BROWN UNIVERSITY
Department of Computer Science
Master's Project

CS-94-M10

"Transaction Management for Multidatabases (Interactions): Synchronization of Transactions Used on Planning Applications"

by

Renato C. Rocha

Transaction Management for Multidatabases (Interactions): Synchronization of transactions used on planning applications

by

Renato C. Rocha
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.

January, 1994

This research project by Renato C. Rocha is accepted in its present form by the Department of Computer Science at Brown University in partial fulfillment of the requirements for the Degreee of Master of Science.

May 1994

Date: 5/10/94

Stanley B. Zdonik

Stanley B. Zdonik

1- Abstract

In general terms a Multidatabase [8] is a distributed heterogeneous database system. We call it heterogeneous because the system is composed of several local databases, each of which has distinct transaction processing. Furthermore, those local databases (LDBS) need to retain as much autonomy as possible, since they may be developed and maintained by different organizations. Distributed homogeneous databases usually have to deal with data distribution and replication to maintain global consistency of the data. Besides those problems (data distributions & replication), Multidatabases have also to deal with heterogeneity and local database autonomy.

In this piece of work a concurrency control mechanism for Multidatabase System is discussed. The problems, goals and design approach for the implementation of the Interaction [11] transaction processing model are reviewed along with an overview of the Multidatabase System prototype developed at Brown University.

2- Introduction

With the evolution of the distributed database system, application developers faced the problem of users that needed to query and update data from different database management systems (DBMS), each of which may support distinct concurrency control, data model and data management language (DML). One type of application with those characteristics is a planning application [15] which requires access to more than one distinct database in order to achieve a given task.

In addition, such a planning application also requires reactiveness; that is, the system has to react to any changes at those DBMS that might affect the planning task. By react we meant that the system is given another execution path in order to achieve the planning application goal. Obviously the system has to undo the excution of the previous path.

A Multidatabase is a collection of pre-existing heterogeneous databases called local databases (LDBS). A Multidatabase supports applications that simultaneously access more than one local database [4].

The basic features of a Multidatabase are:

- Heterogeneity
- Autonomous local DBMS. (i.e. the Data Model and Transaction Processing of the local database should not be changed)
- Support to global applications that can either read or write on the local databases (in our work the global application is called Interaction)
- Local DBMS distributed transparency.

In addition to the above features, Nodine's [11] work also addresses to reactivity as another important feature of a Multidatabase system.

The ultimate goal of the Multidatabase Transaction Manager is to avoid inconsistency retrievals and to preserve global consistency in the presence of Multidatabase updates. However, achieving this goal is more complicated than simply dealing with data-distribution and replication, as in the homogeneous distributed database systems. This is because we also have to cope with the heterogeneity and autonomy of local databases [5].

3- The Concurrency Control Problem.

Because of the conditions stated earlier (heterogeneity & autonomy), synchronization is not a trivial matter. In a Multidatabase, when we want to atomically commit or abort a global transaction, the subtransactions (the ones that touch the local databases) either have to all commit or all abort following a global decision. We will briefly recall the steps of the most used commit protocol for distributed databases (two-phase commit). First the global transaction sends a prepared message to every local transaction. Those transactions try to execute all instructions and reply to the global transaction with "abort" or "prepare". At this point the local transaction is either in the abort state or in the prepared state. After receiving every reply the global transaction makes its decision (abort/commit) and sends a message to every participating local transaction. The local transactions then either commit (release their locks) or abort (undo their work). One important point in this technique is that when the local transactions are in the ready state they have all their actions finished and all changes are written in the stable storage. Those transactions are just wait-

ing for the global decision. However, since we have no control over the transaction managers of the local databases in the multidatabases, we cannot expect every different local database transaction manager to have a mechanism for waiting for global decisions [12]. Therefore, atomic commitment is one of the problems to be addressed for the synchronization of the global transactions. The global serialization is another important issue for Multidatabases. The serializability of the local schedules is, by itself, not enough to maintain the Multidatabase consistency. The Transaction Manager of the Multidatabase needs to validate the local schedules in a way that makes the subtransactions globally serializable. Furthermore, since the local databases are autonomous, they might have independent local transaction that will also influence the global serialization. That is, the MDB has to deal with conflicts between the subtransactions (direct conflicts [5]), and also with conflicts caused by independent local transactions that create a dependency between two subtransactions indirectly (indirect conflicts [5]). For example, if we have two global transactions GT1 and GT2 that access local databases LDB1 and LDB2, and an independent transaction T1 that accesses LDB1, we could have the following local schedules: at LDB1: GT1->T1->GT2 and at LDB2: GT2->GT1 which are correct local serializable schedules, but which violate the global serialization. (GT1->GT2->GT1).

The problem is that the global transaction manager needs to be aware of the local schedule of every local database without violating the autonomy requirement. Later we will see that even though we can observe the execution order of global transactions, we still need to monitor the independent transactions at the local database.

3.1- Maintaining the Global Serialization

Several proposals have been made to solve the problem of the Concurrency Control in MDBS. Those proposals differ by the degree of autonomy given to local databases, and by the degree of concurrency. Some proposals even reject serializability as the correctness criterion. We shall describe below some of the main proposals from the compilation made by Georgakopoulos, Rusinkiewicz and Sheth [5]:

- Monitor the execution of the global transactions at each LDBS.

Logar and Sheth [9] proposed the use of the commands of the LDBS and the Operating System to check if the local serialization of the global transactions were consistent with the global serialization order of the multidatabase. This approach however, might potentially violate the autonomy of the LDBS.

- Force local conflicts [5]. The basic idea is to require that every global transaction access a particular piece of data (i.e. ticket) in the local database in order to force a local conflict whenever two global transactions access the same database. Hence, by enforcing the local serializability we are also enforcing global serializability.

- Control the submission and execution order of the global transactions [1]. The basic idea is to have a global transaction manager which enforces the rule that if two global transactions are accessing the same LDBS, one global transaction cannot execute until the other is done with the LDBS. However, we still have to consider the potential of indirect conflict caused by an independent transaction. Otherwise, we could have the global transaction violated even though global transaction execution is enforced.

- Prevent two global transactions from executing concurrently at more then one site.

Breitbart, Silberschatz and Thompson [2] proposed a site graph where a cycle is formed every time two global transactions access the same LDBS in two different sites. If we carefully release locks we guarantee global serializability under direct and indirect conflicts, but the degree of concurrency under this proposal is rather low.

- Use a less strict correctness criterion than serializability.

Du and Elmagarmid [3] proposed the concept of "quasi-serializability" which assumes no value dependencies between two local databases. Therefore, the violation of the global serialization caused by indirect conflicts does not generate data inconsistency.

Nodine [11] also has two proposals for the global serialization problem:

- Dictate the global serialization order to the local transaction manager.

This approach has similarities with some of the approaches mentioned above:
As to monitoring the execution of the global transactions, we do monitor them but instead of interfering with the local transaction manager, we mimic the execution of the global transactions at the local database so that we enforce the global serialization based on what the monitored local serialization order is. As to controlling the submission and execution order of the global transactions, Nodine's approach also assumes that global transactions that execute at the same local database conflict. However unlike the Alonso, Molina & Salem approach, the transactions are not prevented from executing, but are aborted if the serialization enforcer "notes" a violation to the global serialization.

- Validate the local serialization order.

This is a more optimistic version of the previous approach. Instead of dictating the global serialization order to the local transaction manager, each local serialization order list from the local databases is reported to the global transaction manager which validates them.

We now, discuss the implementation of those approaches on the Brown University Multidatabase system (Mongrel). A more complete explanation of the serialization enforcement can be found at [14]. The serialization is achieved through the cooperation of two modules. The first module (at the global level) is the concurrency control manager which coordinates the execution of the global transactions. The second module (on the local level) are the agents (more particularly, the serialization enforcers) which coordinate the execution of each local transaction that access the local database.
The first approach - Dictating the global serialization to the local transaction manager- was implemented in the so called normal scheme. In that, the global transaction manager dictates the global serialization order and communicates the correct order to the agents. The responsibility to enforce this order is with the agents. The second approach - Validating the local serialization order was implemented in the so called "certification scheme". In this scheme, the agent manager mimics the serialization order at the local database according to the transaction manager's protocol. The

agents then report the serialization order at the local database to be validated by the global serialization manager.

In order to enforce the global serialization, the agents need to know at which point in time (serialization point) the local concurrency control protocol defines the serialization order. So that, the agent can use mechanisms to synchronize the global serialization point with the serialization point. Our system provides two distinct serialization points:

1) Begin order, that is the global serialization order is the same as the global transaction's begin.

2) Commit order, that is the global serialization order is the same as the global transaction's commit. The Implementation of the agents took into account the most common concurrency control used, like serialization graph test, two phase locking and timestamp. We describe below each of them and how the agents identify at which point the local protocol serialize the local transactions:

The main idea behind the serialization graph test is to construct a graph where every transaction is a vertex. When transaction A read/writes a piece of data X that transaction B wrote/read, or when the transaction A writes some data X that transaction B wrote, then an edge from A to B is created. The serialization will be violated when a cycle is formed. The serialization point with this protocol is not easily predicted since the graph can have edges throughout the lifetime of the transaction, and even after it commits. In order to be able to define the serialization point on the local database we use the same idea of forcing conflicts on the local database proposed by [5]. Since Nodine's approach assumes that all global transactions conflict with each other when accessing the same local database, while using the forced conflict scheme we expect a rather low concurrency since the step's first action is to take the "ticket". That is, the transaction that could not take the ticket have to wait until the transaction which took the ticket is complete.

In the two-phase locking protocol the first phase (growing phase) tries to obtain all the locks that will be needed for the data that will be updated in the transaction. Once this transaction updates the data it releases the lock and no more locks can be granted to this transaction (shrinking phase). As far as the serialization point goes, it is hard to determine exactly when the last lock is granted and, for that matter, when the local transactions have their serialization order defined. We determined the commit time of the transaction to be the moment all locks are released, since at this point the transaction has done all its changes and any other transaction will see those "permanent"

changes. This moment is easily observed by the agent.

The timestamp protocol defines its serialization point at the time the transactions begin, that is when the timestamp is issue. The problem with timestamp protocol is that local transaction manager restarts an aborted transaction automatically. The agent then, will not be notified that the transaction was restarted and this could change the serialization order. The solution for this problem is to use the local forced conflict scheme as we did with the serialization graph test protocol. By doing so, we expect the agent to detect a conflict before the timestamp protocol does, and to issue an abort.

3.2 - Atomic Commitment.

As we pointed out before, atomic commitment of the global transaction is another difficult problem. The main problem is that on standard commit protocols the local transaction managers wait for a global decision. However since we are dealing with heterogeneous and autonomous transaction managers the local transaction managers might not wait for the global decision. For example, on a two-phase commit protocol when the local transactions receive the prepare message, they answer with "ready" (if they executed the task and are ready to commit) or "abort" (if something went wrong). Then the local transactions wait for a global decision, which is sent by the global transactions.

Peter Muth & Thomas Rakow presented two approaches to achieve atomic commit of global transactions. Those approaches differ by the point in time when the global decision is made in the commit protocol. The main proposals are either "local commit before global decision" or "local commit after global decision". Here we will explain the general idea; a more complete explanation can be found in the Muth & Rakow [12] paper.

The basic idea of the "commit after global decision" approach is that after receiving the prepare message, the local transaction executes until its last instruction. It answers with "ready" (if ready to commit) or "abort" (if something goes wrong). The main difference here is that the local transaction is not waiting for the global decision, meaning that the local transaction manager can decide to abort at any time even if it had replied "ready". Therefore, in order to maintain the ato-

micity, we have to provide a way to redo this transaction if a global decision to commit is made. However, care must be taken not to violate the global serialization order. That is, the first local serialization order in which the global decision was made cannot be changed. For example, assume that in a given local database LDB1 we have global sub-transactions GST1 that sends a "ready" message to a global transaction GT1. Further assume that we have another global sub-transaction GST2 that conflicts with GST1 on some piece of data X. The local serialization order then will be; GST1->GST2. However if for any reason GST1 aborts after the global decision was made and GST2 reads or writes data X, then by redoing GST1 we are going to change the local serialization order to GST2-> GST1. This would conflict with the global serialization order creating a cycle. The solution is to maintain a redo-log for the repeated execution of the local transactions. The redo-log contains the actions of the global transactions.

In the "commit before global decision" approach, the local transactions go all the way to commit or abort once the "prepare" message is received, and they reply to the global transaction with the appropriate answer ("abort" or "commit"). If the global decision is "abort", then a message is sent to all local transactions to abort. If on the other hand, the global decision is "commit", no additional message is sent to the local transactions. In this case, an undo procedure for the local transactions must be provided.

In order to achieve an atomic commitment we use the two-phase commit protocol following the "local commit before global decision" algorithm defined in [12]. This technique seems to be more optimistic and less complex than the alternative "local commit after global decision". That is, while the "local commit after global decision" approach requires a redo procedure and a redo-log, the "local commit before global decision" only requires an undo procedure. Furthermore, the second approach also would require less communication. We assume that the local transactions ensure transaction atomicity.

4. The Interaction Model

The Interaction model proposed by Marian Nodine [11] is a two-level open-nested transaction

model, which was conceived to support planning applications on Multidatabases. The model has some similarities to Sagas [6] and ConTract [17].

The similarity to Sagas consists in the way the long term transactions are broken into smaller transactions, and in the idea of having compensating transactions to undo the effects of any transactions that have been committed. However, the relationships between those smaller transactions are more complex in the Interaction Model. As an open-nested transaction model, the Interaction has its global transactions defined as a partial order since those transactions can execute concurrently. Furthermore, since the Interaction model is planning application oriented two kinds of dependencies are established between global transactions:

1- "Execution dependency" [11]. This is the dependency generated by the order in which the open-nested transaction executes its global transactions. Using the well known travel example, an "execution dependency" is the dependency between a global transaction which makes a flight reservation, and a global transaction which makes a rental car reservation at the destination of the flight. The execution of the second global transaction depends upon the successful completion of the first.

2- "State dependency" [11]. In order to carry out planning applications, the Interactions keep some "internal variables" (or "state variables") which are used by the global transactions. Whenever two global transactions conflict on an "internal variable" a dependency is formed. As an example of a "state dependency" we take the same two global transactions mentioned earlier, (i.e. flight and car reservations), and add the condition that they read a variable budget. The execution of the rental car reservation then depends upon the available budget (for example- if there is no money left after buying the plane ticket we cannot rent a car).

The ConTract model relates to the Interaction model in terms of structure. Like the Interaction model, ConTract consists of a group of atomic blocks of work which map into transaction on the database. Also like the Interaction model, ConTract has conflict handling, although of a different kind. While the Interaction model undo the transaction which caused the conflict, ConTract either prevents the conflict operation, or adjust the data. This second method of conflict handling, called "exit invariant", is problematic if incorporated into a multidatabase system. This is because independent transactions on local databases could potentially have their work lost as part of an "exit

invariant" procedure. It is clear that a local user should not have his/her work interfered with because of a multidatabase transaction. The autonomy of the local databases would thus be disturbed by "exit invariant" conflict handling.

In general terms, there are two basic proposals for integrating the local database information into the Multidatabase: 1) The restricted Model [7] where the global application is given a choice of pre-determined tasks that can be performed at the local database. 2) The unrestricted Model which allows arbitrary tasks to be performed at the local database by the global application. For that matter the compensation procedures have to be defined along with the transaction. The Interaction Model uses the step approach [16]. A "Step" is an atomic procedure that can access a local database in order to achieve some unit of work for a global transaction. The step approach combines some of the characteristics of the two proposals above. Like the restricted model, the step approach also provides a pre-determined set of procedures that can be executed at the local database. Furthermore, some of the flexibility of the unrestricted model is also incorporated into the step approach since the steps can be grouped into atomic local transactions. The steps are statically associated with the compensation steps, which are capable of undoing the effects of the steps. By recording the sequence of compensating steps as the transaction progresses executing steps, the recovery control is able to perform the necessary recovery procedures with no additional information.

4.1- Interaction requirements

Usually a transaction is required to have the following properties: 1) atomicity, 2) consistency, 3)isolation and 4) durability. However, in the Interaction Model, the Interaction, as a long term transaction, relaxes the atomicity and isolation properties. The full atomicity property is replaced by "semantic atomicity". Because a particular piece of information (i.e. state variables) can be used during the execution of more than one global transaction, the serializability is not enough to achieve correctness. That is, the "state dependency" which exists between global transactions must also be considered too for the correctness of the Interactions. This can be derived from Nodine's definition for Interaction correctness [10]: "An Interaction history is correct if its atomic

blocks (global transactions or single steps) are executed serializably, and each InterAction manages its information consistently".

In order to enforce serializability of the global transactions, considering also the effect of the independent local transactions ("indirect conflicts" [5]), Nodine showed on theorem 4.1{[10] p.27} that two conditions were necessary:

1- The local sub-transactions serialization order on each database has to be consistent with some unique global serialization order

2- All sub-transactions that belong to the same global transaction have to commit atomically

4.2- Interaction Features

We enumerate below some of the main features of the Interaction Model.

- *Non Atomicity.* Because the Interaction is a long-term transaction, atomicity is released. However even if the Interaction does not run completely, the system has a way of leaving the Interaction in a state that is semantically equivalent to some state it would have reached if it had not run at all [11]. We achieve this "semantic atomicity" by semantically undoing the undesirable effects of the global transactions or single steps. The granularity of the compensation is at the level of the steps. A more comprehensive explanation of the recovery issues can be found in Nodine and Zdonik work [15].

- *Persistence.* The Interaction can run for an indeterminate period of time. During this time the Interaction can be Active or Inactive. The Interaction keeps all its variables persistent to be able to resume execution in case of a system crash or a change of status from inactive to active.

- *Interactiveness.* The planning applications require access to the data for long periods of time. That might lead to situations in which the application developer needs to interact with the system as the interaction progresses.

- *Reactivity.* One of its novel contributions to the Multidatabase world is reactiveness. Reactiveness allows the Interaction to take the appropriate action to regain consistency in case some operation was performed at the local database that conflict with the Interaction's work.

- *Flexibility.* This is another feature which stems from the planning characteristic of the application. The language for specification of the Multidatabase application allows the application developer to have an Interaction which executes global subtransactions under a conditional flow.

4.3- Interaction structure

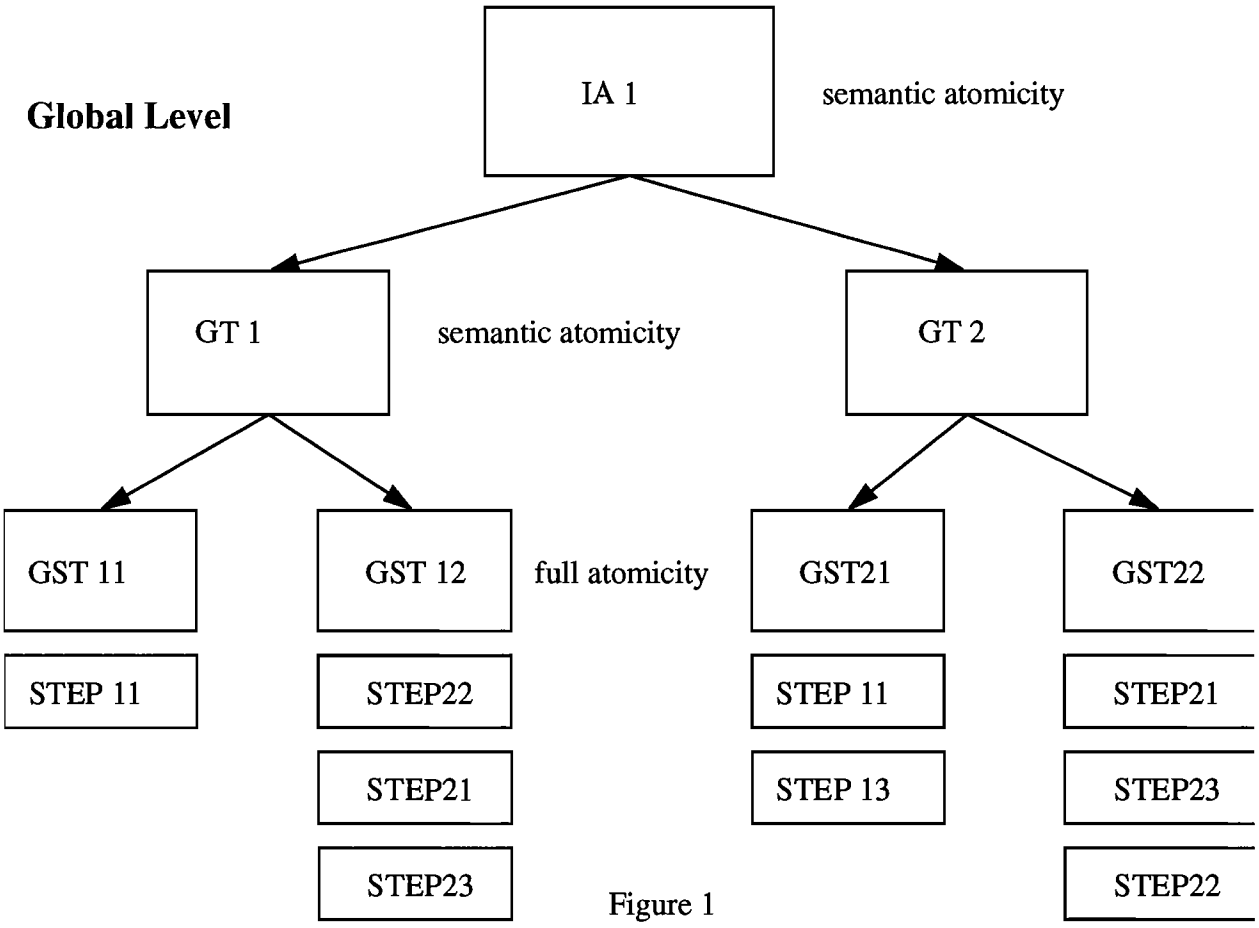

Figure 1

Figure 1 shows the Interaction structure at the global level. IA1 represents an interaction (long-term transaction) compound by two global transactions GT1 and GT2 which can execute concur-

rently and cooperate to achieve some task. The global transactions have a set of global subtransactions (i.e GST11, GST12, GST21 & GST22) which perform some work on a given local database. The global subtransaction is constructed by a group of steps which are a preset collection of atomic local transactions defined at the step library. Each step has a compensating step associated with it that knows how to undo the effects caused by the step. As the transactions progress in the multidatabase, the recovery control logs information about the interdependency among the global transactions (at the global level) as well as the sequence of compensating steps for the steps running at the local transactions. Therefore, when needed, the recovery control knows how to roll back a global transaction by executing another global transaction formed by group of compensating steps. From the point of view of the multidatabase the global transaction issued by the recovery control follows the same commit protocol as an ordinary transaction. That's why we say that the global transactions are semantically atomic, because instead of cancelling the effects of the aborted transaction we issue another transaction to compensate those effects, leaving the data in a state semantically equivalent to the state prior to the aborted transaction run.

Another important characteristic is that the interactions prevent global subtransaction of the same global transaction from accessing the same local database. That is because we want to ensure global atomicity by preventing steps (from the same global transaction) from accessing globally uncommitted data. For example, assume that we have GST11 that calls step21 and GST12 that calls step22 and step23. Further assume that step21 and step22 conflicts on some data X and the local serialization is step22 > step21 > step 23. Although step21 is accessing locally committed data, since GST11 and GST12 belong to the same global transaction, data X is still globally uncommitted.

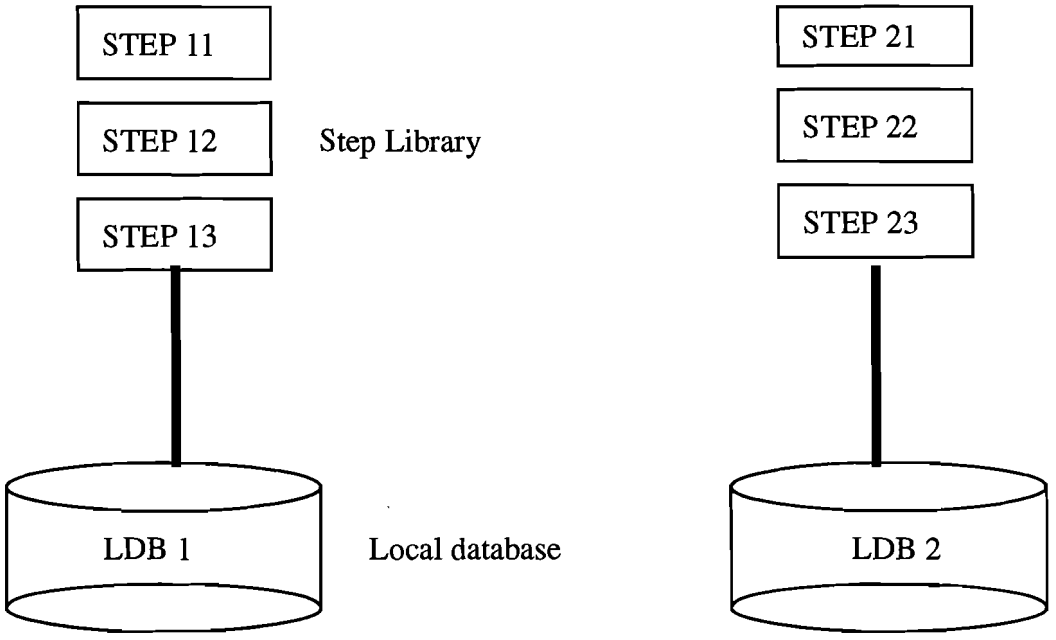

Figure 2

Figure 2 shows a set of steps that can be provided to access the local database. The agents define an interface that encapsulates the data in the database by the step library. From the point of view of the multidatabase these steps are the only way to get information from the local database. This restriction allows the multidatabase to abstract from the data manipulation issues of the local transaction manager. Each local database provides a customized set of steps. This information is made available to the application developer by the TASL module (see section 5).

The Interaction model also defines "events" and "weak conflicts". An "event" signals whenever some update external to the Interaction occurs on some specific data that causes a violation of some condition. We can use an event as a flow control of the Interaction as well (i.e. delaying the execution of the Interaction until some event occurred).

A "weak-conflict" defines which condition should be monitored, the event to be signaled and the set of operations that should be executed in case this condition is violated. We shall discuss next the modules of the system which implement the Interaction module.

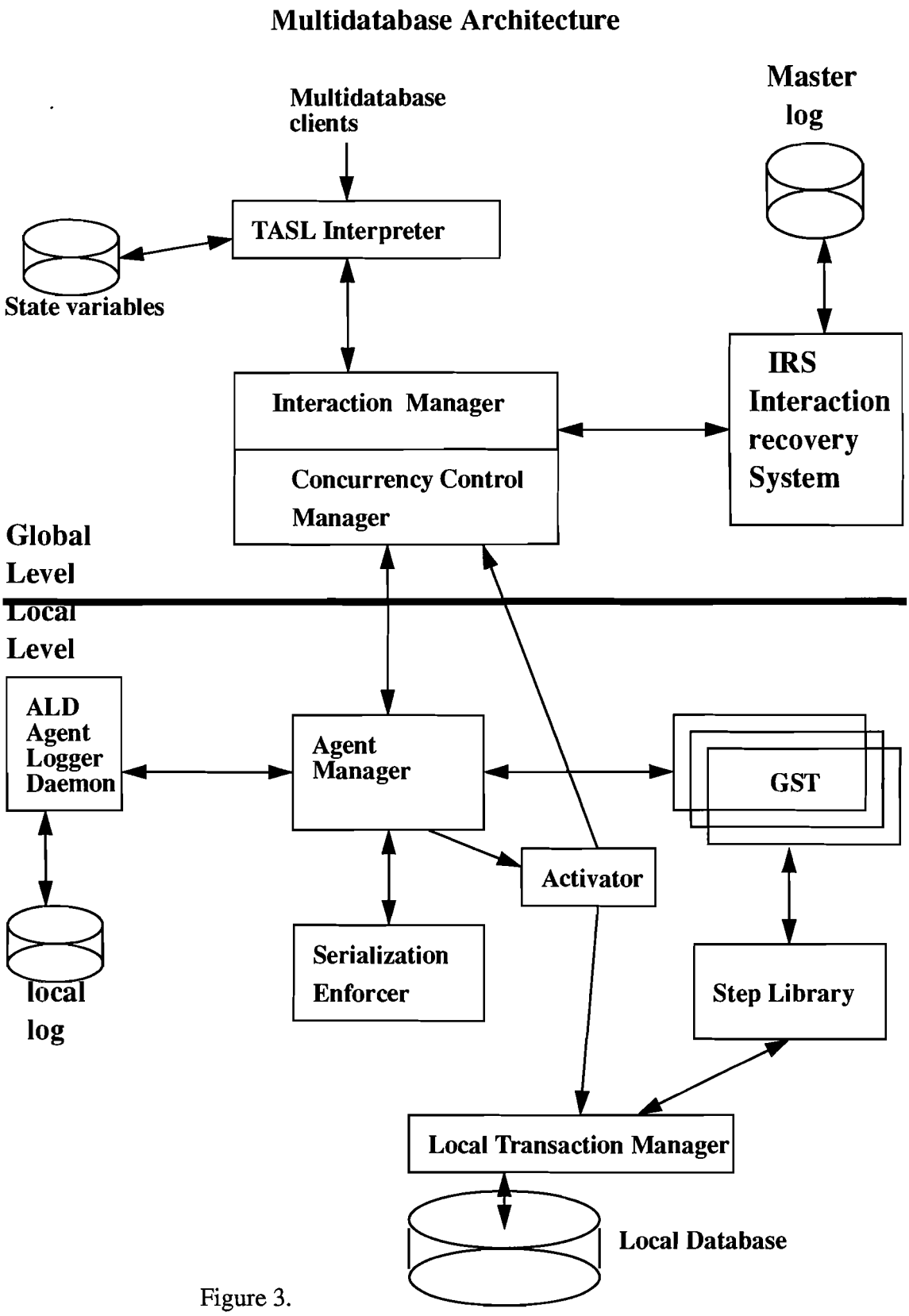
Multidatabase Architecture
Multidatabase clients
Master log
TASL Interpreter
State variables
Interaction Manager
Concurrency Control Manager
IRS Interaction recovery System
Global Level
Local Level
ALD Agent Logger Daemon
Agent Manager
GST
Activator
local log
Serialization Enforcer
Step Library
Local Transaction Manager
Local Database

Figure 3.

5 - Design Overview

The structure of our system is shown on figure 3. We divided the figure in two parts to distinguish the global from the local components. We shall briefly describe each of the modules.

5.1- TASL Interpreter

This module provides an X11 user interface and a specification language to write the Interaction application. The Tasl module is the one which drives the system and, ultimately, supports the reactiveness of the system. It is Tasl's responsibility to maintain the consistency of the data (state variable) due to state dependencies. Tasl provides the application developer with the set of services (Steps) which can be executed in each of the local databases. There is one Tasl shell for every Interaction.

5.2- Interaction Recovery System

The Interaction Recovery System logs the dependency tree of execution, and the order in which the global transactions were executed. When a transaction needs to be undone the recovery system figures the order in which the "compensating steps" have to be performed.
The Recovery System starts a global transaction in order to compensate the effects of some global transaction.

5.3 - Interaction Manager

The Interaction Manager (IM), as its name says, manages the Interaction throughout the lifetime of an Interaction. The IM handles all the communication between the agents, Tasl and recovery system. It is also the Interaction Manager who requests the local agents to monitor the occurrence of a determined event at the local database. In addition, the Interaction Manager receives a signal from the agents notifying the occurrence of a particular event
The IM creates an object (IA) for every Interaction started, which contains information about the

Interaction and a list of global transaction running under this Interaction. Each IA is associated with one Tasl shell, and will only service and communicate with this Tasl shell which requested the creation of the IA. Every time a global transaction begins, a global transaction object is created containing information about every global subtransaction started and the correspondent local database where the global subtransaction is running at. The IM starts up the concurrency control manager which takes care of the synchronization of the global transaction.

5.4 - Concurrency Control Manager

The concurrency control manager synchronizes the global transaction started by the distinct Interactions. As we presented before, there are two basic schemes for achieving global serialization: normal and certification schemes. Furthermore we also presented two mechanisms which define the global serialization point: The "begin" or "commit" order. All combinations of those schemes and mechanisms have advantages and drawbacks. It is not clear which of them has the best performance on average. Therefore, we decide to implement all cases in order to compare them.

There are two modules that cooperate to achieve global serialization in our Multidatabase system: 1- Concurrency Control Manager, 2- Serialization Enforcer. The first is responsible for dictating (normal scheme) or validating (certification scheme) the global serialization order. The second, is responsible for ensuring that the global serialization order has been followed at the local databases. The serialization enforcers mimic the behavior of the local concurrency control and can determine what is the serialization order. Figure 06 shows the concurrency control manager and its interface with the rest of the interaction manager. We create a uniform interface for the concurrency control manager, and by using inheritance we can chose at runtime which scheme (normal or certification) and serialization point (begin or commit time) are to be used. For that matter there are four cases to be considered: 1) normal scheme with serialization point at begin time (CCM_BO), 2) normal scheme with serialization point at commit time (CCM_CO), 3) Certification scheme with serialization point at begin time (CCM_C_BO), and 4) Certification scheme with serialization point at commit time (CCM_C_CO) figure 07.

The concurrency control provides three basic services that in cooperation with the serialization

enforcer ensure proper synchronization among the global serializations. We describe below each of the services.

1- beginChkGso - called by the time we start a global transaction. If the second case was chosen (CCM_BO), the CCM dictates the serialization order and passes it to the serialization enforcer. Otherwise nothing has to be done by this procedure but return OK.

2- CommitChkGso - called by the time we commit a global transaction. It checks if the global transaction is prepared to follow the serialization order to commit. In the normal schemes it dictate the order and wait for the replies of the serialization enforcer to check if the order was followed. In the certification schemes it requests the local serialization order from the local databases to be checked.

3- AbortChkGso - called to abort a non committed global transaction in order to remove the given global transaction from the serialization order.

The Concurrency Control Manager is, also, responsible for keeping track of the handlers to communicate with each of the agents at the local databases. The CCM provides therefore distribution transparency.

5.5 - Agent Manager

The Agent Manager, basically, interfaces with the global level encapsulating the local database data model and transaction processing algorithms. The agent channels all requests from the global level forwarding to the proper modules to be handled. The modules that the agent has to communicate with are: 1) serialization enforcer in order to synchronize the global subtransactions. 2) activator to request an event set. 3) agent logger daemon to log what has been done in the local database and to get the compensate steps to reverse the effects of an global subtransaction. The Agent manager, also, starts up a server (GST) for every global subtransaction that needs to execute in the local database. This GST manages the global subtransaction, and evokes the proce-

dures to be performed at the local database from the step library.

5.6- Activator

The activator sets an event which should be monitored at the local database. The activator does this by polling the local database so often to check if some particular data update (that is an event) has occurred. When this event occurs the activator signals to the IM module. Other options were also considered to achieve the activator task, but the one just described is simpler and provides great deal of local database autonomy.

5.7- The Step Library

The step library has a set of procedures that can be performed by the local transaction processing. Each local database has its own step library tailored for it. The step is the finer granularity of the Interactions. For each step there is a compensate step associate with it that reverse its effect.

6- Design Issues

The System just described, was implemented in C++, on top of UNIX™, following the Client/ Server architecture. We used ObjectStore™ DBMS to store for each Interaction, its entire definition, its execution state, and the state of its variables.

Some issues remain to be considered in the attempt to increase performance. These are asynchronous remote procedures calls, and threads usage. Furthermore, the original idea was to run experiments to compare the performance of the four different cases implemented in the concurrency control manager, so we could decide which one is more suitable.

7- Conclusion

Multidatabases meet the needs of applications which require to query and update data from distinct DBMS simultaneously. The most important features are heterogeneity and autonomy.

Those same features are also the ones which cause most of the problems faced on the implementation of a multidatabase. In particular on issues like global serialization and atomic commitment. The Interaction Model [11] follows a two level open-nested transaction model. The top level (Interaction) is defined as a sequence of atomic tasks (i.e. global transaction). The global transaction is composed by a set of atomic operations on the local databases called steps. We recall that Interactions and global transactions are semantically atomic. That is by compensating the effect of the operations, the data will be left in a state that is semantically equivalent to the state before the operations have ran.

The approach taken to solve the atomic commitment is the "local commit before global decision" described in [5]. The Interaction model proposes two schemes for ensuring serialization. In the first (normal scheme) the global transaction manager dictates the serialization order to the local agents which make sure the order is followed. In the second (certification scheme) the global transaction request the local serialization order list from all local agents to be validated.

The Interaction model introduces two new features on the multidatabases systems. The first is reactiveness. The second is the concept of the step library at each local database. The step concept is a mix of the two approaches for integrating local databases (restricted and unrestricted model). The steps differ from the restricted model in that we can construct a local transaction based on the set of steps available for the local database. furthermore, the step also differ from the unrestricted model because only these set of steps can access the local database.

We now, present the top level design for the transaction manager of our multidatabase.

8- Appendix: Design Documentation

I - Class Diagram

I.1- Interaction Manager

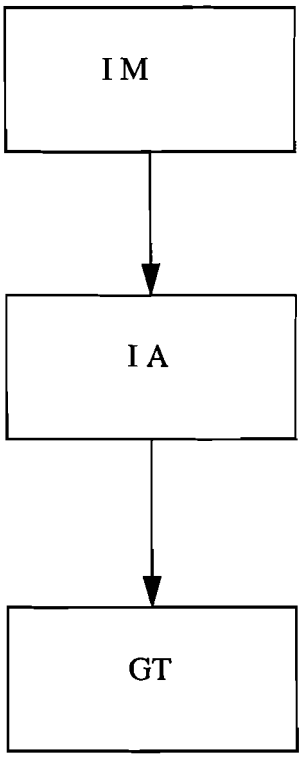

Figure 4.

Figure 4 shows the class diagram of the Interaction Manager. The class IM abstracts the data necessary to keep track of all Interactions on the system. Every time a Interaction begins, an instance of class IA is constructed and its address put into a linked list. There is only one instance of the IM on the system. The class IA has the address of every transaction started by the Interactions. the Interaction class also keeps a handle to communicate with the TASL shell which requested the Interactions' start. The global transaction class (GT) maintain the database's names of the local databases that are being query or updated by the global transactions.

I.2- Concurrency Control Manager

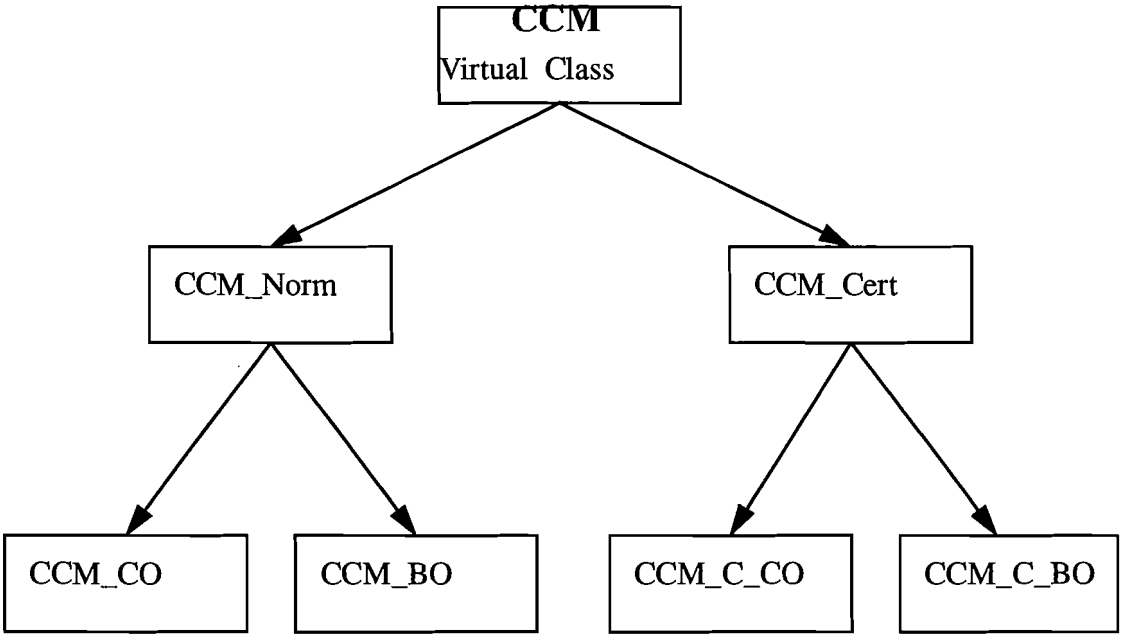

Figure 5.

The class CCM abstracts the global serialization order list which represents the serialization order that has to be followed by the global transactions. It also keeps a list of all local databases and handles available at the system. Figure 6 and 7 show the methods of every class and its relationship.

GLobal Level: Interaction Manager & Concurrency Control

Class Diagram - Base Classes

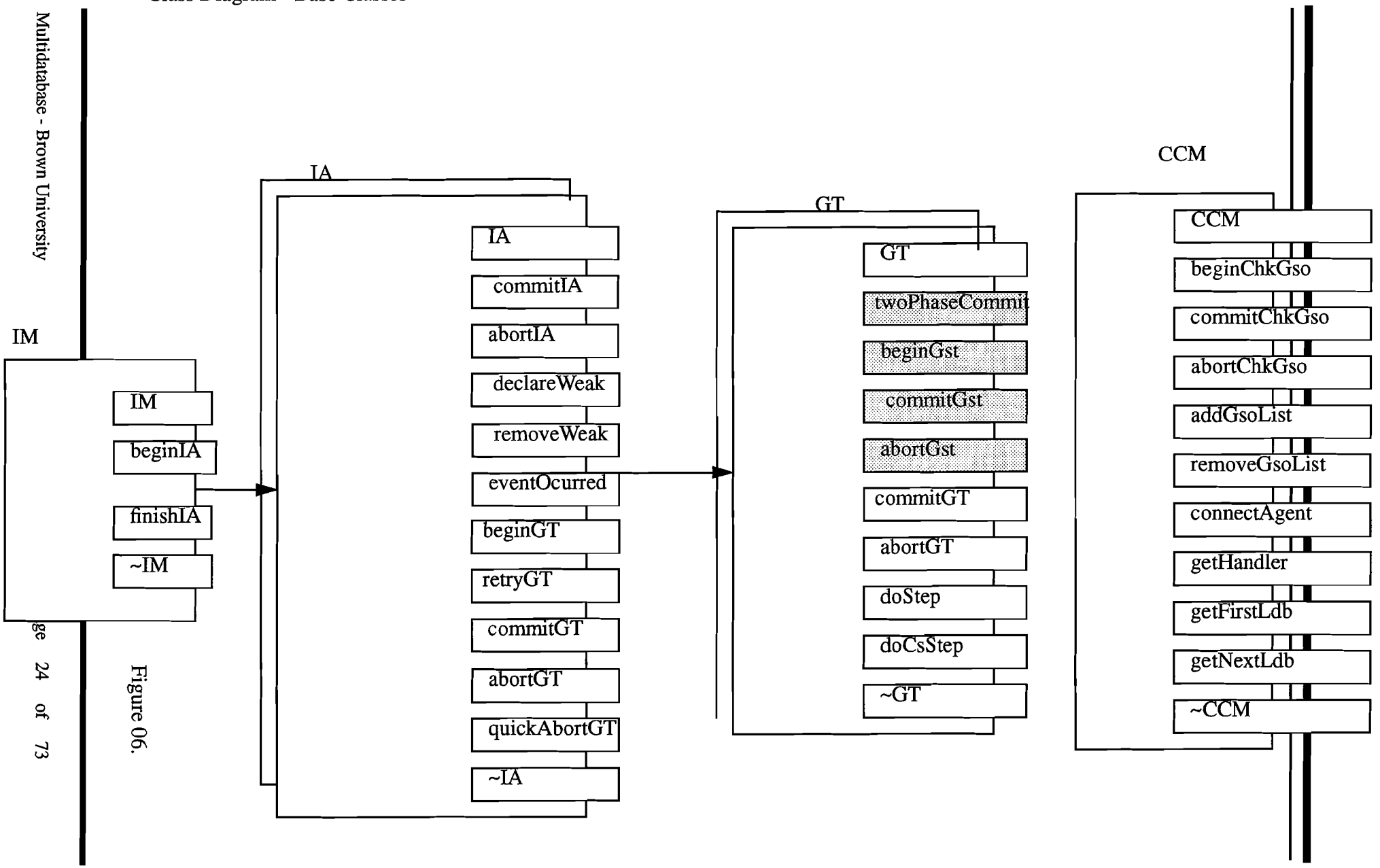

Figure 06.

private member functions

Note: For information about parameters please consult header files.

Class Diagram: Derivate Classes

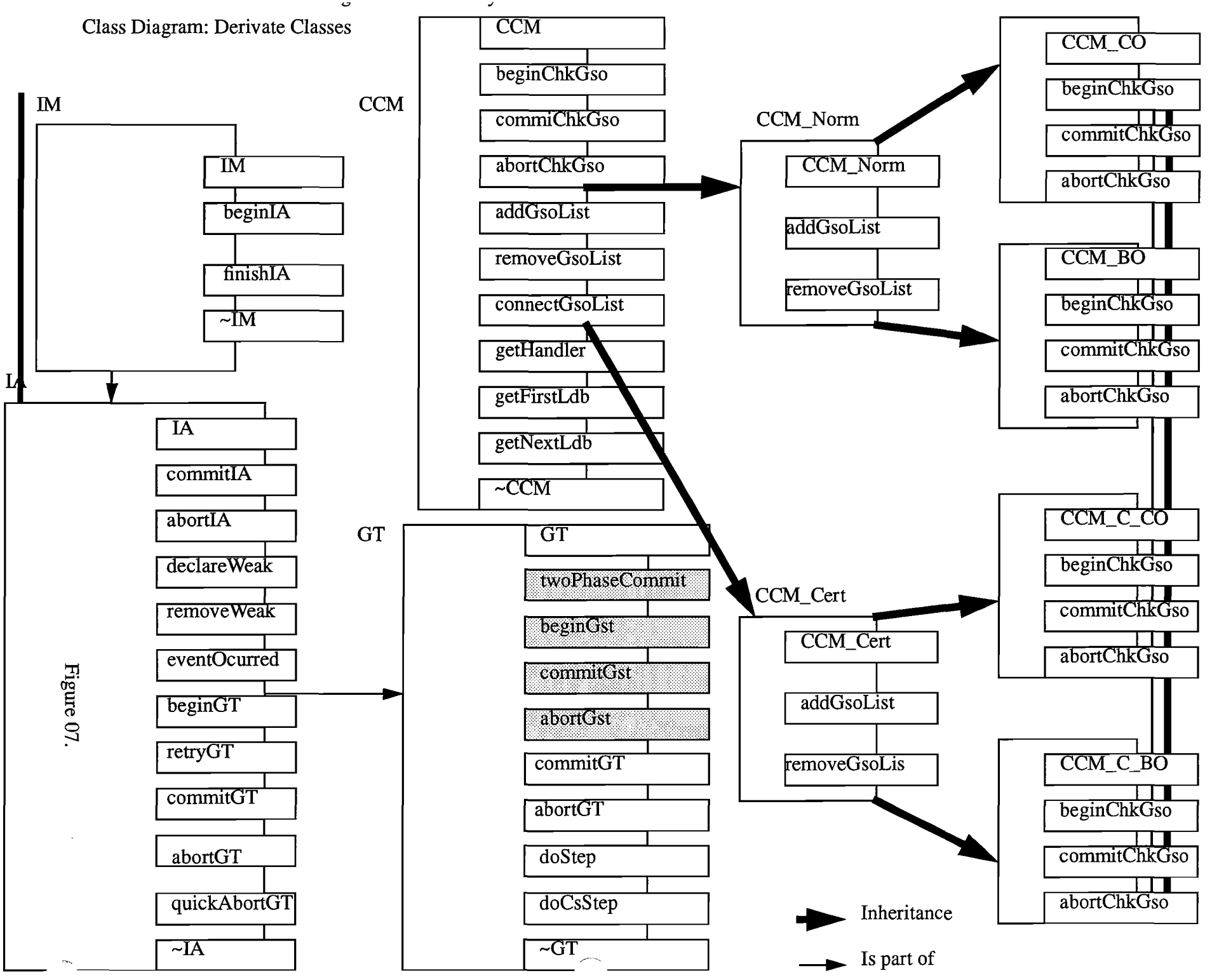

Figure 07.

II- Description of the Main Classes

II.1- Global level

1- Class IM

- Abstraction:

This Class is part of the Interaction Manager module. It keeps track off all the InterActions running at the system and start-up the concurrency control manager.

- Data Members:

1. ObjList im_ia_List; // list of active IA oids

- Private Member Functions:

- Public Member Functions:

1.	IM()	// Constructor of Class IM
2. Oid	beginIA(char* serverName)	// begins Interaction
3. Status	finishIA(signComAbort flag)	// finish interaction
4.	~IM()	// destructor

-Relationships:

1.1 - Member Functions of Class IM

1.1.01- Member Function: IM::im

-Semantics:

Constructs IM class initializing im_ia_list .

-Called by:

Tasl module

-Calls:

-Parameters:

-Returns:

1.1.02- Member Function:IM::begin IA

-Semantics:

This function call the constructor of class IA to create another Interaction and add the object id to the im_ia_list. It creates handlers to communicate with TASL module and the IRS module Request service IALOG_LOG_IA_BEGIN to the IRS server to log IA begin at the master log Returns IA oid if succeed or NULL if fails.

-Called by:

Tasl Interpreter

-Calls:

ObjList::append(oid)
IA::IA(Tasl handler)
IALOG_LOG_IA_BEGIN

-Parameters:

char* servername- server name where IM server is running

-Returns:

IA * iaoid if succeed otherwise Null.

1.1.03- Member Function:IM::finish IA

-Semantics:

Calls abortIA or commitIA according to the flag passed. If function (abortIA or commitIA) succeed, call the destructor ~ia() and remove iaOid from the im_ia_list. Otherwise return NOT_OK.

-Called by:

Tasl Interpreter

-Calls:

ObjList::remove(Oid iaoid)
IA::~ia()
IA::abortIA()
IA:: commitIA()
IALOG_LOG_IA_ABORT
IALOG_LOG_IA_COMMIT

-Parameters:

OId iaoid- object id of the Interaction
SignComAbort flag- indicates which function to call (abortIA or commitIA)

-Returns:

OK if succeed or NOT_OK if fails.

1.1.04- Member Function:IM::~im

-Semantics:

This function calls emptyP to check if some interaction still active. If not, destroy instance of class IM.

-Called by:

Tasl Interpreter

-Calls:

ObjList::emptyP()

-Parameters:

none

-Returns:

nothing

2- Class IA

- Abstraction:

This Class manages the InterAction running at the system and its Global Transactions.

- Data Members:

1. Oidlist ia_gt_list // list of active GTs with respectives GT oids

- Private Member Functions:

- Public Member Functions:

1. IA(CLIENT* taslhandle) // Constructor of Class IA
2. Status commitIA(); // commit Interaction
3. Status abortIA(); // abort interaction
4. int declareWeak(char* ldbname, ACtype event_type, int gtoid, int stepid,,int argc,char** argv,ACcond cond,char* value) // returns event id
5. Status removeWeak(char* ldbname, int eventid); // remove weak conflict
6. void eventOcurred(int evId) // informs the ocurrence of a event
7. Oid beginGT(intlist predList) // starts new GT
8. Status retryGT(Oid gtOid) // abort and retry GT
9. Status commitGT(Oid gtOid) // commit GT and destroy GT instance
10. Status abortGT(Oid gtOId) // calls IRS (major abort)
11. Status quickAbortGT(Oid gtOid) // aborts running GT and destroy GT instance
12. ~IA() // destroy instance IA class

-Relationships:

...

2.1- Member Functions of Class IA

2.1.01- Member Function: IA::ia

-Semantics:

This function creates an instance of class IA and stores the handler to communicate with TASL.

-Called by:

IM::beginIA()

-Calls:

-Parameters:

CLIENT* taslhandler- handler to communicate with TASL.

-Returns:

2.1.02- Member Function: IA::commitIA

-Semantics:

This function checks for running GTs at the ia_gt_list. Request service AC_REMOVE _ALL to the agent server to remove weak conflicts from this IA. Returns OK if succeed or NOT_OK if fails.

-Called by:

Tasl

-Calls:

ObjList::emptyP()
AC_REMOVE_ALL

-Parameters:

None

-Returns:

OK if succeed
NOT_OK if fails

2.1.03- Member Function: IA::abortIA

-Semantics:

Request service ILD_ABORT_IA to IRS server to procede the abortion of the IA.

-Called by:

Tasl

-Calls:

ILD_ABORT_IA

-Parameters:

None

-Returns:

OK if succed or NOT_OK if fails

2.1.04- Member Function: IA::declareWeak

-Semantics:

Request service AC_ASSIGN_EVENT_ID to the agent server to set a weak conflict.

-Called by:

Tasl

-Calls:

AC_ASSIGN_EVENT_ID

-Parameters:

char*	ldbname - local database name
AcType	eventType-
int	gtoid - Global transaction Id.
int	stepId- step identification to the step library
int	argc
char**	argv
AcCond	cond -
char*	value

-Returns:

int evid- event id.

2.1.05- Member Function: IA::removeWeak

-Semantics:

Request AC_REMOVE_EVNT service from the agent server to remove a weak conflict from the activatore given the local database name anda the event Id.

-Called by:

Tasl

-Calls:

AC_REMOVE_EVNT

-Parameters:

char *	ldbname- local database name
int	eventId.

-Returns:

OK if succeed or NOT_OK if failed

2.1.06- Member Function: IA::eventOcurred

-Semantics:

Signals to Tasl that an event occured in the local database by requesting service TS_EVEN-T_OCCURRED to the Tasl server for this particular IA.

-Called by:

Activator

-Calls:

TS_EVENT_OCCURRED

-Parameters:

int evId- event id.

-Returns:

void

2.1.07- Member Function: IA::beginGT

-Semantics:

Calls the constructor of class GT. Calls the beginChkGso at the CCM to check serialization according to the policy used.
If beginChkGso succeed:

request service GTLOG_LOG_GTBEGIN at the IRS server
if service succeed:
adds an GT oid at the ia_gt_list
return OK

else

destroy instance of GT
return NOT_OK

-Called by:

Tasl
IRS

-Calls:

OidList::append(gtoid)
CCM:: beginChkGso(gtOid)
IRS::logBeginGT
GT::gt()
GT::~gt()

-Parameters:

IntList preList - previous list of GTs in the execution order. NULL if this is a compensating GT only.

-Returns:

gtOId if succeed or NULL if fails.

2.1.09- Member Function: IA::commitGT

-Semantics:

```
Ckeck If GToid is in ia_gt_list
if found:
    calls GT::CommitGT
     if succeed :
          request service GTLOG_GT_COMMIT at the IRSserver
           call destructor for GT
           remove gtoid from ia_gt_list
           return OK
     else
          request service GTLOG_GT_ABORT at the IRSserver
           call destructor for GT
           remove gtoid from ia_gt_list
            return NOT_OK
else
    retunr NOT_OK
```

-Called by:

Tasl Interpreter
IRS

-Calls:

OidList::remove(gtoid)
GT::~GT()
GT::commitGT(ccmOid)
GTLOG_LOG_GT_ABORT
GTLOG_LOG_GT_COMMIT

-Parameters:

GT * gtOid - object identity of the global transaction

-Returns:

OK if succeed or NOT_OK if fails

2.10- Member Function: IA::abortGT

-Semantics:

Request service ILD_ABORT_GT from IRS to perform abortion of the GT.
if succeed:
    request GTLOG_LOG_GT_ABORT from the IRS server
    return OK
else
    return NOT_OK

-Called by:

TASL
IRS

-Calls:

ILD_ABORT_GT
GTLOG_LOG_GT_ABORT

-Parameters:

GToid - global transaction id.

-Returns:

OK if succeed or NOT_OK if failed

2.11- Member Function: IA::quickAbortGT

-Semantics:

```
Check GToid at the ia_gt_list.
 if found
      call GT::abortGT
      if GT::abortGT succeed
           destroy GT object
           remove GToid from ia_gt_list
           request service GTLOG_LOG_GT_ABORT to IRS.
           return OK
        else
           return NOT_OK
  else
     return NOT_OK
```

-Called by:

TASL

-Calls:

GT::abortGT()
GTLOG_LOG_GT_ABORT
oidList::remove(gtoid)

-Parameters:

Oid gtoid- Global transaction id.

-Returns:

OK - if succeed
NOT_OK - if failed

2.12- Member Function: IA::~ia

-Semantics:

Calls destructor of OidList to destroy ia_gt_list
destroy instance of IA

-Called by:

IM::finishIA(iaOid,flag)

-Calls:

-Parameters:

none

-Returns:

OK if succeed
NOT_OK - If failed

3- Class GT

- Abstraction:

This class manages the excution of the Global Transaction and its Global Sub-Transactions.

- Data Members:

1. SysLdbList local_ldb_List // list of all Local Databases being accessed by the global transaction thru its GTS

- Private Member Functions:

1. Status twoPhaseCommit() // executes prepare & commit phase
2. Status beginGST(Ldbname) // begin Gst at the local database
3. Status commitGst(ldbname) // commit Gst at the local database
4. Status abortGST(Ldbname) // aborts Gst at the local database

- Public Member Functions:

1. gt() // Constructor of class GT
2. Boolean commitGT(sysldblist) // commit GT
3. Boolean abortGT() // abort GT
4. Boolean doStep(char* ldbname,
   int stepid,
   int argc,
   char** argv) // request AM to execute step
5. Boolean doCsStep(char* ldbname,
   int Csid,
   int argc,
   char** argv) // request AM to execute Compensate step
6. ~gt() // Destructor

-Relationships:

3- Member Functions of Class GT

3.01- Member Function: GT::gt

-Semantics:

This function creates an instance of the class GT . . Remote procedure calls createGTdir at ILD.

-Called by:

IA::beginGT(predList)

-Calls:

ILD::createGTdir ?????

-Parameters:

none

-Returns:

none .

3.02- Member Function: GT::twoPhaseCommit

-Semantics:

For every local database that participates at the GT, get agent handler at the CCM and rpc AM for vote at the local database. (prepare phase) If any local database answer no return NOT_OK. If all answers OK, for every local database rpc AM to commit . If successful comit return OK.

-Called by:

GT::CommitGT()

-Calls:

SysLdbList::findFirst(ldbname,agHandler)
SysLdbList::findNext(ldbname,agHandler)
CCM::getHandler(ldbname)
rpc : GST_Vote
rpc: AM_Finish_Gst

-Parameters:

none

-Returns:

OK, NOT_OK

3.03- Member Function: GT::commitGT

-Semantics:

Call CCM::commitChkGso(gtoid,ldblist) to ditacte seialization (according with the serialization policy used). If serialization succed call GT::twoPhaseCommit() otherwise return NOT_OK

-Called by:

IA:: finishGT(gtOid,flag)

-Calls:

GT:: twoPhaseCommit()
GT::AbortGT()
CCM::commitCheckGso(gtOid,ldblist)

-Parameters:

none

-Returns:

OK, or NOT_OK.

3.04- Member Function: GT::abortGT

-Semantics:

Check in the LDB_List which Local Databases Id the Global Transaction is executing on. For each Local Database calls Abort_GST(). If abortGST(ldbid) returns OK then remove ldbId from the ldb_list.

-Called by:

TASL Interpreter

-Calls:

abortGST(gtOid)
removeIntList

-Parameters:

-Returns:

True if suceed or False if Fail.

3.05- Member Function: GT::doStep

-Semantics:

Get agent handler to comunicate with the agent of the local database. If ok get self-pointer to class GT. Executes a RPC to the Agent Manager to execute step. Returns OK if succeed or NOT_OK if fails. It also calls IRS to log

-Called by:

Tasl Interpreter

-Calls:

writeAgtLogGSTrec

-Parameters:

char *	ldbname	- local database name
Oid *	setpid	- step library Id.
int	argc	- Number of arguments
char*	argv	- arguments for the step libraryl

-Returns:

OK if succeed or NOT_OK if fails

3.06- Member Function: GT::doCsStep

-Semantics:

Calls (function-name ???) at the serialization enforcer to execute a compensate step to undo the effect of a global sub-transaction and ultimately the global transaction.
QUESTION: should we log this execution ?

-Called by:

ILD (who at the ILD?)

-Calls:

?

-Parameters:

int ldbId - local database ID.
int CsNu - Compensate step number.

-Returns:

True if succeed or False if fails

3.07- Member Function: GT::beginGST

-Semantics:

Calls addIntList to add another ldbId. It also remote procedure call to record begin of Global Sub-Transaction at the Agent -log (gstFileCreate)

-Called by:

Tasl Interpreter

-Calls:

addIntList

-Parameters:

int ldbId - Local Transaction ID

-Returns:

True if succeed or False if fail.

3.08- Member Function: GT::abortGST

-Semantics:

Send message to local Database (RPC function) to abort the given Global Transaction. If Aborted successfuly call removeIntListt to remove ldbid.

-Called by:

abortGT(ccmOid)

-Calls:

removeIntList(ldbId)

-Parameters:

int ldbId- Local Database

-Returns:

True if succeed or False if Fail

3.09- Member Function: GT::~gt

-Semantics:

Checks for some ldbID at ldb_List . If empty destroy instance of class GT.

-Called by:

finishGT(gtOid,flag)

-Calls:

getIntList

-Parameters:

int GTID - Global Transaction ID.

-Returns:

True if succeed or False if fails.

4- Class CCM

- Abstraction:

This is virtual class that is the base class for the classes CCM_CO, CCM_BO, CCM_Cert_CO, CCM_Cert_BO.

- Data Members:

- Private Member Functions:

- Public Member Functions:

1. ccm() // Constructor of class CCM generates by the compiler
2. Boolean beginChkGso(gtOid) // virtual function
3. Boolean commitChkGso(gtOid) // virtual function

-Relationships:

5- Class CCM_Norm

- Abstraction:

This class ensures Global Serialization Order by calling member functions of one of the two serialization order (Commit Order or Begin Order).

- Data Members:

1. ObjIdList ccm_norm_gso_list

- Private Member Functions:

- Public Member Functions:

Boolean addGsoList(GTID) // add Global Transaction to the list of Global
// Serialization order

Boolean removeGsoList(GTID) // remove Global Transaction of the list of
// Global Serial ization

-Relationships:

Super Class of CCM_CO and CCM_BO
Sub Class of: CCM_Core

5.1- Member Functions of Class CCM_Norm

5.1.01- Member Function: CCM_Norm::addGsoList

-Semantics:

Add Global Transaction oid to the list ccm_norm_gso_list..

-Called by:

CCM_BO::beginChkGso
or CCM_CO::commitChkGso

-Calls:

addObjIdList(oid) note: probably uses templates

-Parameters:

GT * gtOid - global transaction object id..

-Returns:

True if succeed or False if fails

5.1.02- Member Function:CCM_Norm::removeGsoList

-Semantics:

Remove the global transaction object id from the ccm_norm_gso_list.

-Called by:

commitChkGso

-Calls:

removeObjList(oid) Note: probably uses templates

-Parameters:

GT * gtOid- object identity of GT class .

-Returns:

True - if succeed and False if fails

6- Class CCM_CO

- Abstraction:

This class ensures Global Serialization Order as the same as the global transaction commit order.

- Data Member:

- Private Member Functions:

- Public Member Function:

1. Boolean beginChkGso(gtOid);
2. Boolean commitChkGso(gtOid);

-Relationships:

Sub Class of: CCM_Norm

6.1- Member Functions of Class CCM_CO

6.1.01- Member Function:CCM_CO::beginChkGso

-Semantics:

do nothing...

-Called by:

beginGT(prevList)

-Calls:

-Parameters:

GT * gtOid- Global Transaction object id.

-Returns:

True - if succeed and False if fails

6.1.02- Member Function:CCM_CO::commitChkGso

-Semantics:

Calls addGsoList to add the global transaction in the ccm_norm_gso_list. For each local database of the ldb_list calls serializeGT(gtOid). If some of the calls of the serializeGT returns false then return false otherwise return true.

-Called by:

commitGT(ccmOid)

-Calls:

addGsoList(GT * gtOid)
serializeGT(gtOid)

-Parameters:

GT * gtOid - global transaction object id.
intList * ldb- pointer to list of local databases of the system ??

-Returns:

True - if succeed and False if fails

7- Class CCM_BO

- Abstraction:

This class ensures Global Serialization Order as the same as the global transaction Begin order.

- Data Member:

- Private Member Functions:

- Public Member Function:

1. Boolean beginChkGso(gtOid);
2. Boolean commitChkGso(gtOid);

-Relationships:

Sub Class of: CCM_Norm

7.1- Member Functions of Class CCM_BO

7.1.01- Member Function: CCM_BO::beginChkGso

-Semantics:

Calls addGsoList(gtOid) to add the gtOid at the ccm_norm_gso_list. Calls serializeGT for all local databases of the system

-Called by:

beginGT(predList)

-Calls:

addGsoList(gtOid)
serializeGT(gtOid)

-Parameters:

GT * gtoid- global transaction object id
intList * ldb- local database list

-Returns:

true if succeed or false if fails

7.1.02- Member Function: CCM_BO::commitChkGso

-Semantics:

For each local database at the local database list, calls checkSerializeGT(ldbid). If some function return false exit returning false. To all local databases that do not participate at the ldb list call abortGST. if some function return false exit returning false.

-Called by:

commitGT(ccmOid)

-Calls:

checkSerializeGT(gtOid)
abortGST(ldbId)

-Parameters:

GT * gtOid- global transaction oid

-Returns:

true if succeed or false if fails.

8- Class CCM_Cert

- Abstraction:

This class uses Certification to ensure that GSO is maintained.

- Data Member:

1. ListObjIdList ccm_cert_ldb_gso; //the GSO lists of the LDBs

- Private Member Functions:

- Public Member Function:

1. Boolean addGsoList(gtOid);
2. Boolean removeGsoList(gtOid);

-Relationships:

Sub Class of: CCM_Norm

8.1.- Member Function of Class CCM_Cert

8.1.01- Member Function: CCM_cert::appendGsoList

-Semantics:

Appends the local global transaction id list sent by the agent to the proper list at the ccm_cert_ldb_gso

-Called by:

CCM_C_BO::beginchkGso
or CCM_C_CO::commitChkGso

-Calls:

addObjIdList

-Parameters:

int ldbId - local database id.
objIdList * gtList- list of gtOids of the running at the local database

-Returns:

true if succeed or false if fails

8.1.02- Member Function: CCM_Cert::removeGsoList

-Semantics:

for each local database calls removeObjIdList to delete gtOid from every list.

-Called by:

commitChkGso

-Calls:

removeObjIdList(gtOid)

-Parameters:

GT * gtOid - global transaction oid.

-Returns:

true if succeed or false if fails

9- Class CCM_C_CO

- Abstraction:

This class ensures Global Serialization Order as the same order as the Commit order of the Global Transactions.

- Data Member:

- Private Member Functions:

- Public Member Function:

1. Boolean beginChkGso(gtOid);
2. Boolean commitChkGso(gtOid);

-Relationships:

Sub Class of: CCM_Cert

9.1- Member Function of Class CCM_C_CO

9.1.01- Member Function:CCM_C_CO::beginChkGso

-Semantics:

do nothing (returns true)

-Called by:

beginGT(predList)

-Calls:

-Parameters:

GT * gtOid - Global transaction Oid.

-Returns:

true if succeed

9.1.02- Member Function: CCM_C_CO::commitChkGso

-Semantics:

Checks in the ccm_Cert_ldb_List which ldb participate at the given GT. For each ldb calls requestGsoList. Calls appendGsoList(ldbid,Gtlist) for update the ccm_Cert_ldb_List for every list of global transaction received. Check at the participating ldbs on the ccm_cert_ldb_List if the gtOid of the first GT is the same as the given GT. If not call abortGT for all transactions that preceed the given GT.

-Called by:

commitGT(ccmOid)

-Calls:

requestGSOList
appendGsoList
checkGsoList
abortGT

-Parameters:

Gt * gtOid- global transaction Id.

-Returns:

true if succeed or false if fails

10- Class CCM_C_BO

- Abstraction:

This class ensures Global Serialization Order as the same as the Begin orders of the Global Transactions.

- Data Member:

- Private Member Functions:

- Public Member Function:

1. Boolean beginChkGso(gtOid);
2. Boolean commitChkGso(gtOid);

-Relationships:

Sub Class of: CCM_Cert

10.1- Member Functions of Class CCM_C_BO

10.1.01- Member Function: CCM_C_BO::beginChkGso

-Semantics:

do nothing (return true)

.

-Called by:

beginGT(predList)

-Calls:

-Parameters:

GT * gtOid- global transaction Oid.

-Returns:

true

10.1.02- Member Function: CCM_C_BO::commitChkGso

-Semantics:

Checks in the ccmCertGsoList which ldbs participate at the given GT. For each ldb calls requestGsoList. Calls appendGsoList for each list of gtOid received from the function requestGsoList. Check at the participating ldbs on the ccm_cert_ldb_;list if given gT is the firstGT of the list . If not call abortGT for the given GT.

-Called by:

commitGT(ccmOid)

-Calls:

requestGsoList
appendGsoList
checkGsoList
abortGT

-Parameters:

GT * gtOid- global transaction oid.

-Returns:

true if succeed or false if fails.

References:

[1] R. Alonso, H. Garcia-Molina, and K. Salem. Concurrency Control and Recovery for global procedures in federated database systems. A quarterly bulletin of the computer society of the IEEE technical committee on data engineering 10(3) September 1987

[2] Y. Breitbart, A. Silberschatz, and G. Thompson. An update mechanism for multidatabase systems. A quarterly bulletin of the Computer Society of the IEEE technical committee on Data Engineering, September 1987.

[3] W. DU and A. Elmagarmid. QSR: A correctness criterion for global concurrency control in Interbase. In proceedings of 15th International conference very large. Databases 1989.

[4] Ahmed K. Elmagarmid, Jin Jing & Won Kin. Global Commitment In Multidatabase Systems. Department of Computer Sciences Purdue University. March 1991

[5] Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit Sheth. On serializability of multidatabase transactions through forced local conflicts. In 1991 Data Engineering Proceedings, pages 314-323, 1991.

[6] Hector Garcia Molina and Kenneth Salem. Sagas. In ACM SIGMOD proceedings, pages 249-259. ACM, 1987.

[7] Dennis Heimbinger and Dennis Mclead. A Federated Architecture for Information Management. ACM transactions on office Automation Systems 273-278. July 1985.

[8] W. Litwin. From database systems to Multidatabase systems: Why and How. In British National Conference on Databases. Cambridge Press, 1988.

[9] T. Logan & A. Sheth. Concurrency Control Issues in Heterogeneous Distributed Database Management Systems. Unpublished July 1986.

[10] Marian H. Nodine. InterActions: Multidatabase Support for Planning Applications. Technical Report CS 91-64. Brown University, December 1991.

[11] Marian H. Nodine. Interactions: Multidatabase Support for Planning Applications. Phd Thesis - Department of Computer Science, Brown University, May 1993

[12] Peter Muth and Thomas G, Rakow. Atomic commitment for integrated database systems. In 1991 Data Engineering Proceedings, pages 296-304, 1991.

[13] Noela V. Nakos. Specification Environment For Multidatabase Applications. Department of Computer Science. Brown University, May 1993

[14] Sergio Nakai. The Concurrency Control Mechanism of the Mongrel System. Design & Implementation. Department of Computer Science. Brown University May 1993

[15] Marian H. Nodine and Stanley B. Zdonik. Supporting Reactive Planning Tasks On An Evolving Multidatabase. Technical Report Brown University December 1992

[16] Marian H. Nodine and Stanley B. Zdonik. Automating Compensation in Multidatabase. Proceedings of the 27th. Annual Hawaii International Conference on System Sciences, Volume II pp. 293-302. January 1994.

[17] Helmut Waechter and Andreas Reuter. The ConTract model. In A. Elmagarmid editor. Database Transaction Models for Advanced Applications. Morgan-kauffman, 1991