BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M18

"Local Database Support for Long-Term Multidatabase Transactions"

by

Ming-Tsung Lu

Local Database Support for Long-Term Multidatabase Transactions

by

Ming-Tsung Lu
B.A., Soochow University, Taiwan 1990.

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

May, 1993

Local Database Support for Long-Term Multidatabase Transactions

by

Ming-Tsung Lu
B.A., Soochow University, Taiwan 1990.

Abstract

This paper presents the interface between the Mongrel prototype multidatabase system and its local database systems. The multidatabase supports long-term planning transactions that are reactive to the changes in the real world. The interface supports operations on heterogeneous databases that enforce local database autonomy. The data model and operations of the local databases have to be encapsulated to the multidatabase users. The Multidatabase has to be informed of all the relevant changes made to the local databases. This paper describes the design of a general-purpose interface, which is implemented on top of an Object-oriented Database called ObjectStore.

1. Statement of Contributions

In this project, we designed and implemented the interface between the multidatabase system and the local database systems and assisted the integration of the entire system.

The components of the interface we designed and implemented include:

* Activator.

* Global Subtransactions.

* Jump Vector

* Local Database Schema Definition.

* Step Libraries and Step Library Functions.

* Interactive Test Driver acting as TaSL stub.

This research project by Ming-Tsung Lu 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.

Date 28 April, 1993

Stanley B. Zdonik

Stanley B. Zdonik

2. Project Overview.

2.1 What is a Multidatabase System?

Some applications, especially planning applications, require information from several different existing databases. Manipulation of the information located in these databases requires an additional software layer on top of the existing database systems. This layer is called a *Multidatabase system*. A multidatabase is a collection of separate local databases. A user of a multidatabase can access and update the information in these local databases in a consistent way to accomplish some common purpose. Therefore, the multidatabase system extends user capabilities, enables users to access and share data without learning the intricacies of different database management systems. In addition, the existing programs and procedures of the local databases remain operational in the integrated multidatabase environment.

2.2 Characteristics of Local Databases:

The local databases have the following characteristics:

(1) *Heterogenity* - The different local databases may support different data models, data manipulation languages and concurrency control protocol.

(2) *Autonomy* - There should not be any modifications made to the existing local database functions or protocols to support the multidatabase. While the multidatabase is accessing the data in the local database, the user from the local database can still run independent transactions to access the information in the local database. The independent transactions are not part of the multidatabase transactions.

(3) *Distribution* - The local databases may be geographically distributed.

Therefore, it is essential that we take the above characteristics into consideration and provide an uniform interface between the local databases and the multidatabase.

2.3 Characteristics of Long-Term Transactions:

Traditional database transactions were developed in the context of data processing application in which most transactions are noninteractive, of short duration and atomic. However, planning application tasks tend to be *Long-Term Transactions* consisting of short activities separated by long periods of idle time. It is not reasonable to represent a planning task to the database as a seri-

alizable transaction because it may have a huge impact on database performance. Also, planning applications need to respond to the changes in the real world. Therefore, a planning application

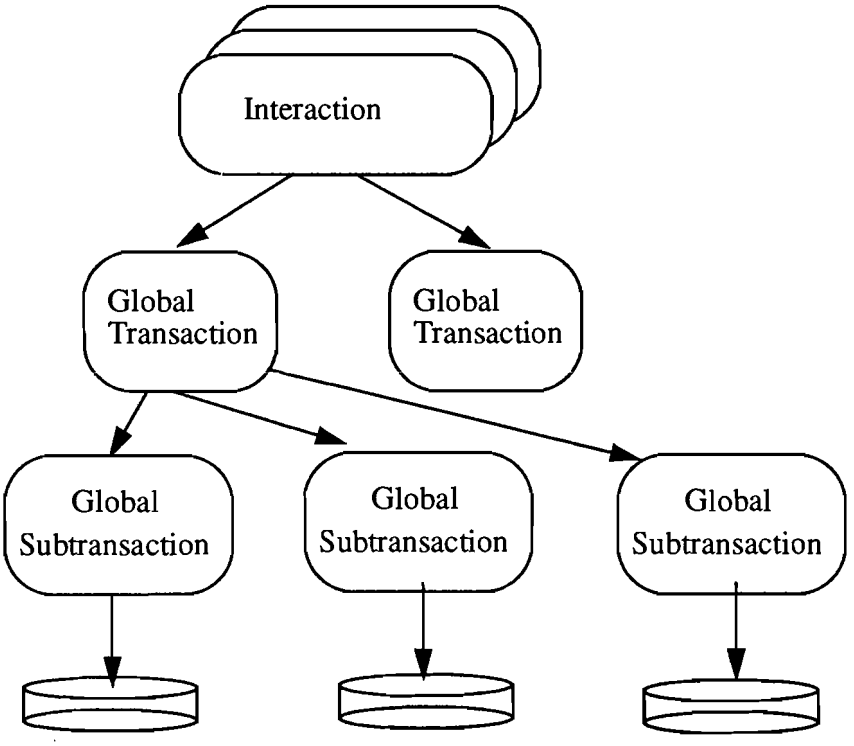


Figure 1 - Interaction Structure.

can not run in isolation.

In this system, we call a Long-Term Transaction an *InterAction*. It is a partially-ordered set of atomic *Global Transactions*. The Global Transaction execution dependencies are represented by the partial order of the Global Transactions. Each Global Transaction consists of a set of *Global Subtransactions*, each of which executes on a single local database. (See figure 1.)

2.4 Overall Structure.

We designed and implemented a multidatabase system, called Mongrel. It supports the long-term transactions and is responsible for ensuring that the tasks it executes on the local databases maintain data consistency. This system has two logical levels, a Global Level and a Local Level. (See figure 2.) The local level consists of a set of local databases and the interface sitting on top of them. The information in the local databases is accessed and updated using Global Subtransac-

tions. The local level provides an uniform access interface, which ensures that the user of the multidatabase can access information in the local databases using a single database access model, and without having to learn the local data models or to know in which local database the information

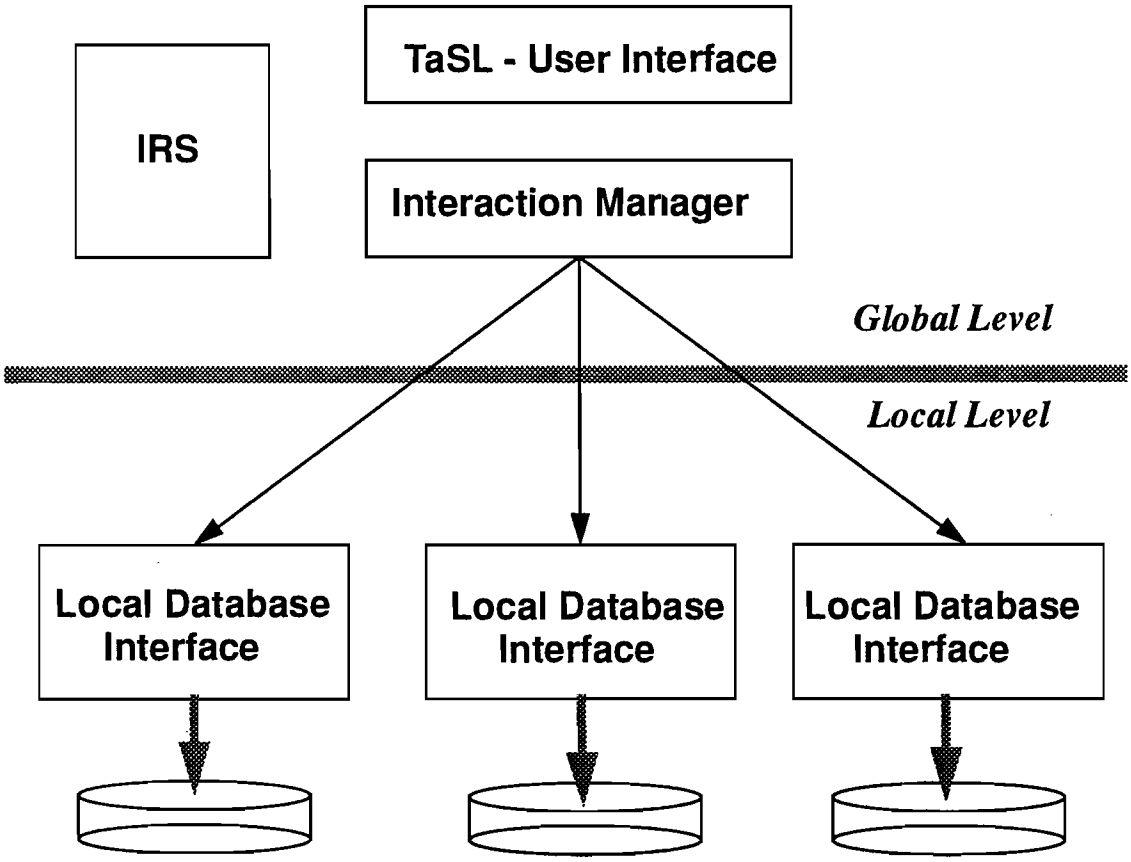


Figure 2 - Multidatabase Structure.

is stored. That is to say, the details of the local databases should be kept hidden from the multidatabase users. In addition, we also monitor the changes made to the local database so that the multidatabase system can react to the changes. Multidatabase users can tell the system what kind of changes are relevant to the execution of the Interactions. The local level is responsible of checking to see if the event occurs in the local database and notifies the user.

3. Overview of Local Level.

The Local Level consists of five major components: Agent Manager, Global Subtransactions, Step Library, LRS and Activator. (See figure 2.) The design and implementation of the Global Subtransaction, Step Library, Activator and the Schema of the local databases are described in

Local Level - Local Database Interface.

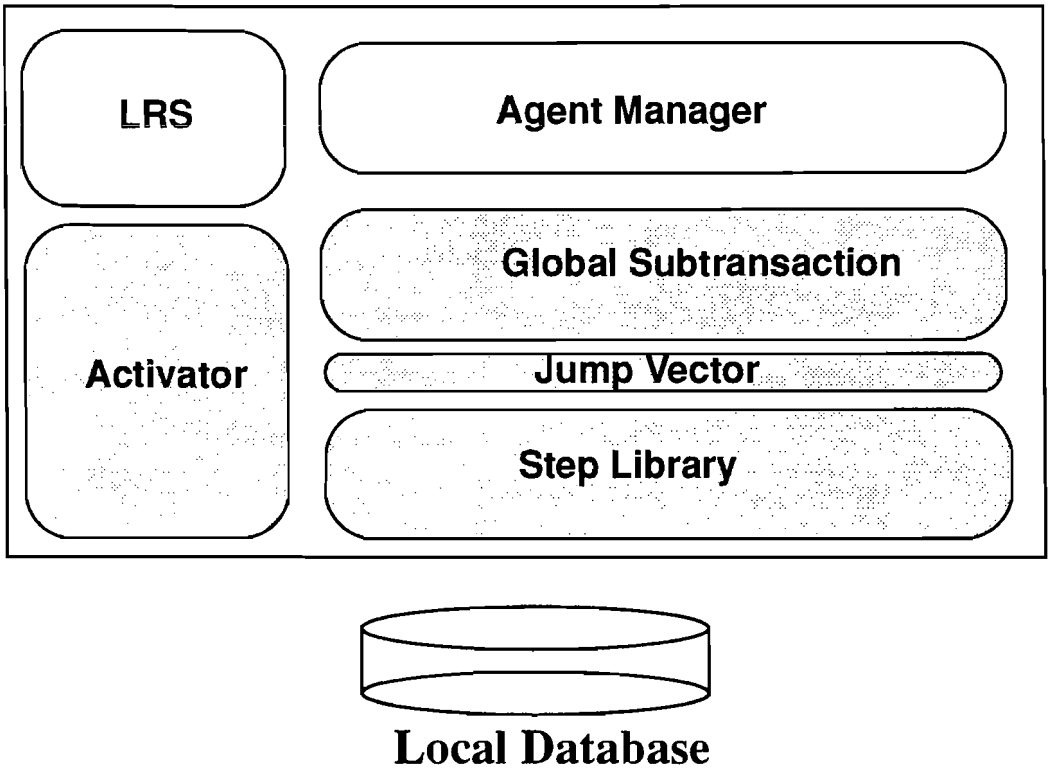


Figure 3 - Structure of the Local Level.

this paper.

3.1 Local Level Setup.

The Local Level basically consists of four processes running concurrently, including an Agent Manager server, an LRS server, an Activator server and an Activator client. To support the multi-

database operations on a local database, we need to start an Agent Manager server on a local database. The Agent Manager forks an LRS server and an Activator server, which forks an Activator client. (See figure 4.) When a Global Subtransaction is about to start, a Global Subtransaction process is forked and it lives until the Global Subtransaction is either committed and aborted. There is one Global Subtransaction process per active Global Subtransaction. Global Subtransactions access the local database through the functions defined in the Step Library. Note that Step Library

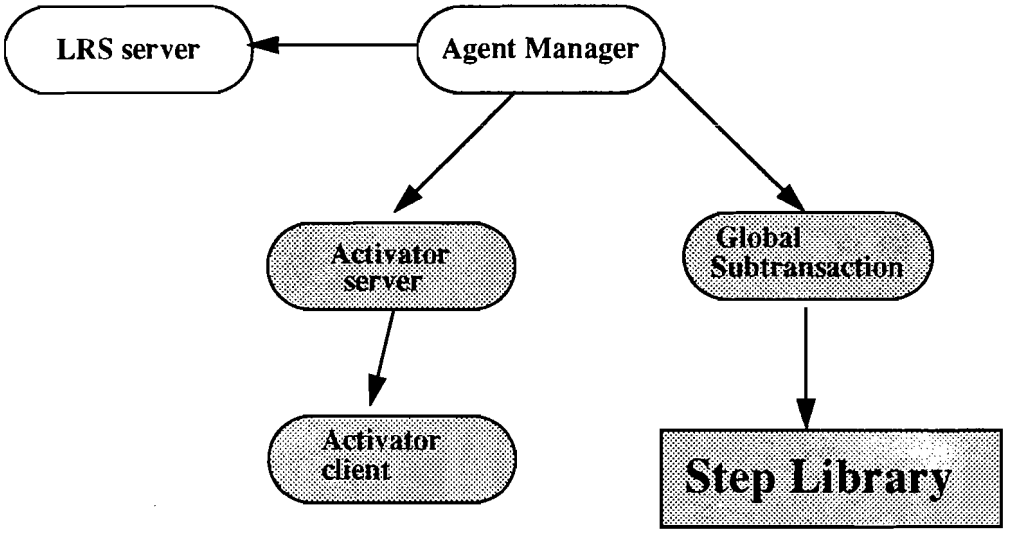


Figure 4 - How processes are created.

is not a process. It is a shared library. There is a Jump Vector acting as the interface between the Global Subtransactions and the Step Library.

If an Interaction wants to monitor the occurrence of a certain event, it tells the Agent Manager and the Agent Manager tells the Activator to check on that event. The Activator client will be explained later in this section.

During the life time of an Interaction, we may have to back out the effects of (undo) the Global Subtransactions committed long time ago. Therefore, a Global Subtransaction has to log the parameters necessary to the compensating steps. The LRS is responsible for doing the logging. If later on we need to undo a committed Global Subtransaction, we can retrieve the parameters to the compensating functions from the LRS and run the function to abort the Global Subtransaction.

3.2 Activator

An event is some change in the multidatabase that influences the past execution of the Interaction. It is an update operation on a specific data item that violates some conditions in the database. An event is provoked by some other Interaction or some independent transactions in the local database.

During a specific execution span of an Interaction, there are some conditions that should be preserved for its overall effects in the multidatabase to remain consistent. We call these conditions *weak conflicts*.

There are events on the local database that are explicitly being waited for. We call them *wait events*.

The job of the Activator is to monitor the changes made to the local database to check if the changes may affect the execution of Interactions. The Activator receives messages from the Interaction Manager via the Agent Manager. The messages indicate events that are interesting to some Interaction. The events are stored in the event table, along with the conditions the Activator needs to check on. Each event is either a wait event or a weak conflict. Wait events are events on the local database that are explicitly being waited for. They are removed from the event table as soon as they occur. Weak conflict events are events on the local database that indicate that a weak conflict has been violated. The Activator polls the local database every certain period of time to see if the events in the event table have occurred. If so, it notifies the Interaction Manager. The Activator client process is responsible of calling the Activator server every certain period of time to poll the local database.

3.3 Global Subtransaction

An Interaction is divided into several atomic parts called Global Transactions. Each Global Transaction consists of a set of Global Subtransactions, each of which executes on a single local database. No two Global Subtransactions of the same Global Transaction execute on the same local database. A Global Subtransaction server process is forked by the Agent Manager when the Glo-

bal Subtransaction begins. The process terminates when the Global Subtransaction commits or aborts. The Global Subtransactions access the local database through Step Library. There are usually a number of Global Subtransaction servers running concurrently on top of the local database. Each Global Subtransaction process represents a Global Subtransaction of some active Global Transaction.

In addition to accessing the data in the local database, the Global Subtransaction also needs to talk to LRS in order to record the necessary information for later execution of compensating steps.

3.4 Jump Vector

The Jump Vector is the interface between Global Subtransactions and Step Library. It contains function pointers to the functions in the Step Library as well as the function pointers to their corresponding compensating steps. When a Global Subtransaction needs to run a function in the Step Library, it gets the pointer to the function it wants to run from the Jump Vector and executes the function. The Jump Vector also provides a convenient way to find the compensating step for the Step Library Functions. To run a compensating step in order to abort a committed Global Subtransaction, the Global Transaction gets the pointer to the compensating step from the Jump Vector and retrieves the parameters to that function from LRS. Then it can do the compensating step.

3.5 Step Library

Step Library is the interface between the Global Subtransactions and the local database. It defines how the multidatabase can access the local database and provides a uniform interface to the users of the multidatabase. Each local database has its own Step Library. It provides the functions the multidatabase needs to access the information in the local database. Since there is one Global Subtransaction process per Global Transaction running on the local database, the Step Library is constructed as a shared library in order to save main memory space. This is the bottommost layer of the Local Level.

4. Problems.

We encountered some problems when we designed and implemented this system. The major problems are listed as follows:

(1) Keeping Event Tables Persistent.

Each Activator has its own event table, which records all the events interesting to the currently running Interactions. However, if the system crashes, we want to recover the information in the event table. Thus, we need to make the table persistent.

(2) Compatibility between ObjectStore and RPC.

Since we use ObjectStore to construct the local databases and we do Remote Procedure Calls a lot. We encountered a lot of problems when we try to link the object files generated by C++ compiler and the ones generated by the ObjectStore C++ compiler. The ObjectStore compiler always complains about "function undefined" if we try to compile a file, in which we make an RPC call, using ObjectStore compiler. An interesting thing is that we later found out that other ObjectStore users in industry also have the same problems.

(3) Global Subtransaction Memory Space.

We have one Global Subtransaction process running on top of local database for each Global Transaction. If we have lots of active Global Transactions, they will occupy a lot of memory space. However, it is essential that we have one process for each Global Subtransaction. We need to keep the Global Subtransaction as small as possible.

(4) Activator Polling Local Database.

The Activator has to monitor the changes made to the local database and checks to see if the update causes the occurrence of an event. The problem is that how does it find out if the data in the local database has been updated?

(5) Step Library Parameters and Return Value.

The Step Library Functions provide the multidatabase with the ways to access the data in the local database. Different functions do different things, (book a flight, cancel a reservation....) so they take different parameters and return different values. However, they need to provide a uniform interface. How do we arrange the parameters and return values so the user can pack the parame-

ters and unpack the results in the same way for all the Step Library Functions?

(6) Code Reusability.

Since the Local Level has to support the multidatabase running on top of all kinds of different databases. They can be databases of a completely different data model, or of different schema. How do we design the Local Level so that if we need to migrate the Local Level to a different database, we need to change only a small portion of the code to support the new local database?

(7) Finding the Compensating Step for a certain function.

Each Step Library Function has a corresponding compensation step. For example, if we have a function which books a plane ticket, there must be a function which deletes the reservation. We need to provide an easy way to find the compensating step of a function.

5. Solutions

The solutions to the problems we encountered are described as follows.

(1) Keeping Event Tables Persistent.

We use the ObjectStore database to store the event table persistently. Thus, the event table is kept in a separate database in the ObjectStore. There is another option for doing this. We can also keep the event table using the UNIX file system. However, since we need to store some complex data, such as the parameters to the Activator library functions, it is easier if we use the ObjectStore to store the event table. Using the UNIX file system can increase the portability of the Activator code. But it is harder to store and retrieve the event table information.

(2) Compatibility between ObjectStore and RPC.

A stub file is created for the ObjectStore functions making RPC calls. When the ObjectStore functions make RPC calls, they call the functions defined in the stub file. Then the stub file functions makes the RPC call. We compile the stub file using AT&T C++ Preprocessor and compile the ObjectStore functions using ObjectStore C++ compiler, then link them together. Thus we avoided the compatibility problems of RPC and ObjectStore.

(3) Global Subtransaction Memory Space.

Since every Global Subtransaction accesses the local database in a uniform way, we defined a Step Library for the local database. To save memory space, we construct the Step Library to be a shared library. Therefore, all the Global Subtransaction processes share the same copy of the Step Library code when they run.

(4) Activator Polling Local database.

The Activator forks a client, which goes to sleep and wakes up every five minutes. When it wakes up, it calls the Activator server to poll the local database. Theoretically, a better approach is to poll only when there is an update operation on the local database. In this method, the Activator always receives an asynchronous copy of the actual input message to the local database. The messages in this stream are in the local database's Data Manipulation Language. The Activator knows how to parse these messages to determine which messages update the local database, and what relations or sets in the database they update. After the Activator determines that, it checks whether constraints were violated by the update event.

This is indeed a good approach, however, this approach requires knowledge of the Query Manager of the local database system and may violate the local database autonomy. Therefore, we choose to use a simple polling approach. The advantage of the approach we chose is that it is simple and it does not require parsing the local database input. However, the drawback is that it is not very efficient. Each time it wakes up, it needs to poll the local database for every potential event.

(5) Step Library Parameters and Return Value.

Every Step Library Function has the same arguments, argc and argv. Each function interprets argc and argv differently inside the functions. When the functions return, we pack everything into a structure called stepReturnStruct and return that structure.

(6) Code Reusability.

In order to enhance the reusability of the code, we defined the Step Library to interface between the Local Level and the local database. There is also an interface between the Activator and the local database, called ACLDBinif. If we need to support the Local Level on top of a different ObjectStore database, the only parts that need to be changed are the Step Library, and the ACLDBintf class. However, since the whole Local Level is very specific to the local database system we are using. We need to redefine the Global Subtransaction, Step Library and the Activator for different local databases.

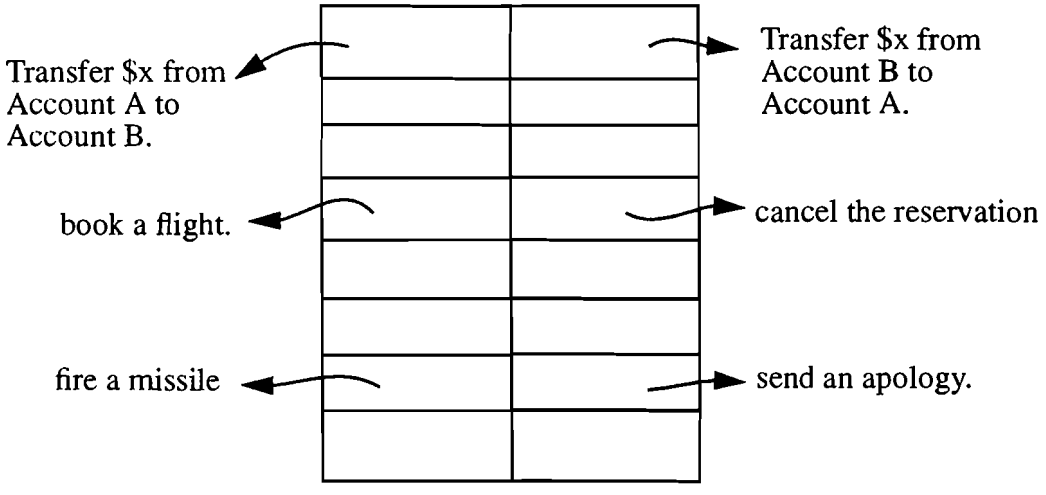


Figure 5 - Jump Vector - Like a mirror.

(7) Finding the Compensating Step for a certain function.

The Jump Vector is defined as a 2-D array of function pointers. (See figure 5.) The 2-D array serves like a mirror which reflects the function to its compensating step. Thus we can find the compensating step function very easily. The parameters to the compensating steps can be retrieved from the LRS system.

6. Bibliography

[1] Marian H. Nodine, Stanley B. Zdonik. Supporting Reactive Planning Tasks on an Evolving Multidatabase.

[2] Marian H. Nodine. InterActions: Multidatabase Support for Planning Applications.

[3] Marian H. Nodine. Supporting Long-running Tasks on an Evolving Multidatabase Using InterActions and Events.

[4] E. Simon, J. Kiernan, C. de Maindreville. Implementing High Level Active Rules on top of a Relational DBMS.

[5] Henry F. Korth, Eliezer Levy, Abraham Silberschatz. A Formal Approach to Recovery by Compensating Transactions.

[6] Nandit Soparkar, Henry F. Korth, Abraham silberschatz. Failure-Resilient Transaction Management in Multidatabase.

[7] Dimitrios Georgakopoulos, Marek Rusinkiewicz, Amit Sheth. On Serializability of Multidatabase Transactions through Forced Local Conflicts.

[8] Ahmed K. Elmagarmid, Jin Jing, Won Kim. Global Commitment in Multidatabase Systems.

[9] W. Richard Stevens. Unix Network Programming.

7. Appendix.

[A] Activator

[B] Global subTransaction and Jump Vector.

[C] Local Databases and Step Library.

MONGREL - MULTIDATABASE SYSTEM

Design Document

Activator

Ming-Tsung Lu

Brown University
Providence, RI

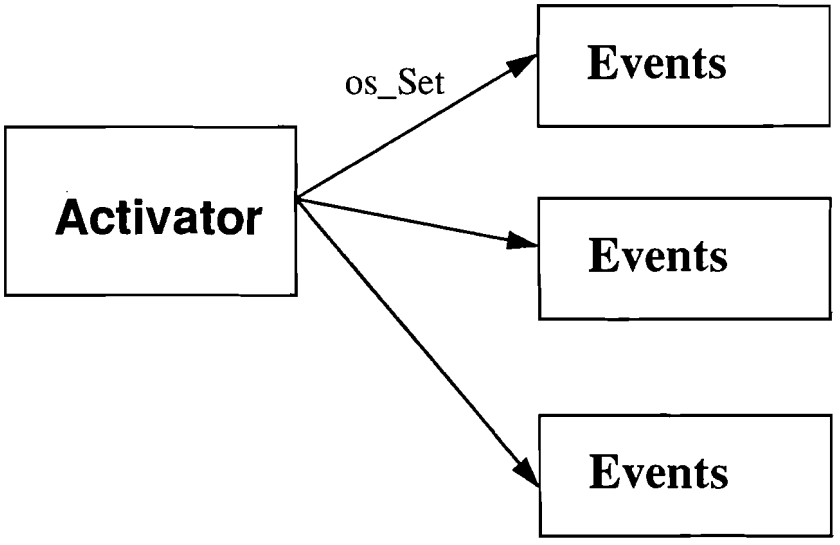
I. Introduction

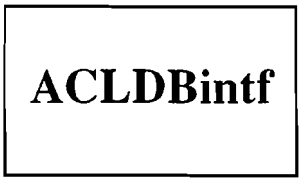
The activator plays an essential role in the multidatabase system. During the time span of a long term transaction, any outside events, such as cancellation of flight reservation, can affect the result of the execution of long-term transactions. To monitor the changes made to the local databases, we need a process which is in charge of keeping track of the changes to the databases and report relevant information to the Interaction manager.

There are three processes in this Activator system. ACcreate sets up the Activator database. This process is for creating activator only. Activ_init is an RPC server. It takes RPC calls from the Agent Manager to insert an event, remove events...etc. Activ_init is forked by Agent_init, which forks ACalarm. Since the Activator needs to poll the local database every certain period of time, it needs another process to remind it. Thus, Activ_init forks another process, ACalarm. Once the ACalarm process is created, it goes to sleep and wakes up itself every five minutes. Then it calls the Activ_init and the Activ_init does a poll on the local database. It checks to see if the conditions in the event table is violated. If any of them is, it calls the IM to notify the occurrence of the event.

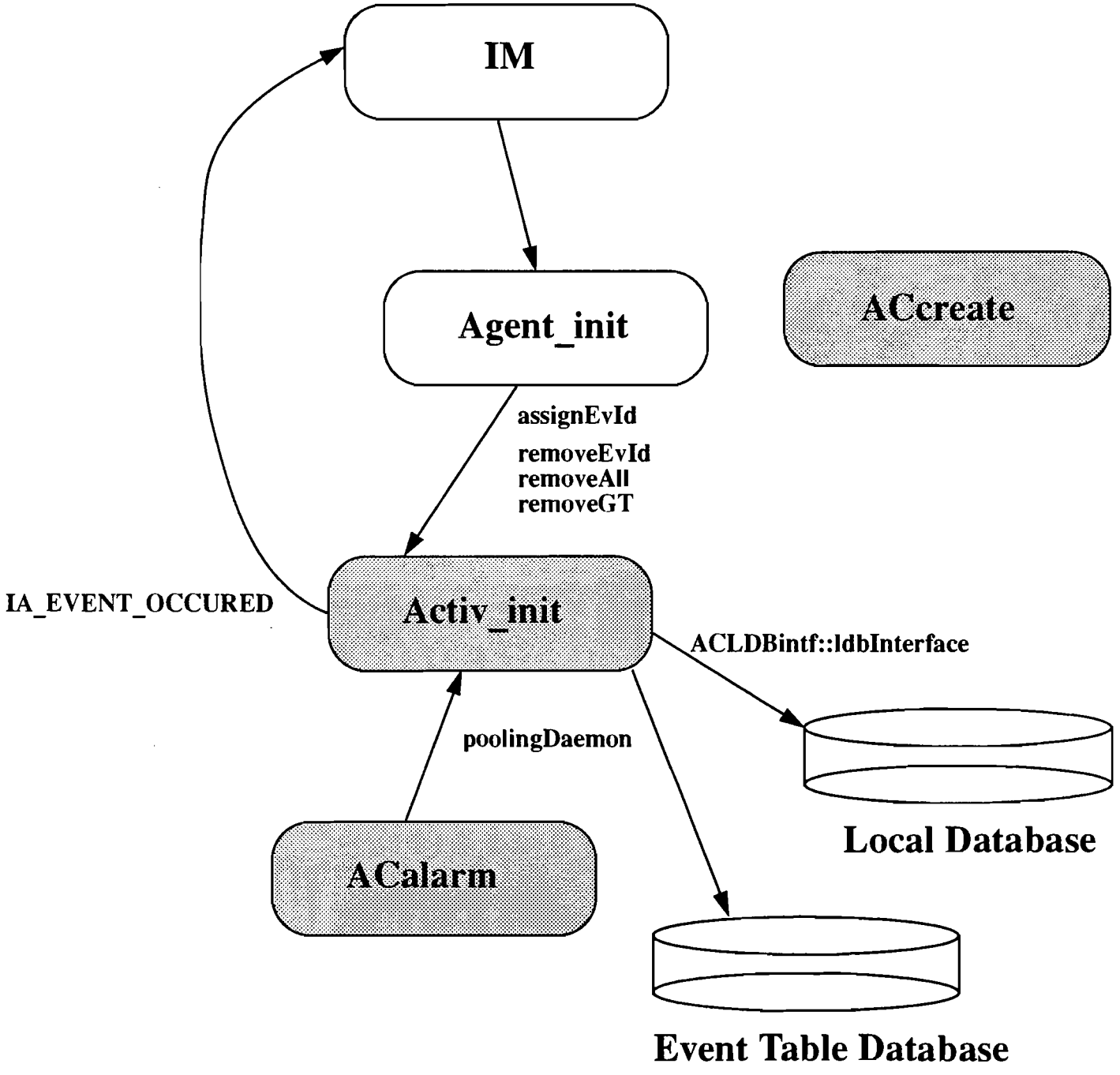
Code reusability has been considered through out the design of this system. Therefore, if we need to support Activator on top of a new ObjectStore database, all we need to do is to change the ACLDBintf.cc file in order to support that new database. None of the rest of the system needs to be changed. However, if we have to support a new database using anything other than ObjectStore, we need to change the program and use the persistence provided by the database system.

II. Class Diagram of Activator





III. Process Diagram of Activator



IV. Description of the Classes

* Class Activator

- Abstraction:

There is one instance of this class in every local database. We use ObjectStore to store the object of this class persistently in the database so that if the system crashes, we can recover the events easily. It acts as the interface between the agent manager and the event table. To declare an event in the event table, IM has to make an RPC call to the Activ_init server, which calls the methods in this class.

- Data Members:

```
os_Set <Events*> event_table;   // a set of events stored persistently.
int event_id;                   // each new event gets a new event id from this
                                // member.
```

- Public Member Functions:

```
Activator();      // constructor.
~Activator();     // destructor.
int assignEvId(ACtype, int, int, int, char**, ACcond, char*, database*);
Status removeEvId(int);
Status removeGT(int);
Status removeAll(int);
void pollingDaemon(ACLDBintf*);
void printAllEvents();
```

** Member Function:
Activator::Activator
- Semantics:
This is the constructor for the Activator class. There should be one instance of Activator per local database. It initialize the value of event_id to be 0.

- Called by:
ACcreate.cc main().

- Calls:
None.

- Parameters:
None.

- Returns:
None.

** Member Function:
Activator::~Activator
- Semantics:
Destructor.

- Called by:

- Calls:

- Parameters:
None

- Returns:
None.

** Member Function:
Activator::assignEvId
- Semantics:

This member function does the followings:

1. Assigns a unique event id and returns the event id to the agent manager at the end.
   This event id has to be unique through out the entire system. This new id can be obtained from event_id of Activator.
2. Calls EventTable::insertEvent to insert a new event in the event table.

- Called by:

```
int assignEvId(ACtype event_type, int ia, int step, int argc, char** argv,
                ACcond condition, char *check_value);
```

- Calls:

EventTable::insertEvent(int);

- Parameters:

```
ACtype event_type;  // ACtype is defined in Mongrel_types.H. It is an enum type.
                    // It can be either AC_WEAK, which stands for a weak event, or
                    // AC_WAIT, which stands for a wait event.
int ia;             // The id number of the Interaction.
int stp;            // Step id of the Activator step library. Defined in ACstep.H.
int argc;           // Argument count for the Activator step library function.
char** argv;        // Vector of arguments.for the Activator step library function.
ACcond condition;   // ACcond is an enum type defined in Mongrel_types.H It can
                    // be one of the AC_DEL, AC _GT, AC_GE, AC_LT, AC_LE,
                    // AC_EQ.
char *check_value;  // value to compare. "NULL" if condition is "delete"
```

- Returns:

int evid - a unique event id.

- Note:

The event id has to be unique through out the local database.

** Member Function:
Activator::removeEvId
- Semantics:
This member function removes the designated event from the event table.

- Called by:
Status removeEvId(int event);

- Calls:
Events::getEventId();
os_Cursor::remove();

- Parameters:
int event_id; // id of the event to be deleted.

- Returns:
Status OK if the event id is found in the event table and deleted successfully or NOT_OK if the event id can not be found.

** Member Function:
Activator::removeGT
- Semantics:
This member function removes all the events whose log_no equals to the number in the parameter.

- Called by:
Status removeEvId(int event);

- Calls:
Events::getLSN();
os_Cursor::remove();

- Parameters:
int lsn; // log_no of the events to be deleted.

- Returns:
Status

** Member Function:
Activator::removeAll
- Semantics:
This member function removes all the events of an Interaction.

- Called by:
Status removeAll(int r_id);

- Calls:
Events::getIAId();
os_Cursor::remove();

- Parameters:
int r_ia; // id of the interaction to be deleted.

- Returns:
Status OK if the ia id is found in the event table and deleted successfully or NOT_OK if the ia id can not be found.

** Member Function:
Activator::pollingDaemon
- Semantics:
This function is called by an independent function, pollingDaemon, which is called by the ACalarm process using RPC every 300 seconds. It polls the local database and checks to see if the changes in the local database are conflict with the events recorded in the event table.

If an event has occurred, it makes an RPC to IM to notify the occurrence of the event. And it checks to see what type of the event it is. It removes AC_WAIT event as soon as it occurs.

- Called by:
pollingDaemon()

- Calls:
ACLDBintf::ldbInterface(int, int, char**);
Events::getEventId();
Events::getIaId();
Events::getEventType();
RPC to IM (IA_EVENT_OCCURRED); if an event occurs.

- Parameters:
none.

- Returns:
None.

- Note:
In order to resolve the RPC compatibility problems between ObjectStore compiler and the RPC functions, this function is defined separately in the ACactivatorRPC.cc file.

** Member Function:
Activator::printAllEvents
- Semantics:
This function goes through all the elements in the event table and print out all the events. It is created for testing purpose.

- Called by:

- Calls:
Events::printEvent();

- Parameters:
none.

- Returns:
None.

* Class Events

- Abstraction:

The objects of this class are kept as a set in the class Activator. Each object represents an event.

- Data Members:

```
ACtype event_type;    // AC_WAIT, AC_WEAK. indicating whether this is a wait
                      //event or a weak conflict.
int ia_id;            // The Interaction ID. Indicates which IA this event belongs to.
int log_no;           // log sequence number of the global subtransaction.
int event_id;         // The event ID. Assigned by Activator. It is unique through out
                      // the local database.
int step_id;          // Step ID of the activator step library function. The Step ID is
                      // defined as an enum type in ACstep.H file.
int arg_count;        // argc to the Activator step library function.
char** arg_vector;  // argv to the Activator step library function.
ACcond condition;   // An enum type defined in Mongrep_types.H
                      // It can be AC_DEL, AC_GT, AC_GE, AC_LT,
                      //AC_LE, AC_EQ.
                      // AC_DEL: check to see if the data item has been deleted.
                      // AC_GT: check to see if the value  > check_value.
                      // AC_GE: check to see if the value  >= than check_value.
                      // AC_LT: check to see if the value  < than check_value.
                      // AC_LE: check to see if the value <= than check_value.
                      // AC_EQ: check to see if the value == than check_value.
char* check_value;  // This indicates the value to check for.
```

- Public Member Functions:

```
Events();          // Constructor.
~Events();         // Destructor.
ACtype getEventType();
int getIaId();
int getLSN();
int getEventId();
int getArgc();
char** gerArgv();
ACcond getCondition();
char *getCheckValue()
void printEvent();
```

** Member Function:
Events::Events
- Semantics:
Constructor for class Events.

- Called by:
Activator::assignEventId();

- Calls:

- Parameters:
```
ACtype type; // value for event_type;
int iaid;        // value for ia_id;
int lsn;         // value for log_no;
int evid;        // value for event_id;
int stpid;       // value for step_id;
int ac;          // value for arg_count;
char **av;   // value for arg_vector;
ACcond cond;  // value for condition;
char* chv;       // value for check_value;
database *ptr;    // a pointer to the ObjectStore database. The event table is stored
                  // persistently using the ObjectStore. To make the object persistent,
                 // we need a database pointer.
```

- Returns:
None.

** Member Function:
Events::~Events
- Semantics:
Destructor for class EventTable.

- Called by:

- Calls:

- Parameters:
None

- Returns:
None.

** Member Function:
Events::getEventType
- Semantics:
This function returns event_type. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
ACtype

** Member Function:
Events::getIaId
- Semantics:
This function returns ia_id. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
int

** Member Function:
Events::getEventId
- Semantics:
This function returns event_id. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
int

** Member Function:
Events::getLSN
- Semantics:
This function returns log_no. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
int

** Member Function:
Events::getStepId
- Semantics:
This function returns step_id. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
int

** Member Function:
Events::getArgc
- Semantics:
This function returns arg_count. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
int

** Member Function:
Events::getArgv
- Semantics:
This function returns the argument vector. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
char**

** Member Function:
Events::getCondition
- Semantics:
This function returns the condition associated with the event. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
ACcond

** Member Function:
Events::getCheckValue
- Semantics:
This function returns the data member check_value. It is defined as an in-line function.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
char*

** Member Function:
Events::printEvent
- Semantics:
This function prints out all the data members of an event object.

- Called by:

- Calls:
None.

- Parameters:
None.

- Returns:
None.

* Class ACLDBintf

- Abstraction:

This class is the interface between the local database and the Activator. If we need a new activator sitting on top of a different local database using the ObjectStore, the only thing that needs to be changed in the whole activator system is this class. This increases the reusability of the code.

- Data Members:

```
ActFuncPtr function_ptr[AC_STEP_COUNT]; // ActFuncPtr is defined in
                // Mongrel_types.H file. It is a function pointer. This data member is an
                // array of function pointers.
```

- Public Member Functions:

```
ACLDBintf();
Boolean ldbInterface(int, int, char**, ACcond, char*);
```

- Note:

In addition to the member functions of this class, there are several independent functions defined for this class. (See Section V, ACLDBintf.cc for a list of the functions.) They all have the same signature as

*Boolean function_name(int arc, char **arv, ACcond cd, char* cv).*

When the ACLDBintf constructor is called, their addresses are assigned to the function pointer array of ACLDBintf. These functions do the operation on the local database, get the value and compare the value with check_value cv to see if it matches ACcond cd. If it does, it returns TRUE.

** Member Function: ACLDBintf::ACLDBintf

- Semantics:

This is the constructor of the class ACLDBintf. It assigns the function pointers of the independent functions listed in Section V, ACLDBintf.cc to the function pointer array.

- Called by:

Implicitly called in pollingDaemon().

- Calls:

None.

- Parameters:

None.

- Returns:

None.

** Member Function:
ACLDBintf::ldbInterface
- Semantics:
This function is called number of events times by the Activator::pollingDaemon(ACLDBintf*); It calls the independent functions listed in Section V ACLDBintf.cc. If the function returns TRUE, it returns FALSE. If the function returns FALSE, it returns TRUE.

- Called by:
Activator::pollingDaemon(ACLDBintf*);

- Calls:
None.

- Parameters:
```
int index;          // This indicates the function id to the independent functions listed
                    // in Section V.
int ac;             // argc to the independent functions.
char **av;          // argv to the independent functions.
ACcond cmp_cond;    // Condition for comparison.
char *check_value;  // value to compare.
```

- Returns:
Boolean. TRUE if the cmp_cond and check_value condition satisfies.

V. Description of independent functions

ACalarm.C
main() - This is the main function of the ACalarm process. It is a client of Activ_init. This client wakes up itself every 300 seconds and calls the pollingDaemon() function to poll the local database.

Activ_serv_proc.C
rpc_result activ_call_1(rpc_command *proc_call)* - This function is a dispatcher function of the Activ_init server. It unpacks the parameters, make the calls to the functions defined in ACmain.cc. And it packs the result and returns the result back to the client.

Activ_server.C
*main(int argc, char** argv)* - This is the main function of the Activ_init process, which is forked by Agent_init process. It takes three parameters. argv[1] is the activator program number. argv[2] is the agent hostname. argv[3] is the agent program number. It sets up the communication channel to IM, forks the ACalarm process and enters the svc_loop when it finishes all the setup.

ACcreate.cc
main() - This is the main function of ACcreate process. It has nothing to do with the rest of the activator system. It calls the Activator constructor to create a new instance of the Activator class. Then it creates an activator database to store the object persistently. The activator database name is defined in ACTIVATOR_DB, which is defined in AC_name.H.

ACmain.cc
The functions defined in this file act as bridge between ObjectStore functions and the RPC call. The Activ_init dispatcher responds to the clients' request by calling the functions in this file. These functions open the activator database, find the root of the database, and get the Activator pointer from the database pointer. Thus it can call the member functions by using the pointer. After it calls the Activator member functions, it closes the activator database and returns the value.

*int assignEvId(ACtype event_type, int ia, int stp, int arc, char** arv, ACcond condition, char *check_value)* - This function calls the Activator::pollingDaemon member function.

Status removeEvId(int event) - This function calls the Activator::removeEvId.

Status removeAll(int r_id) - This function calls the Activator::removeAll function.

void pollingDaemon() - This function calls the Activator::pollingDaemon function.

ACLDBintf.cc

Independent functions defined in this file acts as interface functions to the local database. They all have the following format:

*Boolean function_name(int arc, char **arv, ACcond, char *cv);*

The functions open the local database, access certain value from the database, then compare the value they get with the value defined in *cv*. If the comparison matches the condition defined in *cd*, they return FALSE, else they return TRUE. There are different sets of independent functions defined for different local databases. Currently there are:

* For Car Rental Company database:

*Boolean checkCarResv(int arc, char **arv, ACcond cd, char *cv)* - It takes two parameters. arv[0] is the plate number of the car. argv[1] is the reservation id. It check the local database to see if the reservation given in the parameter exists.

* For Airlines database.

*Boolean checkPassengerResv(int arc, char **arv, ACcond cd, char *cv)* - This function takes two parameters. arv[0] is the flight number. arv[1] is the reservation id. It checks to see if the reservation given in the parameter exists in the local database.

*Boolean checkFlightCancellation(int arc, char **arv, ACcond cd, char *cv)* - This function takes one parameter. arv[0] is the flight number. It checks the local database to see if the flight exists.

MONGREL - MULTIDATABASE SYSTEM

Design Document

Global Subtransaction
Jump Vector

Ming-Tsung Lu

Brown University
Providence, RI

I. Introduction

Global Subtransaction.
The GST is the bottommost layer in an Interaction. When the Agent Manager starts a GST on a local database, a new process of the GST is created. It exists until the transaction is committed or aborted. In addition to executing the transaction on the local database, GST is also responsible of calling the functions in LRS to log essential information for later execution of compensating steps.

jumpVector.
The jumpVector is the interface between the GST and the Step Library Functions. When the GST calls the functions in the Step Library, it needs to find the addresses of the functions from the jumpVector and calls the function.

** Member Function:
GST::vote

- Semantics:
This function executes the vote protocol specific to the ObjectStore. If the result of the vote is yes, it puts the GST in the prepared state and returns a YES vote. If the database votes NO, it aborts the GST and returns NO.

- Called by:
rpc_result *gst_call_1(rpc_command*);

- Calls:
VoteResponse GST::doOSVote(transaction*);
Status GST::abortGST();

- Parameters:
None.

- Returns:
VoteResponseStruct*

** Member Function:
GST::doStep

- Semantics:
This function does the specified step in the context of the active transaction for the Global SubTransaction object. It gets the function pointer of the step library function from the jumpVector and calls the function. It sets the transaction status to TS_ACTIVE.

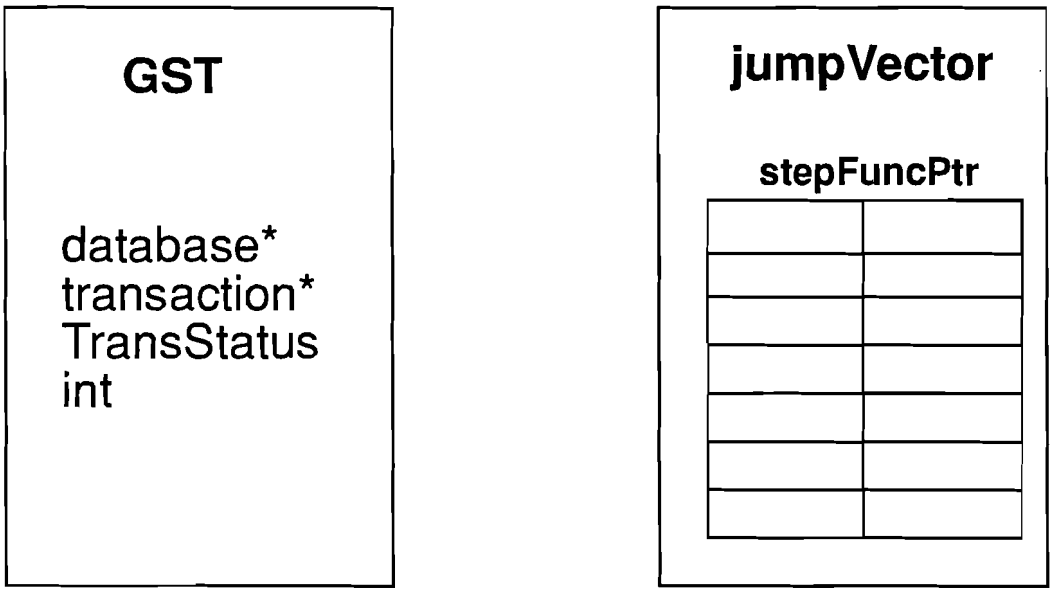
- Called by:
rpc_result *gst_call_1(rpc_command*);

- Calls:
jumpVector::getStepFuncPtr(StepIDs);
Step Library Function
step_info packStepInfo;
logStepInfo;

- Parameters:
StepIDs stepID; // step id for the step library function.
int argc; // for step library function.
char **argv // for step library function.

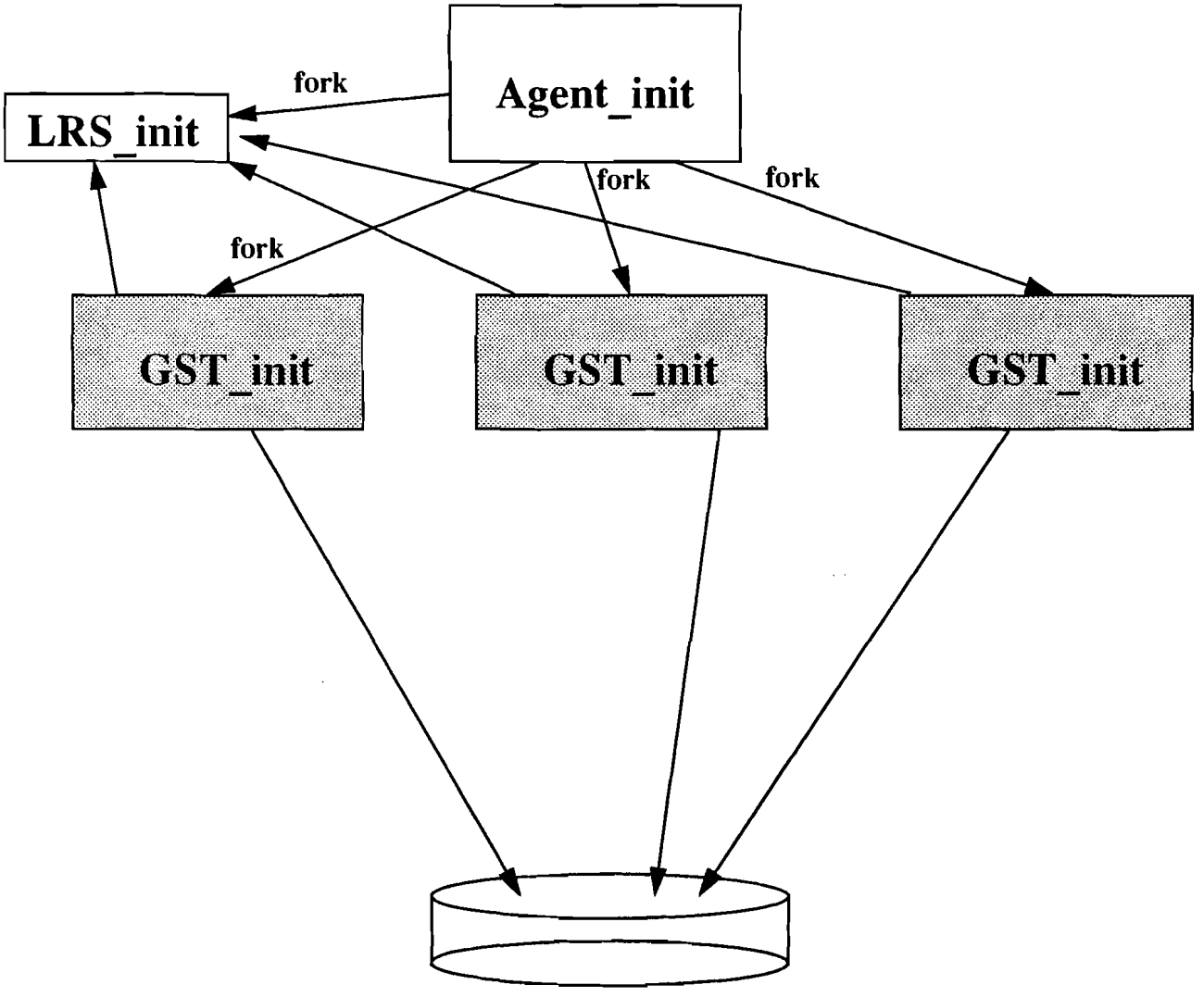
- Returns:
taslReturnStruct*

II. Class Diagram of GST



One instance of each class is created for each local database. They are created when the server starts. The pointers to the objects are stored as global variables.

III. Process Diagram of GST



IV. Description of Return Structures.

The GST does the retrieving/updating operations on the local database. It has to pass information back and forth between LRS and Agent. In order to make the information passing easier, we defined some structures which serve as return structure to the calling functions. Here is a description for the structures.

comp_info: defined in LRS_global.H
When GST calls LRS to get information for running compensating steps, this structure is returned from LRS. It contains a number of array vectors of arguments for running compensating steps.

```
typedef struct {            // Structure to return the compensation information
int LSN;
int num_rec;
int cstep_id[MAX_REC];
int argc[MAX_REC];
char *argv[MAX_REC][MAX_PARAM];
} comp_info;
```

step_info: defined in LRS_global.H
When the GTS calls LRS::logStepInfo, it packs the information it wants to log into this structure and passes it to LRS.

```
typedef struct      {       // Structure to store the step information
int step_id;                // Step Info
int step_argc;
char **step_argv;
int cstep_id;               // CStep Info
int cstep_argc;
char **cstep_argv;
return_info r_info;         // Ret Val Info
} step_info;
```

cstep_info: defined in LRS_global.H
When the GTS calls LRS::logCStepInfo, it packs the information it wants to log into this structure and passes it to LRS.

```
typedef struct {        // Structure to store the compensation step information
int cstep_id;
int cstep_argc;
char **cstep_argv;
return_info r_info;
} cstep_info;
```

taslReturnStruct: defined in Mongrel_ALD.H
GST::doStep and GST::doCStep returns this structure. This structure is returned to IM via Agent. It is ultimately parsed by the TaSL shell.

```
typedef struct {
Status  ret_stat;
int     argc;
char  ** argv;
} taslReturnStruct;
```

VoteResponseStruct: defined in VoteRespStruct.H
GST::vote returns this structure. If the result of the vote is YES, then the log sequence number, which is important to the log, is returned in this structure

```
typedef struct {
VoteResponse vote_result;
int lsn;
} VoteResponseStruct;
```

V. Description of the Classes

* Class GST

- Abstraction:

Once a GST begins on a local database, a GST server is created and an instance of the class GST is created. There is only one GST on one local database for one Global Transaction.

- Data Members:

```
database *localDB;     // a pointer to an ObjectStore database.
transaction *localDBTranID;  // a pointer to an ObjectStore transaction.
TransStatus gst_state;    // the status of transaction.
                          // This is defined in Mongrel_types.H
int log_seq_no;       // log sequence number. for LRS logging identification.
```

- Public Member Functions:

Public:

```
GST();
Status commitGST ();
Status abortGST ();
VoteResponseStruct *vote();
taslReturnStruct *doStep(stepIDs, int, char **);
taslReturnStruct *doCStep(stepIDs, int, char**);
Status doCSub(int);
~GST();
```

Protected:

```
VoteResponse doOSVote (transaction*);
```

** Member Function:
GST::GST

- Semantics:
This is the constructor of the class GST. It opens the local database, sets the get_state to TS_INACTIVE, declares the beginning of the transaction, and gets a log sequence number from LRS.

- Called by:
void GST_manager();

- Calls:
int logGSTBegin();
database::open();
transaction::begin();

- Parameters:
None.

- Returns:
None.

** Member Function: GST::commitGST

- Semantics:

This function puts the GST in commit state and completes the local commit process for this database.

- Called by:

rpc_result *gst_call_1(rpc_command*);

- Calls:

Status logGSTCommit(int);

- Parameters:

None.

- Returns:

Status;

** Member Function: GST::abortGST

- Semantics:

This function aborts the Global Subtransaction. If the GST is in either TS_INACTIVE or TS_ACTIVE states, it aborts the GST directly. If the GST is in the TS_PREPARED state, it gets the LSN for the compensating step information in the log and runs it using doCSub. If the GST is in the TS_COMMITTED state, this method should not be called.

- Called by:

rpc_result *gst_call_1(rpc_command*);

- Calls:

transaction::abort(int);
Status logAbortDecision(int);
transaction::begin();
comp_info *readCompInfo(int);
doCStep
transaction::commit(int);

- Parameters:

None.

- Returns:

Status;

** Member Function:
GST::doCStep

- Semantics:
This function does the compensating step. It is similar to doStep but logs different information.

- Called by:
rpc_result *gst_call_1(rpc_command*);

- Calls:
jumpVector::getStepFuncPtr(StepIDs);
Step Library Function
step_info packStepInfo;
logCStepInfo;

- Parameters:
StepIDs stepID; // step id for the step library function.
int argc; // for step library function.
char **argv // for step library function.

- Returns:
taslReturnStruct*

** Member Function:
GST::doCSub

- Semantics:
This function reads the LRS and calls the step library function to run compensating step.

- Called by:
rpc_result *gst_call_1(rpc_command*);

- Calls:
comp_info *readCompInfo(int);

- Parameters:
int logID; // log sequence number.

- Returns:
Status;

** Member Function:
GST::~GST

- Semantics:
The destructor. Ensures the transaction is committed or aborted. If not, aborts the local transaction. Closes the connection to the local database.

- Called by:
rpc_result *gst_call_1(rpc_command*);

- Calls:
Status GST::abortGST();
database::close();

- Parameters:
None.

- Returns:
None.

** Member Function: GST::doOSVote

- Semantics:

This is a protected member function. It commits the transaction, puts the GST in commit state and returns YES.

- Called by:

VoteResponseStruct *GST::vote();

- Calls:

transaction::commit(transaction*);
Status logPrepared(int);

- Parameters:

transaction*

- Returns:

VoteResponse;

* Class jumpVector

- Abstraction:

This class provides an interface between Step Library functions and the GST class. This class is basically a 2-D array of function pointers pointing to the step library functions. This class has to be modified if we need to support the GST operation on top of a different database using the ObjectStore.

- Data Members:

stepFuncPtr function_ptr[STEP_COUNT][2]; // 2-D function pointers.

- Public Member Functions:

jumpVector();
stepFuncPtr getStepFuncPtr(stepIDs); /* Step ptr */
stepFuncPtr getCompFuncPtr(stepIDs); /* Comp step */

- Note:

The 2-D array serves as a quick way to find the compensating step for a certain step. For example. if function_ptr[5][0] points to makeFlightReservation, then function_ptr[5][1] points to cancelFlightReservation. By doing this we make it a lot easier to find the corresponding compensating steps for step library functions. The first index of the array is defined as an enum type in stepIDs in StepID.H

** Member Function: jumpVector::jumpVector

- Semantics:

This is the constructor for the jumpVector class. It assigns the addresses of functions to the 2-D array.

- Called by:

void GSTManager();

- Calls:

None.

- Parameters:

None.

- Returns:

None.

** Member Function: jumpVector::getStepFuncPtr

- Semantics:

This function returns the function pointer to the step library function. The desired function is designated in the parameter.

- Called by

taslReturnStruct *GST::doStep(stepIDs, int, char**);

- Calls:

None.

- Parameters:

stepIDs

- Returns:

stepFuncPtr; // a function pointer.

** Member Function: jumpVector::getCompFuncPtr

- Semantics:

This function returns the function pointer to the compensating step. The desired step is designated in the parameter.

- Called by:

taslReturnStruct *GST::doCStep(stepIDs, int, char**);

- Calls:

None.

- Parameters:

stepIDs;

- Returns:

stepFuncPtr; // a function pointer.

VI. Description of independent functions

GST_serv_proc.C

rpc_result gst_call_1(rpc_command *proc_call)* - This function is a dispatcher function of the GST_init server. It unpacks the parameters, make the calls to the functions defined in ACmain.cc and packs the results returned by them. And it packs the result and returns the result back to the client.

GST_server.C

*main(int argc, char** argv)* - This is the main function of GST_init process, which is forked by AM::beginGST(). It takes three parameters. argv[1] is the GST program number. argv[2] is the LRS program number. argv[3] is the LRS hostname. It creates an instance of GST and jumpVector, sets up the communication channel to LRS server and then enters the svc_loop when it finishes all the setup.

gstCallLrs.C

The functions defined in this file act as bridge between ObjectStore functions and the RPC call. Whenever the GST methods need to make an RPC to the LRS, it calls the functions defined in this file. The functions in this file pack the parameter, make the RPC call, get the result from LRS and unpack the result returned from RPC. The functions here use three global variables defined in GST_server.C in order to communicate with LRS. They are: int GST_LRSpnum, char *GST_LRShostname, and CLIENT *GST_LRShandle. The functions are described as follows:

int logGSTBegin() - This function makes an RPC call to LRS_LOG_GST_BEGIN.
*Status logStepInfo(int lsn, step_info *sinfo)* - This function makes an RPC call to LRS_LOG_STEP_INFO.
Status logGSTCommit(int lsn) - This function makes an RPC call to LRS_LOG_GST_COMMIT.
Status logGSTAbort(int lsn) - This function makes an RPC call to LRS_LOG_GST_ABORT.
Status logPrepared(int lsn) - This function makes an RPC call to LRS_LOG_PREPARED.
Status logAbortDecision(int lsn) - This function makes an RPC call to LRS_LOG_ABORT_DECISION.
*comp_info *readCompInfo(int lsn)* - This function makes an RPC call to LRS_LOG_GST_BEGIN.

GST_serv_procStub.C

The functions defined in this file act as a bridge between ObjectStore functions and the RPC call. This stub file is created to resolve the conflict between OSCC and RPC function calls.

void callGstDestr() - calls GST destructor.
Status callCommitGst() - calls GST::commitGST().
Status callAbortGst() - calls GST::abortGST();
*taslReturnStruct *callDoStep(int stepid, int ac, char **av)* -
calls GST::doStep(int, int, char**)
Status callDoCSub(int subid) - calls GST::doCSub(int).
*VoteResponseStruct *callVoteGst()* - calls GST::callVoteGst().

MONGREL - MULTIDATABASE SYSTEM

Design Document

Local Databases and Step Library

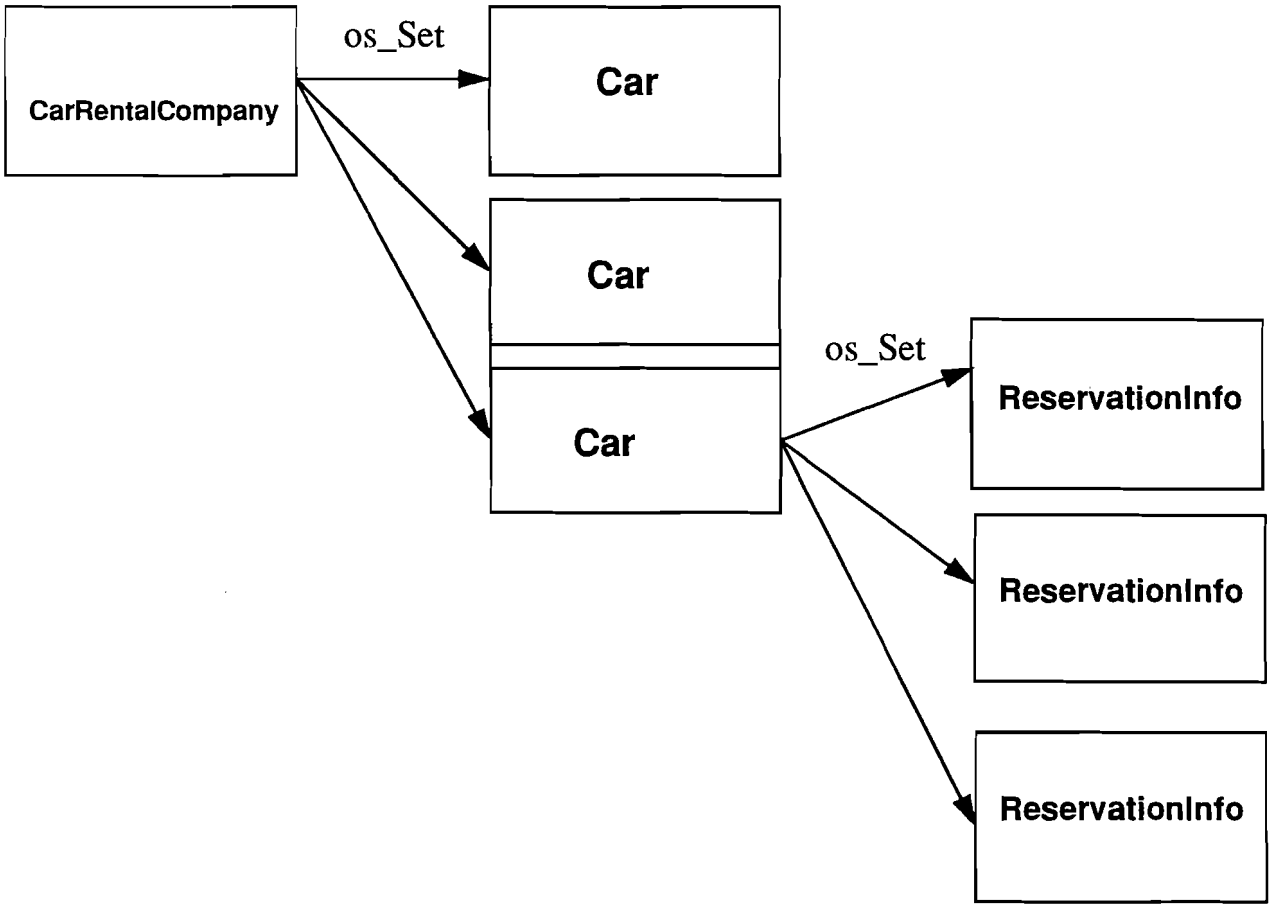
Ming-Tsung Lu

Brown University
Providence, RI

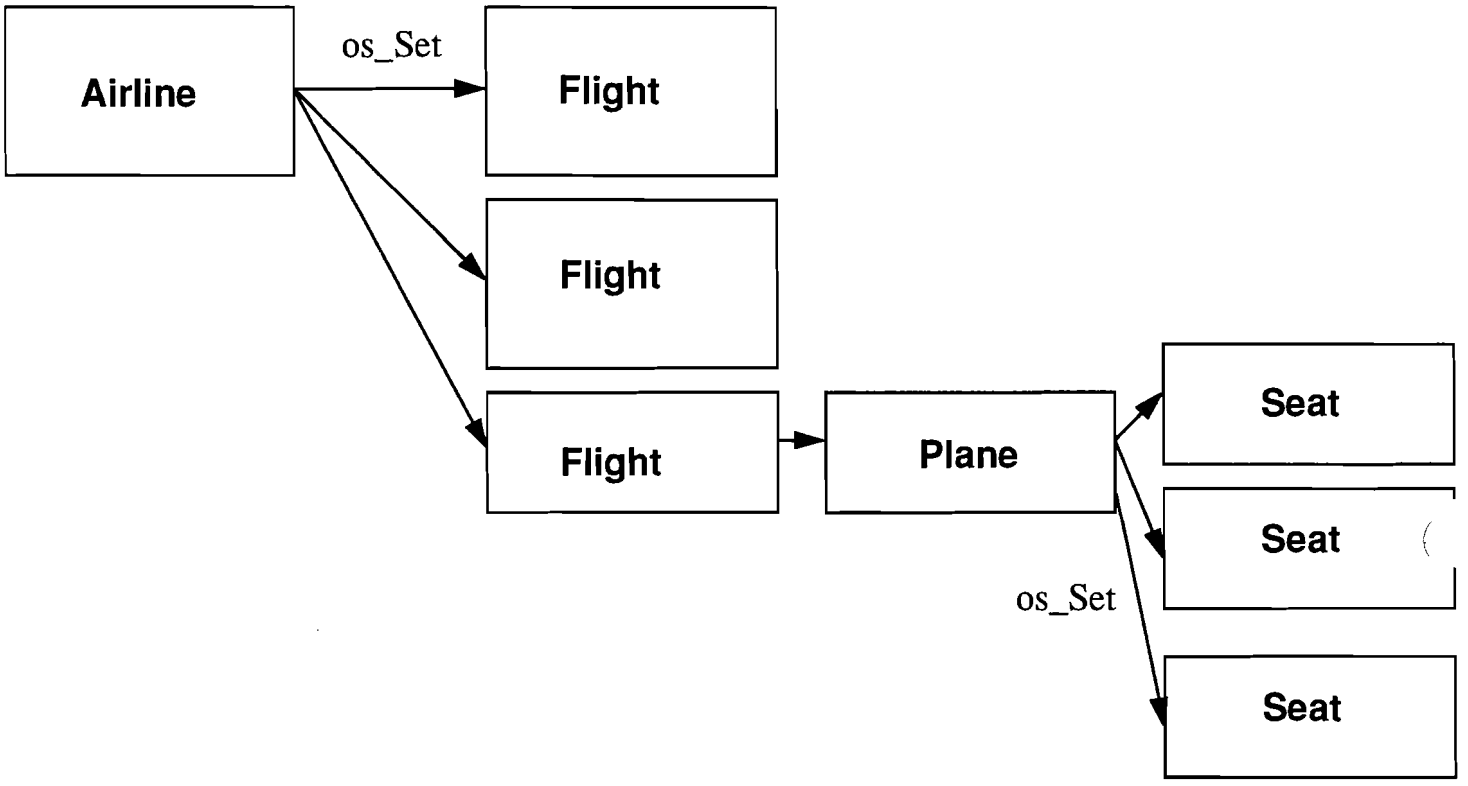
I. Introduction

In order to support the operation of the multidatabase, two sets of local databases have been created using the ObjectStore. One supports a car rental company database system, the other supports an airlines database system. Each of them have their own step libraries, which is constructed as shared libraries. The multidatabase accesses the local databases via the step libraries. The system described in this document is the bottommost layer of the entire multidatabase.

II. Class Diagram of Database Schema: Car Rental Service.



II. Class Diagram of Database Schema: Airlines Reservation.



III. Step Library Functions.
* Introduction:

Why Use Shared Library?

Step Library defines a set of functions for the multidatabase to access the local databases. In the multidatabase system, each Interaction has one and only one Global Subtransaction (GST) accessing one local database. Each GST is a process. Since in a local database, every GST has a uniform way of accessing the data. Considering the situation when there are hundreds of GSTs running concurrently on top of a local database (See figure 1), if the Step Library is not shared library, we will waste a lot of main memory space storing repeating executable code. Therefore, it is a better idea to keep the Step Library as a shared library to save space. (See figure 2.)

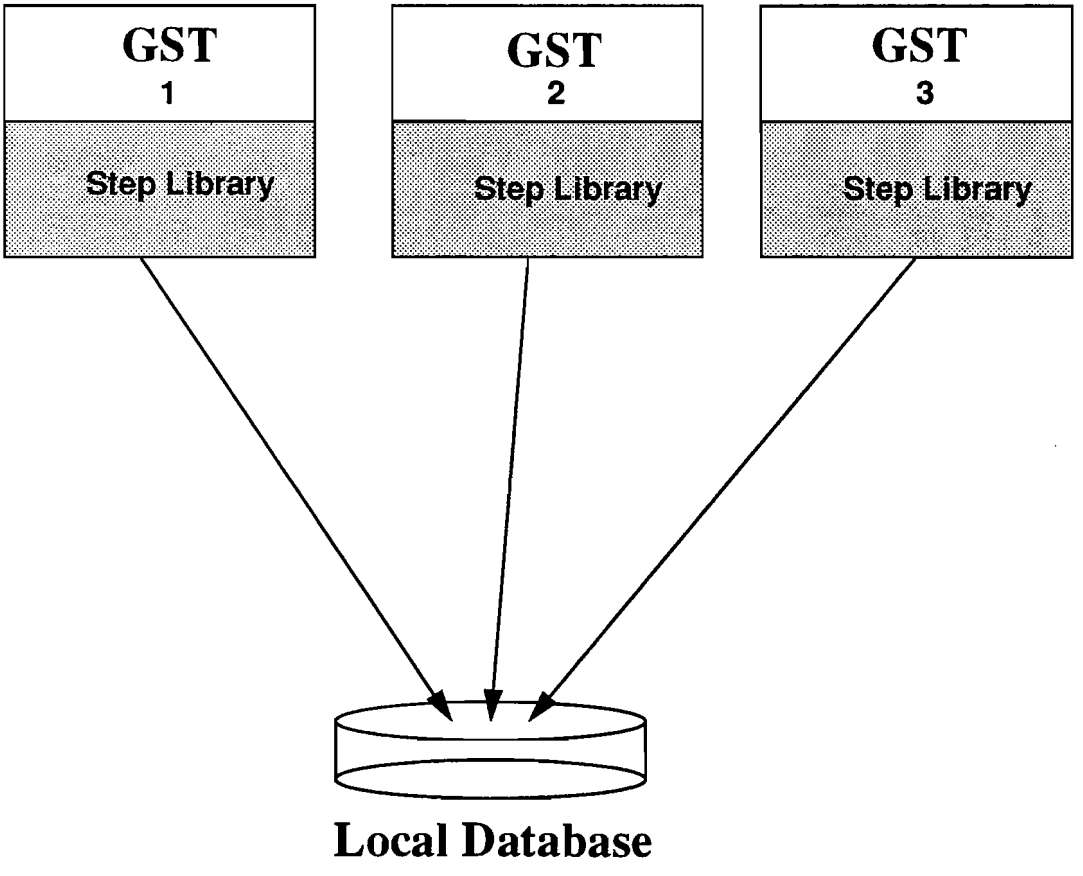


Figure 1. Step Library - None Shared Library Method.

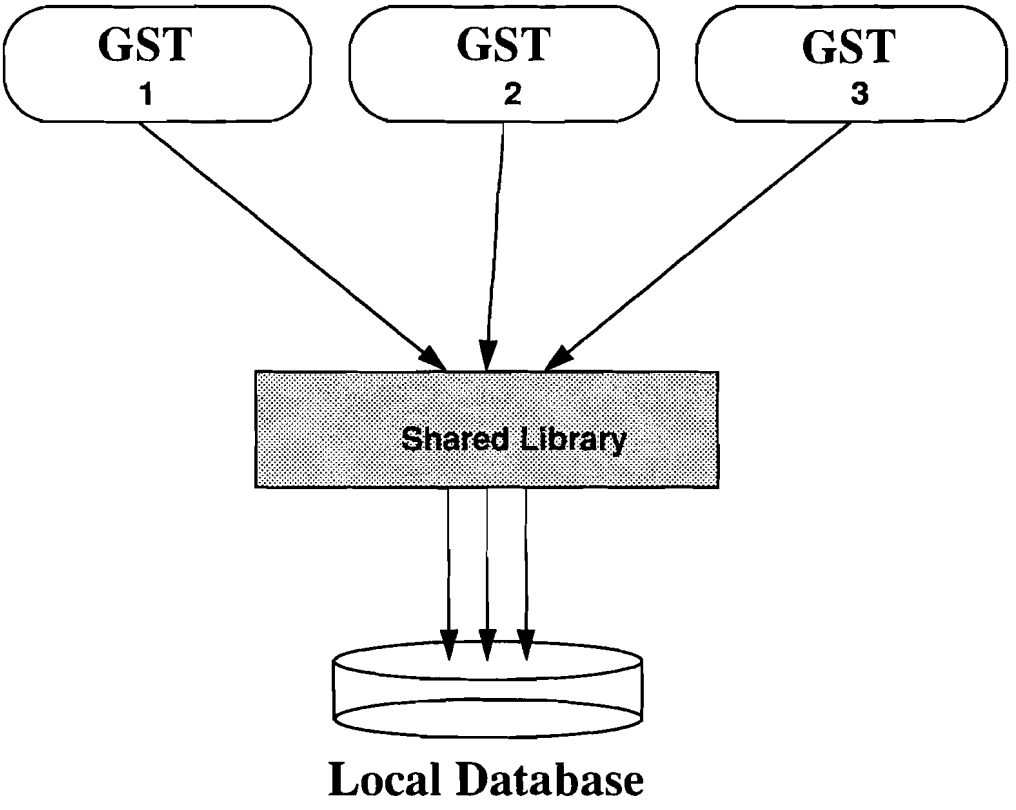


Figure 2. Step Library - Shared Library Method.

Format of Step Library Functions.

Each Step Library Function has the following format:

*stepReturnStruct *function_name(database *ptr, int argc, char **argv)*

Every function assumes the database has been opened before it is called. Thus, as the database has been opened, we get a pointer to the database which is passed to the step in the first parameter. The argc and argv are for the execution of steps.

Every function returns a structure called *stepReturnStruct*, which is defined in Mongrel_ALD.H. It has the following format:

```
typedef struct {
    Status  ret_stat;
    int     ret_argc;
    char  ** ret_argv;
    int     comp_argc;
    char  ** comp_argv;
} stepReturnStruct;
```

ret_stat indicates the status of the execution of the function. If the execution is successful, it returns an *OK*. If it is not able to complete the execution because of some natural reasons, say no cars available, then it returns a *RETRY*. If the error is caused by some programming error, say unable to open the database, it returns *NOT_OK*. In addition to the ret_stat which indicates the status of execution, we also put the error message in *ret_atgv[0]* if *ret_stat* is not *OK*.

When the *ret_stat* returns *OK*, the rest of the members in this structure return a lot of essential information for the system.
ret_argc indicates the number of values it is going to return in *ret_argv*.
ret_argv contains all the information the user needs to know after the completion of function execution. Take the airline reservation step as an example. If we book a ticket from a flight, we need to know the departure time, arrival time, seat number, reservation id of the reservation. All the information is stored and returned in the *ret_argv* vector. All the values in the ret_argv are converted to strings.

In order to support compensating operations, the arguments necessary to the execution of compensating steps are returned in *comp_argc* and *comp_argv*. The argc and argv for the compensating steps can be found in these two return values. All the values in comp_argv are converted to strings.

Compensating Steps.
The compensating steps for the step functions can be found in the jumpVector array.(See the design document for Global Subtransactions.)

Forcing Conflict.
In order to enforce conflict, every transaction has to take a "ticket" from the local database, increment the ticket and write it back to the database. This operation is supported in the takeTicket step.

* Description of Step Library Functions.
** Car Rental Service
stepReturnStruct *reserveCar(database *db_ptr, int ac, char **av)

- Semantics:
This function makes a reservation on a rental car and returns a reservation along with the stepRetiurnStruct.

- Parameters:
av[0]: name of the client.
av[1]: address of the client.
av[2]: phone number of the client.
av[3]: driver's license number of the client.
av[4]: begin date of reservation.
av[5]: end date of reservation.
av[6]: birthday of the client.
av[7]: Car Type. "SC" - Subcompact, "CP" - Compact, "MD" - Medium,
"LG" - Large,

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 5.
ret_argv[0] = reservation id.
ret_argv[1] = year of the car.
ret_argv[2] = make of the car.
ret_argv[3] = model of the car.
ret_argv[4] = plate number of the car.
comp_argc = 2.
comp_argv[0] = plate number of the car.
comp_argv[1] = reservation id.

** Car Rental Service

stepReturnStruct *deleteCarResv(database *db_ptr, int ac, char **av)

- Semantics:
This function deletes a reservation on a rental car.

- Parameters:
av[0] = plate number.
av[1] = reservation id.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 0.
ret_argv =NULL
comp_argc = 8.
comp_argv[0] = plate number of the car.
comp_argv[1] = reservation id.
comp_argv[0]: name of the client.
comp_argv[1]: address of the client.
comp_argv[2]: phone number of the client.
comp_argv[3]: driver's license number of the client.
comp_argv[4]: begin date of reservation.
comp_argv[5]: end date of reservation.
comp_argv[6]: birthday of the client.
comp_argv[7]: Car Type. "SC" - Subcompact, "CP" - Compact, "MD" - Medium,
"LG" - Large,

** Car Rental Service

stepReturnStruct *takeTicket(database *db_ptr, int ac, char **av)

- Semantics:
This function takes a ticket, increment it by one. It does this as aspect of enforcing Global Transaction serializability.

- Parameters:
None

- Return Value: (On Success of Transaction.)
NULL

** Car Rental Service

stepReturnStruct *doNothing(database *db_ptr, int ac, char **av)

- Semantics:
This function does not do anything. It is the compensating step of takeTicket.

- Parameters:
None.

- Return Value: (On Success of Transaction.)
NULL.

** Airlines Reservation

stepReturnStruct *makeReservationFlightNo(database *air_db, int argc, char **argv)

- Semantics:
Given the flight number, this function reserves a seat in the given flight.

- Parameters:
argv[0]: flight number.
argv[1]: class: "1: First class. "2": Business class. "3": Coach class.
argv[2]: seat preference. "1": window, "2": aisle, "3": middle.
argv[3]: name of the passenger.
argv[4]: address of the passenger.
argv[5]: phone number of the passenger.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 8.
ret_argv[0] = flight number.
ret_argv[1] = departure date.
ret_argv[2] = departure time.
ret_argv[3] = reservation id.
ret_argv[4] = seat row number.
ret_argv[5] = seat column number.
ret_argv[6] = arrival date.
ret_argv[7] = arrival time.
comp_argc = 2.
comp_argv[0] = flight number.
comp_argv[1] = reservation id.

- Note:
This function shares the same compensating function with makeReservationDatePort, which is deleteReservationFlightNo.

** Airlines Reservation

stepReturnStruct *deleteReservationFlightNo(database *air_db, int argc, char **argv)

- Semantics:
Given the flight number and reservation id. This function deletes the reservation.

- Parameters:
argv[0]: flight number.
argv[1]: reservation id.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 0.
ret_argv = NULL.
comp_argc = 6.
comp_argv[0]: flight number.
comp_argv[1]: class: "1: First class. "2": Business class. "3": Coach class.
comp_argv[2]: seat preference. "1": window, "2": aisle, "3": middle.
comp_argv[3]: name of the passenger.
comp_argv[4]: address of the passenger.
comp_argv[5]: phone number of the passenger.

- Note:
This is the compensating function of makeReservationFlightNo, and makereservationDatePort.

** Airlines Reservation

stepReturnStruct *makeReservationDatePort(database *air_db, int argc, char **argv)

- Semantics:
Given the travel date and departure, arrival ports, this function finds a flight which matches the description and makes a reservation on that flight.

- Parameters:
argv[0]: date of travel.
argv[1]: departure port.
argv[2]: arrival port.
argv[3]: class: "1: First class. "2": Business class. "3": Coach class.
argv[4]: seat preference. "1": window, "2": aisle, "3": middle.
argv[5]: name of the passenger.
argv[6]: address of the passenger.
argv[7]: phone number of the passenger.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 8.
ret_argv[0] = flight number.
ret_argv[1] = departure date.
ret_argv[2] = departure time.
ret_argv[3] = reservation id.
ret_argv[4] = seat row number.
ret_argv[5] = seat column number.
ret_argv[6] = arrival date.
ret_argv[7] = arrival time.
comp_argc = 2.
comp_argv[0] = flight number.
comp_argv[1] = reservation id.

- Note:
This function shares the same compensating function with makeReservationFlightNo, which is deleteReservationFlightNo.

** Airlines Reservation

stepReturnStruct *addFlight(database *air_db, int argc, char **argv)

- Semantics:
This function adds a flight record to the local database.

- Parameters:
argv[0]: flight number.
argv[1]: departure port.
argv[2]: departure date.
argv[3]: departure time.
argv[4]: arrival port.
argv[5]: arrival date.
argv[6]: arrival time.
argv[7]: plane id number.
argv[8]: plane type. "1": 737, "2":747.
argv[9]: first class price.
argv[10]: business class price.
argv[11]: coach class price.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 0.
ret_argv = NULL.
comp_argc = 1.
comp_argv[0] = flight number.

** Airlines Reservation

stepReturnStruct *cancelFlight(database *air_db, int argc, char **argv)

- Semantics:
Given the flight number, this function cancels the specific flight.

- Parameters:
argv[0]: flight number.

- Return Value: (On Success of Transaction.)
ret_stat = OK.
ret_argc = 0.
ret_argv = NULL.
comp_argc = 12.
comp_argv[0]: flight number.
comp_argv[1]: departure port.
comp_argv[2]: departure date.
comp_argv[3]: departure time.
comp_argv[4]: arrival port.
comp_argv[5]: arrival date.
comp_argv[6]: arrival time.
comp_argv[7]: plane id number.
comp_argv[8]: plane type. "1": 737, "2":747.
comp_argv[9]: first class price.
comp_argv[10]: business class price.
comp_argv[11]: coach class price.

** Airlines Reservation

stepReturnStruct *takeTicket(database *air_db, int argc, char **argv)

- Semantics:
This function takes a ticket, increment it by one. It does this in order to enforce serializability.

- Parameters:
None

- Return Value: (On Success of Transaction.)
NULL

** Airlines Reservation

stepReturnStruct *doNothing(database *air_db, int argc, char **argv)

- Semantics:
This function does not do anything. It is the compensating step of takeTicket.

- Parameters:
None.

- Return Value: (On Success of Transaction.)
NULL.