commit, then all the ones that did commit have to be semantically undone and finally, the global transaction has to be undone (which in the Mongrel system would require mostly only changes in the log files). In the second case, the local transactions attempt to commit first. If they are all successful, then the global transaction commits. If at least one fails then the abort procedure is the same as in the first case with the difference that the global transaction does not have to be undone since it was never committed. We opted for the second approach.

The Mongrel system uses a two-phase commit protocol with a few simple variations. When a global transaction reaches its commit point, the IM performs the following steps:

(1) Checks with the CCM, using the service commitCheckGSO. The CCM in turn, depending on whether a non-certifier or certifier mechanism is in use, checks with the participating Agents' Serialization Enforcers (SE) using the services checkSO and requestSO respectively. CheckSO indicates the SE it should verify if the local transaction of the global transaction that wants to commit can commit in the local database. RequestSO just returns a copy of its local serialization order list to CCM. In either case, the CCM validates or invalidates the request depending on the information it receives from all the participating SEs.

(2) If the CCM validates the request, all the local transactions execute a vote procedure. This vote procedure places the databases in the "Prepared" state of a two-phase commit protocol. If any of the databases is unable to go to the "Prepared" state, all other local transactions are aborted, regardless of their vote.

(3) In the case that all votes were successful, the IM sends a commit message to all the participating databases to commit the local transactions.

One of the important features of the Mongrel system is the fact that the DBMSs that form part of it retain their autonomy; in other words, in addition to the Mongrel clients, a DBMS may have other users which are not related to the Mongrel system. In fact, these independent users should not need to know of the Mongrel system to operate in the DBMS, nor should they even notice the fact that the database is part of the Mongrel MDBS (except for performance drop). This was accomplished by enforcing two constraints. First, the independent clients' transactions also reach the DBMS through the Agents. And second, when a commit process is performed, all other transactions must be put on hold (independent and Mongrel transactions). This second constraint was enforced by having the Agent Manager lock the database when a commit process is started (i.e. the vote request is received). The Agent Manager unlocks the database when the transaction either commits or aborts. Since all transactions must go through the Agent (constraint one), and the interface to the Agent is the Agent Manager it was the logical choice to enforce this constraint.

3.4 Architecture of the Concurrency Control Mechanism

The concurrency control mechanism clearly requires a global level component which can produce a global serialization order. The Concurrency Control Manager (CCM) is the component of the IM that is responsible for this task. Moreover, it also acts as the central certifier in those cases. The CCM relies on their Agent's counterparts, the Serialization Enforcers(SE) to service the three functions it provides to the IM. Both the CCM and the SEs work as validators; that is, they are consulted to validate some action before it is performed. The CCM provides services that verify if a global transaction may begin (beginChkGSO), commit (commitChkGSO), or abort (abortChkGSO). To reply one of these requests normally require checking some internal data and/or requesting some information from one or more SEs.

The SE provides 5 services, namely--beginCheck, serializeNow, checkSO, requestSO, and cleanUp. A detailed description of these services follows.

(1) beginCheck, called to verify whether or not a local transaction for a given global transaction can be started at the local database. More than checking, it updates local data structures such as local serialization lists. In only one case it actually has to verify whether the local transaction can be started. In all other cases this function always responds affirmatively. To illustrate a reason for rejection, it is necessary to understand the fact that a local transaction may be serialized before it has even started. In the Non-Certifier Begin Order case, the local transactions have to be serialized when the global transaction starts. However, the CCM does not know which databases are going to be accessed, and thus, cannot determine where local databases are going to be created. Thus, the CCM broadcasts to all DBMSs in the system to serialize a local transaction for the starting global transaction. However, it is not actually necessary to start a transaction. On the contrary, in several cases it would cause serious performance problems. The Non-Certifier Begin Order Timestamp Agent keeps a list of the local transactions. This list is not used to keep the order of the transactions (it can use the Timestamp mechanism for this purpose), but to keep track of the status of the local transactions. Since the SE receives the message serializeNow before the transaction is actually started, it just adds this transaction to the local list and marks the transaction as "Inactive". When the transaction starts, its label is changed to "Active". Thus, to ensure the Begin Order constraint, this function (i.e. beginCheck) verifies that no transaction that follows the one that is trying to start has been labeled "Active", and thus has already started in the local database. If at least one is "Active", the request is rejected.

(2) serializeNow, used only in the non-certifier cases, indicates the SE that it is time to serialize a given local transaction. The action that the SE performs varies depending on whether a Begin or Commit system is in use and also on whether it is a Timestamp,

Two-Phase Lock or Graph-Based DBMS (for a detailed, per-case description, please refer to the appendix).

(3) checkSO, used only in the Non-Certifier Begin Order case. Called to verify if a local transaction can commit. For the two cases in that use the "ticket" algorithm, it is irrelevant (i.e. they always return a positive answer). These two cases are for the Graph-based databases, and one of the Two-Phase Lock approaches (see section 3.2). In the two approaches for Timestamp-based databases, it is also irrelevant. Therefore, it is only useful in the Two-Phase Locking case which uses a local serialization list to enforce the GSO. In this case, it checks whether the transaction that made the request is actually the first in the local list. If it is not, it rejects the request; if it is, it allows it to continue (after a transaction commits or aborts, it is taken off the local list).

(4) requestSO, used only in the certifier cases. If a local serialization list is kept, it sends a copy of this list to the CCM. In several cases, the length of this list is always one when this service is requested because the list is empty before the request, and the first instruction is to append itself to the local list.

(5) cleanUp, called after a transaction finishes, either commits or aborts, to inform the SE that it can get rid of any information that it may be yet holding.

Appendix : Design Documentation - The C++ classes

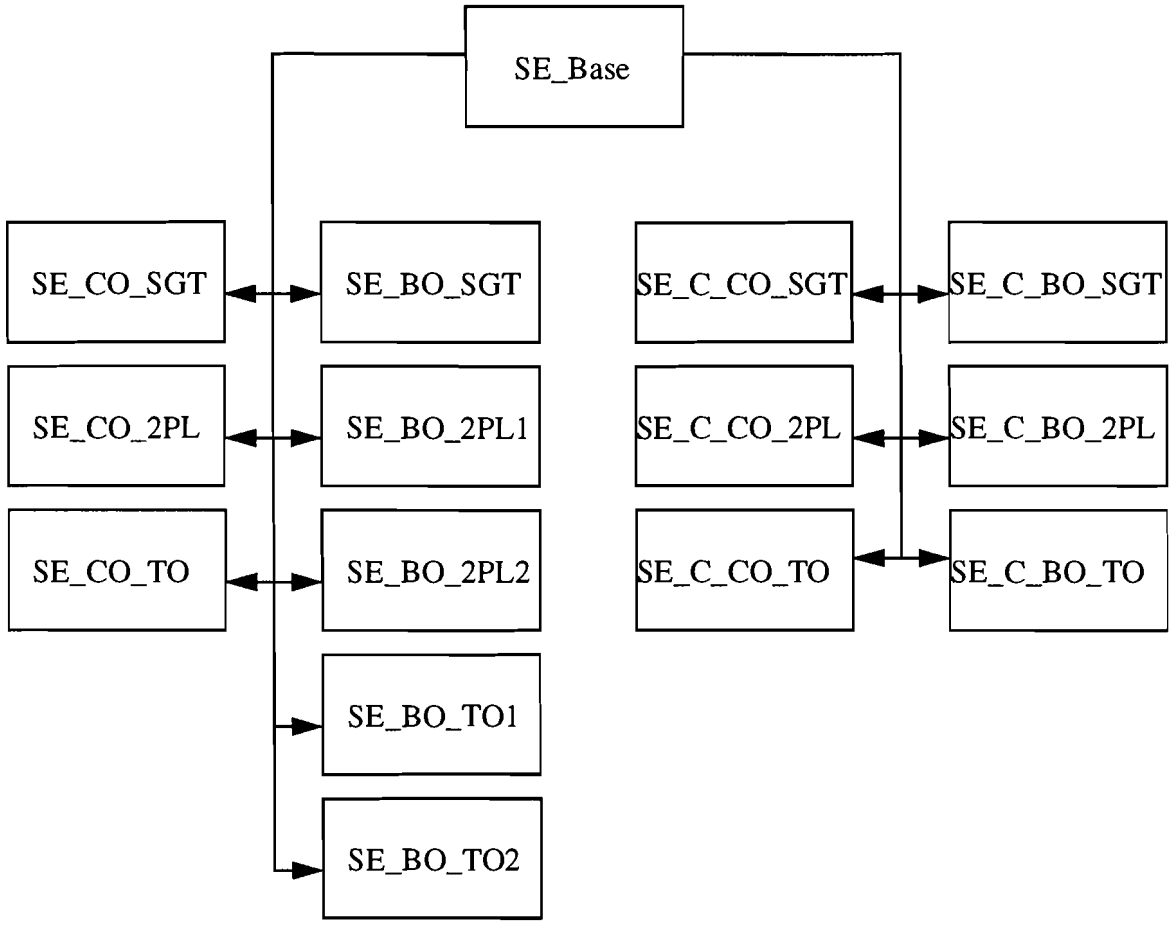
Design Specifications

Two classes conform the core portion of the Agent, namely, the Agent Manager (AM) and the Serialization Enforcer (SE). The AM is the interface of the entire Agent process. Among its functionalities, it is in charge of directing the calls it receives to the proper Agent entities. This requires keeping track of all the different processes in the Agent. Moreover, it starts and finishes GSTs, taking care it is done correctly.

The SE has a more passive role. It is responsible for the proper serialization of the transactions executed at any particular database. In essence, it acts like a validator. When a GST is to be either started or finished, the SE is first consulted (by the AM). It also provides functions that indicate the SE when to serialize a transaction and that check the serialization order, for the non-certifier cases. For the certifier cases, it has a function that returns the Local Serialization List (LSO).

Class Diagram

The AM does not have any superclasses nor subclasses, thus its graphical representation will not be presented. The SE, on the other hand, has a complex structure. The abstract base class, SE_Base, defines the public interface of any implementation of the SE. Since several serialization approaches are being considered and different concurrency control protocols (lock, timestamp, and graph-based) require customized serialization synchronization mechanisms, there are fourteen differen Serialization Enforcers. Each of these is implemented using a different subclass of SE_Base. The following diagram depicts these relationships.



Following, a detailed description of the various AM and SE classes is included. The member functions of these classes are also presented with particular detail.

Description of the Classes

Class AM

- Abstraction:

The AM, or Agent Manager, is the interface of the Agent. All messages to any of the components of the Agent module (SE, GSTs, Activator, and ALD) are first examined by the AM which then redirects this message to the proper recipient. When a new GST process is started for a given global transaction with identifier GTid, the AM stores the RPC client handle to this new process in an internal table. This table is then indexed using the GTid to obtain the corresponding RPC handle. To serve its function as a dispatcher, the AM stores RPC handles to all the other processes that form the Agent Module. All redirections but the ones for the SE need to be made using RPC calls (since the SE is the only other component that is in the same process).

When a global transaction commits, no other transactions should be allowed to make any changes in the databases accessed by the commiting transaction. The AM provides a mechanism to ensure this is not violated. If the vote, the first stage of the global 2-phase-commit, is succesful, the AM sets some internal flags. The AM can then be consulted to check whether the database is in the middle of a commit process or not. These flags are reset by either commiting or aborting the transaction that performed the succesful vote.

- Data Members:

```
CLIENT * IM_handle;          //RPC handle to the IM
CLIENT * ACTIV_handle;       //RPC handle to its Activator
CLIENT * ALD_handle;         //RPC handle to its ALD
CLIENT * PNUM_handle;        //RPC handle to the Program Number Server
char * agent_name;           //Name of the Agent-usually same as LDB
DoubleIntList map_table;     //GTid-GST RPC handle table
char * agent_hostname;       //Hostname where the AM process is running
int agent_prognum;           //Program number of the Agent server process
Boolean in_commit;           //Indicates if a GT is in the commit process
int gt_in_commit;            //Indicates which GT is commiting, if any
SerialMethod serial_case;    //Indicates which serialization method is in use
```

- Private Member Functions:

None.

- Public Member Functions:

```
//Constructor
AM(CLIENT * IM_h, CLIENT * AC_h, CLIENT * PNUM_h, char * ag_name,
        char * hname, int pnum, SerialMethod s_case);

//returns in_commit
Boolean isInCommit();

//checks if in_commit is TRUE and gt_in_commit is GTid
Boolean isGTInCommit(int GTid);

//sets in_commit to TRUE and gt_in_commit to GTid
void setGTInCommit(int GTid);

//if GTid is gt_in_commit then sets in_commit to FALSE and gt_in_commit to -1
void resetGTInCommit(int GTid);

//gets the RPC handle for a given GTid
CLIENT * map(int GTid);

//starts a new GST process and inserts a new entry into the map_table. it first checks
//with the SE if it's OK to start a new GST.
Status beginGST(Oid GTid);

//commits or aborts a transaction, depending on the flag 'C' or 'A'. Either way, kills
//the GST process, removes the entry from the map_table and tells the SE to clean
//its own data storage.
Status finishGST(Oid GTid, char CAflag);

//sends the stepId, argc and argv parameters to the GST process associated with GTid
//if such a GST process does not yet exist, a new one is exec'd (updating the map_table
//and checking with the SE first) and the call is made.
char * doStep(Oid GTid, int stepId, int argc, char * argv);

//it simply calls the Activator's server with the argument pcall
rpc_result * activCall(rpc_command * pcall);
```

- Relationships:

None.

Member Functions of class AM

1. AM::AM

-Semantics:

The constructor of the class AM. It basically gets some parameters and sets the corresponding data members to these values.

-Called by:

- main //the Agent server's main function.

-Calls:

None.

-Parameters:

CLIENT * IM_h	//RPC handle for the IM
CLIENT * AC_h	//RPC handle for its Activator
CLIENT * PNUM_h	//RPC handle for the Program Number Server
char * ag_name	//the name of the agent - usually same to the LDB's
char * hname	//hostname where the AM is running
int pnum	//program number of the Agent's server
SerialMethod s_case	//serialization case in use

-Returns:

None.

2. AM::isInCommit

-Semantics:

Checks whether any transaction is commiting. Does this by checking the internal flag in_commit.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

None.

-Parameters:

None.

-Returns:

The value of data member in_commit (TRUE or FALSE).

3. AM::isGTInCommit

-Semantics:

Checks whether a particular transaction is commiting. Does this by checking the internal flags in_commit and gt_in_commit.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

None.

-Parameters:

int GTid // the GT to be verified

-Returns:

TRUE if in_commit is TRUE and gt_in_commit equals GTid; FALSE otherwise.

4. AM::setGTInCommit

-Semantics:

Called when a vote was sucessful. Sets in_commit to TRUE and gt_in_commit to the global transaction that just voted succesfully.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

None.

-Parameters:

int GTid // the GT to set gt_in_commit to

-Returns:

None.

5. AM::resetGTInCommit

-Semantics:

Called after a transaction has commited or aborted. If gt_in_commit equals the parameter GTid, resets in_commit to FALSE and gt_in_commit to -1.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

None.

-Parameters:

int GTid // specifies the GT that just finished. it is used to check the validity.

-Returns:

None.

6. AM::map

-Semantics:

Searches the map_table for the RPC handle for a given GTid. If no entry for the GTid is found, returns NULL

-Called by:

- agent_call_1 // the Agent's dispatcher
- AM::doStep
- AM::finishGST
- SE_CO_SGT::serializeNow
- SE_BO_SGT::serializeNow
- SE_BO_2PL2::serializeNow
- SE_C_CO_SGT::requestSO
- SE_C_BO_SGT::beginCheck_II

-Calls:

DoubleIntList::findThroughEntry1

-Parameters:

int GTid // the GT for which the RPC handle is requested

-Returns:

The RPC handle is found, NULL otherwise.

7. AM::beginGST

-Semantics:

It starts a new GST process. It first checks that a GST for the given GTid does not yet exist. If it does, does not proceed reporting the error. If it does not exist, it consults the SE to see if a new GST can be started. If so, a new GST process is exec'd, and a new RPC handle is obtained for this process which is stored in the map_table. If the serialization case is case 4, Certifier Begin Order (CERT_BO), a second check with the SE is necessary.

-Called by:

- agent_call_1 // the Agent's dispatcher
- SE_BO_SGT::serializeNow
- SE_BO_2PL2::serializeNow
- SE_BO_TO2::serializeNow

-Calls:

SE::beginCheck	//check with SE if a new GST can be started
SE::beginCheck_II	//only in case 4, second SE validation
DoubleIntList::findThroughEntry1	//verify if GT is in map_table
DoubleIntList::append	//insert new GST handle in map_table
pnum_serv_call_1	//call to prognum server for new GST's pnum
pack_rpc_call	//rpc utilities
unpack_rpc_result	//
clnt_create	//create RPC handle for new GST

-Parameters:

int GTid // the GT for which a new GST is to be started

-Returns:

OK if all goes fine and a new GST process is started. RETRY if the SE does not allow the initialization of a new transaction. NOT_OK, otherwise.

8. AM::finishGST

-Semantics:

It finishes a GST, by either commiting it or aborting it. It first checks whether such GST already exists. If it does not, it reports the error. However, case 2 Non-certifier Begin Order, is an exception. In this case, if the GT does not have a GST, it cleans up and returns OK. Case 2 is special because it uses a broadcast method to serialize the transactions, and thus every agent in the system knows of its existence but not all may have actually started a GST process for it.

Depending on the parameter CAflag which must be either 'C' or 'A', it calls one of the GST functions commitGST or abortGST. It then removes the entry from the map_table and reports the SE that it can remove GTid from its data storage.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

AM::map	//verify if GT is in map_table
SE::cleanUp	//tell SE to clean its data structures
DoubleIntList::removeThroughEntry1	//insert new GST handle in map_table
gst_call_1	//call GST process
pack_rpc_call	//rpc utilities
unpack_rpc_result	//

-Parameters:

int GTid	// the GT of which the GST is to be finished
char CAflag	//'C' for commit, 'A' for abort

-Returns:

OK if all goes fine and the GST is finished. RETRY if an error packing the rpc call occurs. NOT_OK, otherwise.

9. AM::doStep

-Semantics:

It redirects the step call to the proper GST process for execution. First checks if the GTid has a running GST process associated with it. It it does, the step call and its arguments is sent to it. If it does not, it starts a new GST process, following the same steps as in AM::begin GST (please refer to this function for a more detailed description). It then sends the step call and srguments to this newly created GST process.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

AM::map	//verify if GT is in map_table
SE::beginCheck	//check with SE if a new GST can be started
SE::beginCheck_II	//only in case 4, second SE validation
DoubleIntList::findThroughEntry1	//verify if GT is in map_table
DoubleIntList::append	//insert new GST handle in map_table
gst_call_1	//call GST process
pnum_serv_call_1	//call to prognum server for new GST's pnum
pack_rpc_call	//rpc utilities
unpack_rpc_result	//
clnt_create	//create RPC handle for new GST

-Parameters:

Oid GTid	// the GT for which the step is to be executed
int stepId	// the step id number
int argc	// the number of arguments
char * argv	// string containing arguments

-Returns:

The string returned by the step call. NULL, if anything went wrong. Notice that the step call may return NULL.

10. AM::activCall

-Semantics:

This simple functions passes the rpc call down to the Activator associated with it. It does not process the input nor the output.

-Called by:

- agent_call_1 // the Agent's dispatcher

-Calls:

activ_call_1	//call Activator
pack_rpc_result	//rpc utility

-Parameters:

rpc_command * pcall // the rpc call to pass to the Activator

-Returns:

It returns the rpc_result * returned from the RPC call to the Activator. If an error occurs, it returns an rpc_result which status field is set to NOT_OK and with no arguments.

The Serialization Enforcer Classes

Class SE_Base

- Abstraction:

This abstract base class defines the public interface of any SE class, which must be a sub-class of it.

- Data Member:

None

- Private Member Functions:

None.

- Public Member Function:

```
//check if a new subtransaction can be begun, before it is started
virtual Status beginCheck(Oid) = 0;

//after a new subtransaction has been started, check if it was valid
virtual Status beginCheck_II(Oid) = 0;

//tells the SE it is time to serialize a particular transaction
virtual Status serializeNow(Oid) = 0;

//check if the serialization order is valid when commiting
virtual Status checkSO(Oid) = 0;

//returns the LSO list, if any
virtual Status requestSO(Oid) = 0;

//cleans up the LSO list and any other internal structures after a transaction is finished
virtual Status cleanUp(Oid) = 0;
```

-Relationships:

Superclass of : SE_CO_SGT, SE_CO_2PL, SE_CO_TO, SE_BO_SGT, SE_BO_2PL1, SE_BO_2PL2, SE_BO_TO1, SE_BO_TO2, SE_C_CO_SGT, SE_C_CO_2PL, SE_C_CO_TO, SE_C_BO_SGT, SE_C_BO_2PL, SE_C_BO_TO.

Class SE_CO_SGT

- Abstraction:

SE for the Non-Certifier Commit Order for a database using a serialization-graph-test concurrency control protocol.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_CO_2PL

- Abstraction:

SE for the Non-Certifier Commit Order for a database using a two-phase-lock concurrency control protocol.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_CO_TO

- Abstraction:

SE for the Non-Certifier Commit Order for a database using a timestamp-based concurrency control protocol.

- Data Member:

```
//the local SO list
OidList LSO_list;
```

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_BO_SGT

- Abstraction:

SE for the Non-Certifier Begin Order for a database using a serialization-graph-test concurrency control protocol.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_BO_2PL1

- Abstraction:

SE for the Non-Certifier Begin Order for a database using a two-phase-lock concurrency control protocol. This approach uses a LSO list to ensure serialization.

- Data Member:

```
//the local SO list
OidList LSO_list;
```

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_BO_2PL2

- Abstraction:

SE for the Non-Certifier Begin Order for a database using a two-phase-lock concurrency control protocol. This approach uses the ticket algorithm to ensure serialization.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_BO_TO1

- Abstraction:

SE for the Non-Certifier Begin Order for a database using a timestamp-based concurrency control protocol. This approach uses a LSO list to ensure serialization.

- Data Member:

//the local SO list
DoubleIntList LSO_list;

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_BO_TO2

- Abstraction:

SE for the Non-Certifier Begin Order for a database using a timestamp-based concurrency control protocol. This approach uses the timestamp mechanism of the database to ensure serialization.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_CO_SGT

- Abstraction:

SE for the Certifier Commit Order for a database using a serialization-graph-test concurrency control protocol.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_CO_2PL

- Abstraction:

SE for the Certifier Commit Order for a database using a two-phase-lock concurrency control protocol.

- Data Member:

None.

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_CO_TO

- Abstraction:

SE for the Certifier Commit Order for a database using a timestamp-based concurrency control protocol.

- Data Member:

//the local SO list
OidList LSO_list;

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_BO_SGT

- Abstraction:

SE for the Certifier Begin Order for a database using a serialization-graph-test concurrency control protocol.

- Data Member:

```
//the local SO list
OidList LSO_list;
```

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_BO_2PL

- Abstraction:

SE for the Certifier Begin Order for a database using a two-phase-lock concurrency control protocol.

- Data Member:

```
//the local SO list
OidList LSO_list;
```

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Class SE_C_BO_TO

- Abstraction:

SE for the Certifier Begin Order for a database using a timestamp-based concurrency control protocol.

- Data Member:

```
//the local SO list
OidList LSO_list;
```

- Private Member Functions:

None

- Public Member Function:

Same as SE_Base (but non virtual)

-Relationships:

Subclass of: SE_Base

Member Functions of Class SE

The descriptions of the member functions of the SE classes that follows include each the details of the different implementations of the SE.

1. SE::beginCheck

-Semantics:

Checks if a new subtransaction can be begun, before it is started. If a LSO list is kept, it appends the new gt id's to the list.

-Called by:

- agent_call_1 // the Agent's dispatcher
- AM::beginGST
- AM::doStep

-Calls:

SE_CO_TO:	Oidlist::append
SE_BO_TO1:	DoubleIntList::findAfterEntry
	DoubleIntList::changeEntry2
SE_C_CO_TO:	Oidlist::append
SE_C_BO_2PL:	Oidlist::append
SE_C_BO_TO:	Oidlist::append

-Parameters:

Oid GTid; //the GT for which the start of a new transaction is to be validated

-Returns:

SE_CO_SGT :	returns OK
SE_CO_2PL:	returns OK
SE_CO_TO:	returns OK
SE_BO_SGT:	returns OK
SE_BO_2PL1:	returns OK
SE_BO_2PL2:	returns OK
SE_BO_TO1:	if any gt id located after the parameter GTid has already started, returns NOT_OK. returns OK otherwise.
SE_BO_TO2:	returns OK
SE_C_CO_SGT:	returns OK
SE_C_CO_2PL:	returns OK
SE_C_CO_TO:	returns OK
SE_C_BO_SGT:	returns OK
SE_C_BO_2PL:	returns OK
SE_C_BO_TO:	returns OK

2. SE::beginCheck_II

-Semantics:

It checks whether the initialization of a new gst was valid. It is only used in case 4, Certifier Begin Order..

-Called by:

- AM::beginGST
- AM::doStep

-Calls:

SE_C_BO_SGT:	AM::map pack_rpc_call gst_call_1 unpack_rpc_result Oidlist::append

-Parameters:

Oid GTid; //the GT for which the start of a new transaction is to be validated

-Returns:

SE_CO_SGT :	returns BUG
SE_CO_2PL:	returns BUG
SE_CO_TO:	returns BUG
SE_BO_SGT:	returns BUG
SE_BO_2PL1:	returns BUG
SE_BO_2PL2:	returns BUG
SE_BO_TO1:	returns BUG
SE_BO_TO2:	returns BUG
SE_C_CO_SGT:	returns BUG
SE_C_CO_2PL:	returns BUG
SE_C_CO_TO:	returns BUG
SE_C_BO_SGT:	if succesful taking the ticket, returns OK. NOT_OK otherwise.
SE_C_BO_2PL:	returns OK
SE_C_BO_TO:	returns OK

3. SE::serializeNow

-Semantics:

This function is used to notify the SE when to serialize a particular GST. Used only in the Non-Certifier cases 1 and 2.

-Called by:

- agent_call_1

-Calls:

SE_CO_SGT:	AM::map
	pack_rpc_call
	gst_call_1
	unpack_rpc_result
SE_CO_TO:	Oidlist::getFirst
SE_BO_SGT:	AM::beginGST
	AM::map
	pack_rpc_call
	gst_call_1
	unpack_rpc_result
SE_BO_2PL1:	Oidlist::append
SE_BO_2PL2:	AM::beginGST
	AM::map
	pack_rpc_call
	gst_call_1
	unpack_rpc_result
SE_BO_TO1:	Oidlist::append
SE_BO_TO2:	AM::beginGST
SE_C_BO_SGT:	AM::map
	pack_rpc_call
	gst_call_1
	unpack_rpc_result
	Oidlist::append

-Parameters:

Oid GTid;	//the GT that is to be serialized

-Returns:

SE_CO_SGT :	returns NOT_OK, if it does not find it in the map_table. if it has problems packing/unpacking, returns RETRY. Otherwise, it returns the result of the attempt to take the ticket.
SE_CO_2PL:	returns OK
SE_CO_TO:	if GTid is the first in the LSO list, returns OK; NOT_OK, otherwise
SE_BO_SGT:	if AM::beginGST does not return OK, then this is returned. Else, returns NOT_OK, if it does not find it in the map_table. if it has problems packing/unpacking, returns RETRY. Otherwise, it returns the result of the attempt to take the ticket.
SE_BO_2PL1:	returns OK
SE_BO_2PL2:	if AM::beginGST does not return OK, then this is returned. Else, returns NOT_OK, if it does not find it in the map_table. if it has problems packing/unpacking, returns RETRY. Otherwise, it returns the result of the attempt to take the ticket.
SE_BO_TO1:	returns OK
SE_BO_TO2:	returns whatever its call to AM::beginGST returns.
SE_C_CO_SGT:	returns BUG
SE_C_CO_2PL:	returns BUG
SE_C_CO_TO:	returns BUG
SE_C_BO_SGT:	returns BUG
SE_C_BO_2PL:	returns BUG
SE_C_BO_TO:	returns BUG

4. SE::checkSO

-Semantics:

It is only used in case 2, Non-Certifier Begin Order. Before commiting the SE is requested to verify that its LSO does not have any conflicts.

-Called by:

- agent_call_1

-Calls:

SE_C_BO_SGT: Oidlist::getFirst

-Parameters:

Oid GTid; //the GT to be validated

-Returns:

SE_CO_SGT :	returns BUG
SE_CO_2PL:	returns BUG
SE_CO_TO:	returns BUG
SE_BO_SGT:	returns OK
SE_BO_2PL1:	returns OK if GTid is the first in the LSO list; NOT_OK, otherwise
SE_BO_2PL2:	returns OK
SE_BO_TO1:	returns OK
SE_BO_TO2:	returns OK
SE_C_CO_SGT:	returns BUG
SE_C_CO_2PL:	returns BUG
SE_C_CO_TO:	returns BUG
SE_C_BO_SGT:	returns BUG
SE_C_BO_2PL:	returns BUG
SE_C_BO_TO:	returns BUG

5. SE::requestSO

-Semantics:

It is only used in the Certifier cases 3 and 4. Before commiting the SE is requested to check/return the LSO list.

-Called by:

- agent_call_1

-Calls:

SE_C_CO_SGT:	AM::map
	pack_rpc_call
	gst_call_1
	unpack_rpc_result
SE_C_CO_TO:	Oidlist::clear
	Oidlist::getFirst
	Oidlist::append
	Oidlist::getNext
SE_C_BO_SGT:	Oidlist::clear
	Oidlist::getFirst
	Oidlist::append
	Oidlist::getNext
SE_C_BO_2PL:	Oidlist::clear
	Oidlist::getFirst
	Oidlist::append
	Oidlist::getNext
SE_C_BO_TO:	Oidlist::clear
	Oidlist::getFirst
	Oidlist::append
	Oidlist::getNext

-Parameters:

Oid GTid;	//the GT to be validated

-Returns:

SE_CO_SGT :	returns BUG
SE_CO_2PL:	returns BUG
SE_CO_TO:	returns BUG
SE_BO_SGT:	returns BUG
SE_BO_2PL1:	returns BUG
SE_BO_2PL2:	returns BUG
SE_BO_TO1:	returns BUG
SE_BO_TO2:	returns BUG
SE_C_CO_SGT:	returns NOT_OK, if it does not find it in the map_table. if it has problems packing/unpacking, returns RETRY. Otherwise, it returns the result of the attempt to take the ticket. The list is always returned empty.
SE_C_CO_2PL:	returns OK, list empty.
SE_C_CO_TO:	returns OK and a copy of the LSO list
SE_C_BO_SGT:	returns OK and a copy of the LSO list
SE_C_BO_2PL:	returns OK and a copy of the LSO list
SE_C_BO_TO:	returns OK and a copy of the LSO list

6. SE::cleanUp

-Semantics:

Cleans up the LSO list and any other internal structures after a transaction is finished.

-Called by:

- agent_call_1
- AM::finishGST

-Calls:

SE_CO_TO:	Oidlist::remove
SE_BO_2PL1:	Oidlist::remove
SE_BO_TO1:	DoubleIntList::findThroughEntry1
	DoubleIntList::changeEntry2
	DoubleIntList::getFirstEntry2
	DoubleIntList::removeFirst
SE_C_BO_SGT:	Oidlist::getFirst

-Parameters:

Oid GTid;	//the GT that is to be cleaned from the LSO list.

-Returns:

SE_CO_SGT :	returns OK
SE_CO_2PL:	returns OK
SE_CO_TO:	if it does not find GTid in the LSO list, returns BUG; else OK
SE_BO_SGT:	returns OK
SE_BO_2PL1:	if it does not find GTid in the LSO list, returns BUG; else OK
SE_BO_2PL2:	returns OK
SE_BO_TO1:	if it does not find GTid in the LSO list, returns BUG; else OK
SE_BO_TO2:	returns OK
SE_C_CO_SGT:	returns OK
SE_C_CO_2PL:	returns OK
SE_C_CO_TO:	returns OK
SE_C_BO_SGT:	returns OK
SE_C_BO_2PL:	returns OK
SE_C_BO_TO:	returns OK

References

- [No91] Marian H. Nodine, "InterActions: Multidatabase Support for Planning Applications," Technical Report No. CS-91-64, Brown University, December 1991.

- [No92] Marian H. Nodine, "Supporting Long-Running Tasks on an Evolving Multidatabase Using InterActions and Events," Brown University, June 1992.

- [GRS91] Dimitrios Goergakopoulos, Marek Rusinkiewicz, snd Amit Sheth, " On Serializability of Multidatabase Transactions Through Forced Local Conflicts," *Proceedings of the IEEE Seventh International Conference on Data Engineering*, April 1991.

- [MR91] Peter Muth and Thomas C. Rakow, "Atomic Commitment for Integrated Database Systems," *Proceedings of the IEEE Seventh International Conference on Data Engineering*, April 1991.

- [EJK91] Ahmed K. Elmagarmid, Jin Jing, and Won Kim, "Global Commitment in Mutidatabase Systems", Technical Report No. CSD-TR 91-017, Purdue University, March 1991.

- [KS91] Henry F. Korth and Abraham Silberschatz, *Database Systems Concepts*, McGraw-Hill, 1991.