BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-93-M12

"Design and Implementation of the Logging and Recovery System for InterAction - Multidatabase Transaction Model"

by

Yashesh V. Bhatia

Design and Implementation of the Logging and Recovery System for InterAction - Multidatabase Transaction Model.

by

Yashesh V. Bhatia
B.E., Bombay University, 1991
Sc.M. Brown University, 1993

Project Report

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

March 1993

This project report by Yashesh V. Bhatia is accepted in its present form
by the Department of Computer Science as satisfying the project requirement
for the Degree of Master of Science.

Date 5/14/93

Stanley B. Zdonik

Stanley B. Zdonik

Contents

Abstract

This project report describes the working, design and implementation of a logging and recovery system for Mongrel - heterogenous, multidatabase transaction manager prototype. The prototype is based on the Interaction transaction model. Logging is primarily write ahead and follows a two level logging architecture. A checkpoint file system is used to store the intermediate temporary information for the log. Interactions provide a unique recovery mechanism consisting of traditional compensation and replacement recovery, an optimized compensation algorithm.

1 Introduction

1.1 Multidatabases and Interactions

A multidatabase is a collection of autonomous databases that are accessed to perform a specific task. Many real world applications exist that need access to multiple data repositories such as

1. Travel Agent
2. Brokerage Firm - Real time stock analysis and trading
3. Transport and Delivery Firm.

These applications are long in duration and have an element of flexibility in them. By flexibility, I mean the need to be able to change the flow of execution dynamically depending on the external environment or the state of the underlying databases.

Conventional transaction models have restricted the efficient use of multidatabases by imposing strict correctness criterion and strong recovery paradigms. These restrictions are referred to as the ACID properties (Atomicity, Consistency, Isolation, Durability). In order to effectively access multidatabase for a long duration planning task, one must relax some of the correctness criterion and recovery paradigms

Interactions [Nod92] is a transaction model for multidatabase planning applications. Formally, Interactions is an open nested, reactive and flexible transaction model for heterogenous multidatabase applications. The model provides a set of correctness criterion and recovery fundamentals that suit the requirements of the planning applications. An Interaction consists of a set of atomic subtasks called Global Transactions that have some user specified ordering amongst them. The ordering represents the execution dependencies among the global transactions. The global transactions, in turn consist of a set of global subtransactions with the following constraints

1. A global sub transaction can run on a single local database
2. Each global transaction can have only one global subtransaction on a local database.

Interactions are defined in detail in [Nod92]

1.2 Planning Applications - Example

Let us consider a Purchase order processing example to study the characteristics of a planning application.
A Purchase order is placed by a customer in order to obtain some goods from a manufacturer. The life cycle of the process is as follows.

1. Validate credit and payment modes for the customer.
2. Check Inventory for the goods required.
3. Book reservations from transport company on the specified date
4. Update the Purchase Order database
5. Update the Inventory database
6. Notify the Accounting section of the transaction.

Graphically, the task is shown below. It represents the subtasks and the dependencies among them.

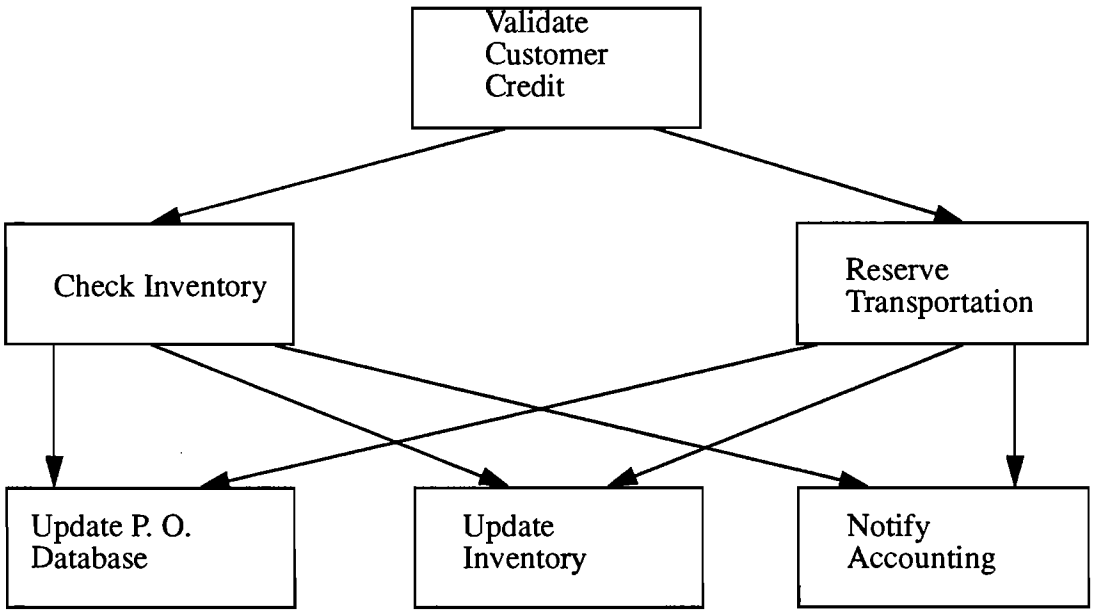


Figure 1 Planning Example - Purchase Order Processing

The application begins by validating the customer credit and ends when the goods are actually delivered. Depending on the customer needs, inventory, back order, and transportation availability the task would last anywhere from a day to a couple of months. Also during the execution, external circumstances may change resulting in a change of plans. For example, if the transport company cancels the reservation, or the inventory cannot maintain the back orders, another reservation would have to be made or the purchase order cancelled.

From the above example, some of the salient features of the application we inferred are

1. Long Duration Tasks - Typically weeks
2. Reactive - Must change if the external environment and the underlying database changes.
3. Interactive - Needs user intervention during malfunction.
4. Flexible - May change the course of execution dynamically.

1.3 Logging

Logging is used widely in database systems to maintain consistency of a database. In simple terms, logging comprises of storing the database state and transaction execution history, both of which are required to recover a database system in the event of a crash. According to [KS91] a failure may be of the following types.

1. Logical Error - Semantic problem with the transaction execution.
2. System Error - Undesirable internal state.
3. System Crash - Computer Hardware breakdown.
4. Disk failure - Loss of data due to mechanical failure of devices.

Additionally, if we consider a multidatabases, there could be failures due to

5. Communication failure - Broken link or connection
6. Site failure - remote database system crash.

Lastly, if we consifer the reactive nature of planning applications we have to provide reactivity when a conflict occurs. Conflicts occur when some actions render work done by another committed transaction inconsistent. The recovery needed for such a failure is reactivity.

In the following implementation, I address recovery in case of conflict only. This is because we are primarily interested in the semantic undo/redo of transactions and analysing recovery methods that can be used for reactivity rather than the conventional ones.

1.4 Recovery

In the previous section we mentioned the logging concept and it's functionality. Recovery is the actual procedure that brings a database in an inconsistent state to a consistent one. Log based recovery consists of the following 2 rules

1. Redo all transactions that were committed before the crash.
2. Undo all transactions that were active before the crash.

Designing a recovery system for a stand-alone database system, one can abstract the undo and

redo in terms of the read and write operations on the database. This means given the functionality to store the read and write values of all the data items, it is straightforward to recover from a crash Now, in the case of designing a multidatabase system, one cannot ignore the semantics of the planning tasks because of the nature of the applications. Specifically, in the previous example if the transport company cancels the vehicle, we must undo the work done by updating the inventory and cancelling the purchase order. These notions of semantic undo/redo are present because the sub tasks are committed on the databases, but the task is yet to be accomplished.

2. Mongrel - System Overview

Mongrel is the multidatabase system that we have implemented to simulate Interactions. The Mongrel architecture is based on the multidatabase architecture in [Nod92]. It consists of a Global Interactions Manager and the individual local database managers. The different components of the system are

1. TaSL - Task Specification Language

The TaSL is a language interface for specifying interactions, global transactions and the dependencies amongst them.

2. IM - Interaction Manager

The IM is responsible for the correct execution of the Interaction. It coordinates and executes the Global Transactions.

3. IRS - Interaction Recovery System

The IRS is in charge of the logging and recovery of complete or partial Interactions.

4. Agent

An Agent is the interface to the local level transaction management. It correctly executes the Global subtransactions. The Agent contains the Step Library which interfaces to the underlying local database.

5. LRS - Local Recovery System

The LRS is responsible for the logging of GST's.

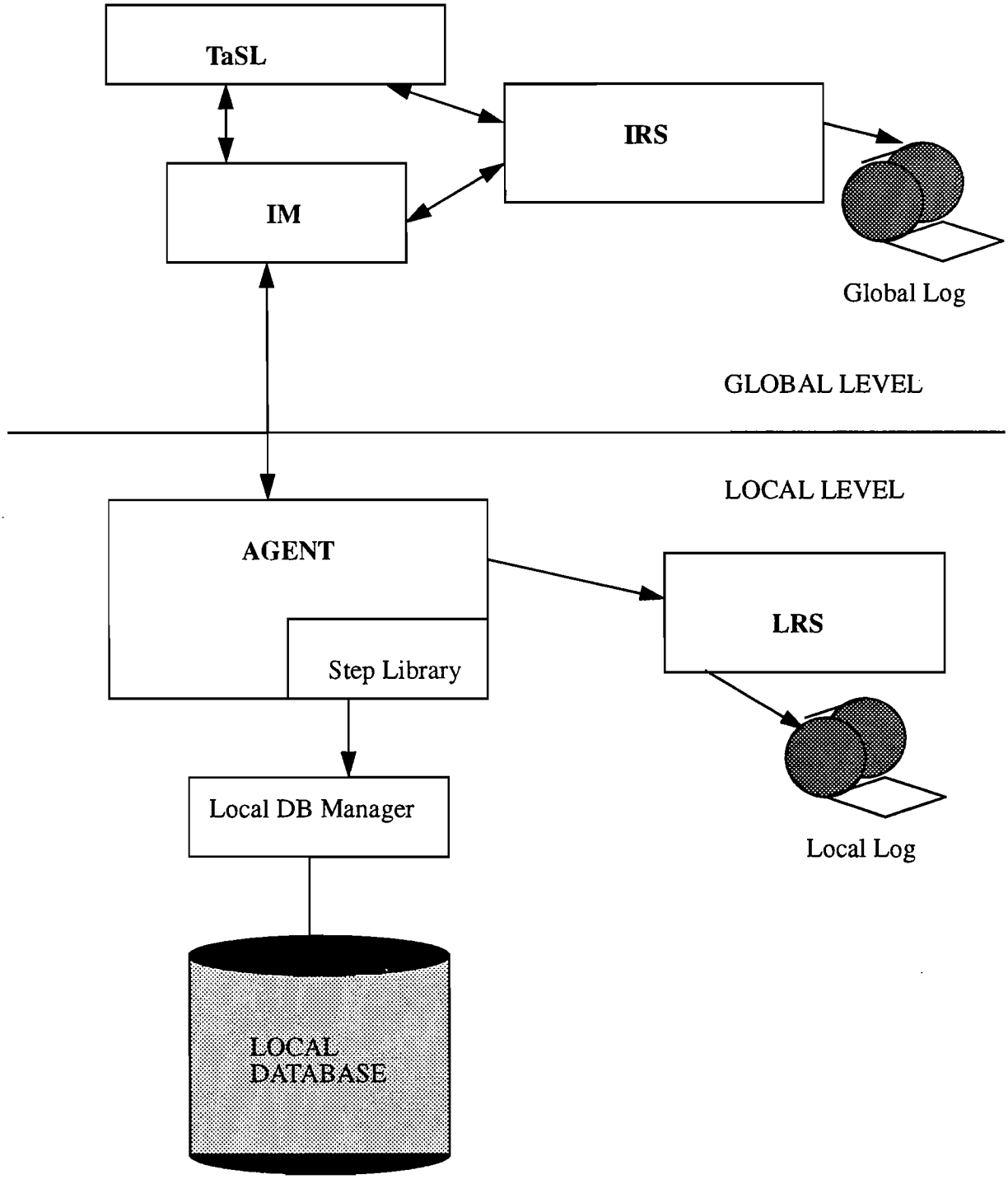


Figure 2 - Mongrel's Two Level Architecture.

3. Logging - Design and Implementation

Logging, as explained earlier, stores information in a stable storage to recover from an inconsistent state in case of failures or as in our case, of external conflicts. External conflicts as described earlier are those actions in a transaction that render work done by another committed transaction inconsistent.

Logging in a distributed system usually consists of multilevel logging. Depending on the architecture of the system, each level maintains a log of relevant data to it's level. Multilevel logging has certain distinct advantages as opposed to a single monolithic log storing data for all the levels. Its advantages are

1. Autonomy of the Logs - Each level can use its own interfaces, formats and mechanisms of logging that are appropriate for it.

2. Reduced Network traffic - Since we are using the SDWMN (Store Data Where Most Needed) paradigm, most of the operations at each level need to access only the local log, thereby reducing the overhead of network data transfer.

In the Mongrel architecture (see Figure 2) we have a two layered architecture comprising of the global and the local level. Each of the individual layers maintain their separate logs that store relevant data. Since Interaction is the transaction model used by Mongrel used by Mongrel, let us study the model more carefully with respect to the architecture. Specifically, we are interested in extracting information required to log at each level.

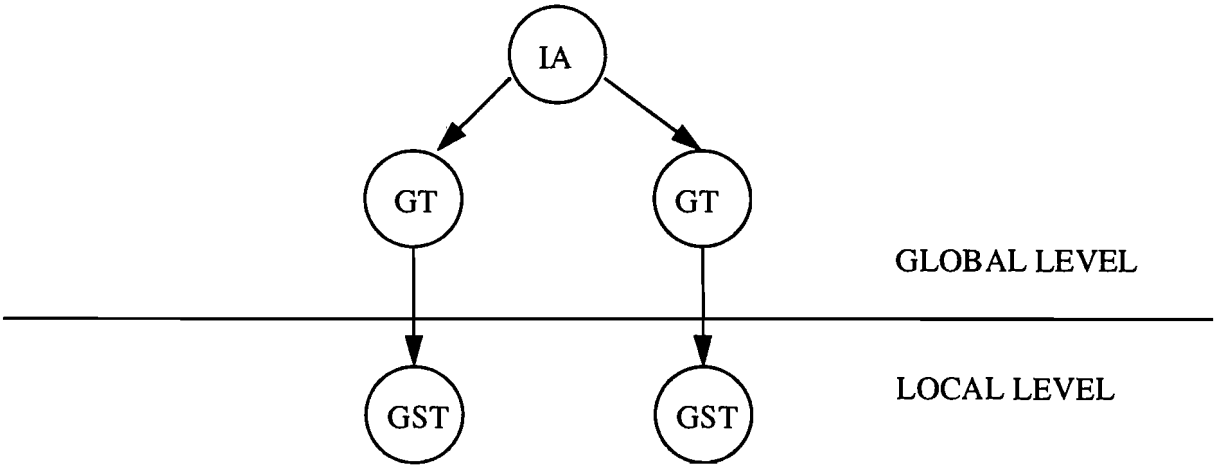


Figure 3 - Interaction Breakup at the two levels

As seen from Figure 3, the Interaction and the component Global Transactions comprise of the Global level Transactions and the Global subtransactions (GST) are the local level transaction. The following sections explain the logging of the Global and local level individually.

Let us consider the POP in Figure -2 in order to have an example for the following sections. In this example, the Interaction consists of processing a purchase order (PO) which begins when the customer places the order and terminates when the goods are delivered. Following are the components of the Interaction. The dependencies amongst the GT's are shown in Figure 2. In this example we assume the GT's are composed of s single GST and execute a single step.

IA_{POP} - Purchase Order Processing
GT_{VCC} - Validate Customer Credit
GT_{CI} - Check Inventory
GT_{RT} - Reserve Transportation
GT_{UPOD} - Update Purchase order Database
GT_{UI} - Update Inventory
GT_{NA} - Notify Accounting

The following sections explain the global and local level logging as implemented in the Mongrel system.

3.1 Global Level Logging

The global level logging system is that entity of the Mongrel architecture which is responsible for the logging and recovery of Interactions and the component Global Transactions. The system is composed of the IRS (Interaction and recovery system) and the global log where the log entries are maintained. The IRS and the global log reside on the same host as the IM. In it's true form, IRS is a process forked by the IM each time the Mongrel system is started (It's like a bootup process for the system).

The main operations and the functionality of the IRS are

1. Store and maintain all the relevant data required by the conflict recovery algorithms.
2 Provide operations to read and write data to the logs
3. Invoke the recovery daemons during the rollback and recovery phase.

Referring to the Figure - 3 it can be seen that Interactions and the Global Transactions are the only two entities that are executing at the Global Level. Hence, information pertaining to Interactions and GT's is logged. Specifically the following actions and information are logged

1. Begin and Termination of Interactions
2. Begin and Termination of Global Transactions
3. Dependencies among the Global Transactions
4. For each Global Transaction, a list of all the global subtransactions and their corresponding local database identifiers.

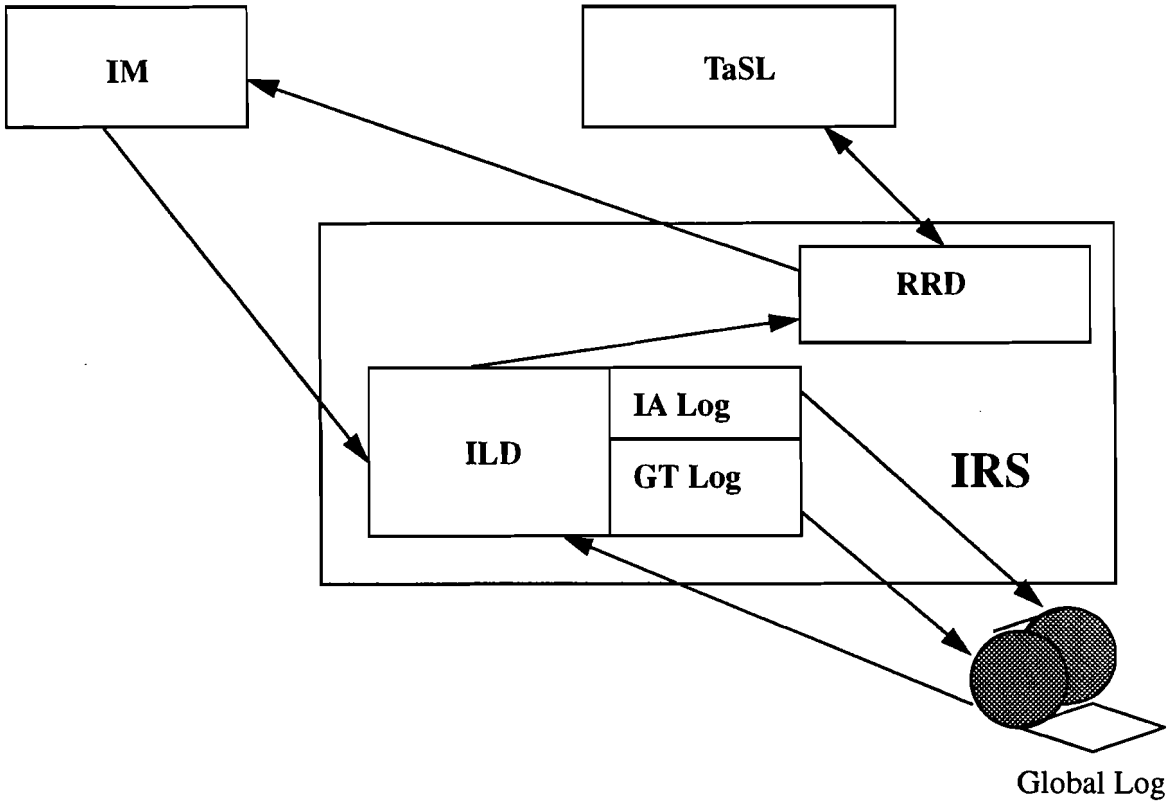


Figure 4 - IRS (Interaction Recovery System)

As seen in Figure - 4 the IRS comprises of the following entities

1. ILD (Interaction Logging Daemon)

The ILD provides the main interface of the IRS to external processes. The ILD is responsible for the complete access to the logs. At the same time, the ILD also invokes the recovery daemon in case of a transaction abort or retry. The ILD consists of a iaLog and gtLog which are used for logging interactions and global transactions.

2. RRD (Rollback and Recovery Daemon)

The RRD is responsible for providing the rollback and recovery functionality. We shall see the actual recovery process and algorithms in the next section. In it's physical form the RRD is a process forked by the ILD with the sole intention of performing recovery of the transactions specified by the ILD. The actual processes and data structures used are explained later in the recovery section.

The RRD is one of the key components since the primary forces of the Mongrel architecture is to

study new algorithms for recovery. The RRD communicates with the IM and the TaSL to achieve its goal of recovery. The RRD requests the IM during compensation and the TaSL during reexecution. The exact nature is defined in the next section. The RRD needs to read the Logs too for information of the dependencies and the GST items and has a reading interface with the ILD.

The logs are maintained at the same level as the IM (and the IRS). We use the UNIX file system to store the data in the log. Since our primary focus was on other issues rather than providing a completely failure safe recovery, we choose to use the UNIX file as stable storage based log and a hierarchical file system to store the temporary data pertaining to the Inter actions and the GT's. Entries to log are accessed using the UNIX file I/O routines and hence is insulated from concurrent usage.

During initialization, if the log is not present one is created and an initial "LOG STARTED" record is entered into the log. Note, all the log entries are appended with timestamps. They do not serve any purpose with the current implementation, but would be of a significant importance dealing with designing overall recovery. The logs are then updated as the interactions and GT's begin to execute. Each time a new Iinteraction or Global Transaction begins, a record for IA Begin and GT begin is entered.

In order to store temporary log data for the active IA's and GT's directories are created with the interaction identifiers (Interactions and Global transactions are uniquely identified by IAID and GTID respectively). The hierarchical checkpoint structure looks like:

/IAID/IAID.depinfo
IAID.active
/GTID/
.
.
.

The temporary data is removed as soon as the global transactions or Interactions are committed. The commit procedure therefore consists of the flushing of specified subdirectories and storing the entries in the log. Finally the "IA Commit" or "GT Commit" are written to the log. Once the interaction is committed, entire subdirectory is flushed into the log and then deleted. The checkpoint (hierarchical) file system is very convenient for the following reasons:

1) Provides Locality:
Since all the log records pertaining to a IA or GT are written to the log consecutively, they provide an element of locality which can be exploited for optimizing log access.

2) Painless abort Mechanisms:
Since the data for active IA's and GT's is never entered into the log, aborting in active GT or IA is simply reduced to removal of files from the hierarchical checkpoint file system which would otherwise be done by scanning entries in the log and undoing them.

Note - This method has a disadvantage in the sense that it can't be used for providing complete

failure recovery.

Log Format - Each entry in the log consists of the following fields

1) Time Stamp - time of logging
2) IAID
3) GTID
4) Record Type - type of operation
5) Log Entry - String of information

A typical log entry would look like

10432187 IAID GTID 1 "COMMITTED"

The above entry stores the commit of the GTID of IAID.

3.2 Local level logging

The local level logging system is that which is responsible for information required to maintain local consistency. The system is comprised of the LRS and the local logs where entries are logged. The LRS and local logs are maintained on the same host as the local database that is a part of the multi-database system.

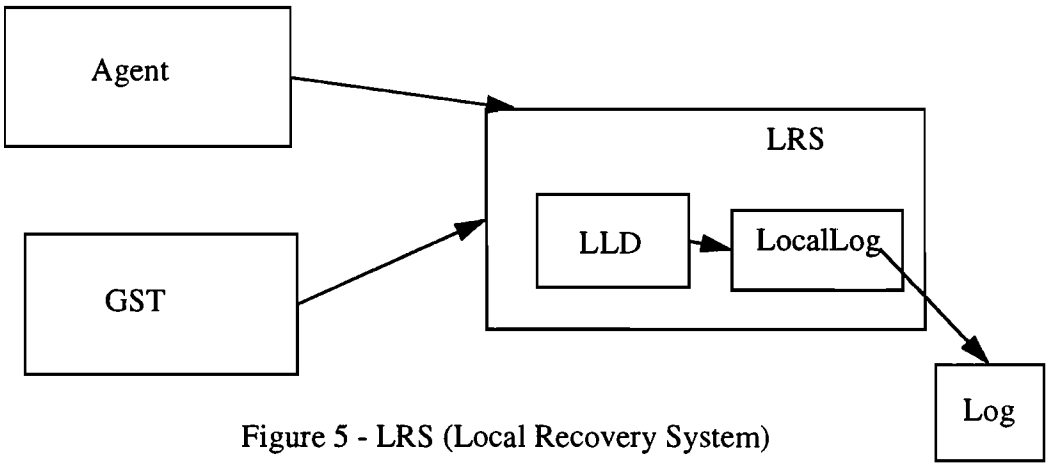


Figure 5 - LRS (Local Recovery System)

As seen from the figure 2 each local database is connected to the multi-database via an agent and the step library. The agent creates the LRS during initialization. The LRS is essentially a server and in order to communicate with it, a process needs a communication handle. The LRS sends its handle to the agent during

initialization.

The agent distributes this handle to all the GSTs which are the primary interface to the LRS.

The LRS is comprised of the following modules:

1) LLD (Local Logging Daemon)
The LLD is the local level logging Daemon and presents the main interface for the LRS. It is responsible for total access to the logs. The LLD calls the local level logging which does the actual physical logging. The abstraction of local log is present in order to insulate the basic log ging functionality from the optimization. By this I mean that the local log could provide an optimized logging strategy that would be insulated by the LLD.

Local Log

The local log provides the actual physical access to the logs. As stated above, it is provided to insulate the physical access from the logical access of the logs.

The local logs are maintained at the same level as the agent. UNIX file system is used for the storage of the log entries as well as the temporary data.

The primary interface to the LRS is the GST. Right from the very begining to the end, the GST logs its action and the information needed to restore the local database.

The communication between the GST and the LRS begins as soon as the GST is created and is via the handle that it receives from the agent. It first requests the logging of a "Begin GST" record. The LRS plays a crucial role here by assigning global sub-transaction identifier (LSN (log sequence Number)) to each new GST. The LSN uniquely identifies a GST on a given host and is used as the key field in further communications.

Steps

Steps are a sequence of operations that act upon the local database. Steps are connected to the host database operations via a table called the step library.

As noted earlier a GST consists of a sequence of steps which it executes after it commences. As soon as the steps are executed, they are logged by calling the LRS. Logging of the steps and its corresponding compensating step are one of the essential actions of the local log. It is important from the perpective of the recovery, because its the compensation of the transaction.

The following information is logged with all the steps

Step ID
argc
argv

Comp Step ID
argc
argv
Return Status
argc
argv

Note the field are delimited by "@". An example log entry would look like

"DECREMENT INVENTORY@2@SPARC 2@100@INCREMENT INVENTORY@2@S-PARC 2@100@OK@0"

The following step decrements the inventory for the Sparc 2 by 100 pieces and the compensating step would increment the inventory of Sparc 2 by 100.

During compensation, the GSTs read the comp information from the logs through the LRS. After reading the compensation step info, executes the step on the local databasae (in inverse order)

After executing all the GSTs participate in a 2 phase commit for the global transactions. During the 1st phase of 2pc if the GST agrees to commit then the GST is commited to the local database and enters a prepared state. During the prepared state the LRS commits the GST in a LDB. At this stage, there are just two options for the GST:

1) Abort Commit Decision : may occur if all GST's cannot commit in which case the compensat ing steps are fired to undo the effects of the step.

2) Commit GST -The GST is committed because all the other participating GSTs were ready to commit.

Logging and Checkpoints

The local log is maintained at the same level as the LRS and the agent.

During initialization, an init record is written to the log. Success records are written to either the log or files in the temporary checkpoints file system. The checkpoints file system maintains temporary data for a specified LSN. The file is created when the LSN is assigned and destroyed when the GS9 enters the prepared state. The files are stored as:

/LSN_files/1
    2
    .
    .

Information in the log is stored as follows:

Timestamp - Timestamp of the operation
LSN - Log Sequence Number
Rec Type - Type of record
Log Entry - String of information

3.3 Recovery - Design and Implementation

The Interaction transaction model provides unique and optimized recovery mechanisms for semantic Undo/Redo of committed transactions. These transactions are of long durations, flexible, reactive and interactive by nature and hence are difficult, and in some cases impossible to recover from.

Compensation is the primary form of recovery in the interaction model. According to [KLS 90] "A compensating transaction is one which undoes the effects of a transaction in a semantic manner, rather than by physically restoring to a prior state".

Using the example in figure 2, if the transaction GT_{UI} consists of decrementing the quantity of required goods then its corresponding compensating transaction would be the incrementing the quantity of the required goods.

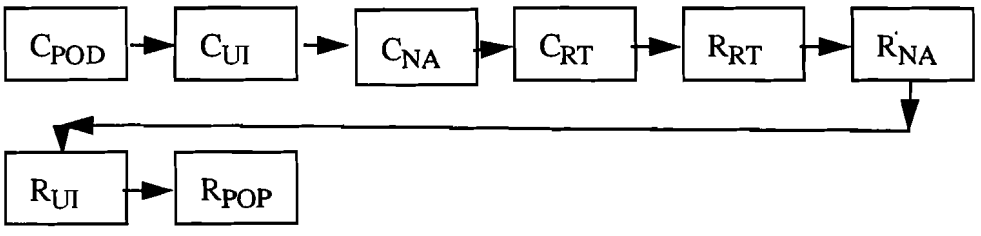
The semantic information needed to compensate a transaction can be acquired by either

1) Extracting compensating actions from the program of the compensated for transaction by examining the database and the logs, or

2) Having the user predefine a compensating transaction for each transaction.

For applications such as the example in figure 2 pure compensation is not the perfect method of recovery.

For example, if the transport company cancels the deliveries of the vehicle then using the compensation based method we could have the following scenario

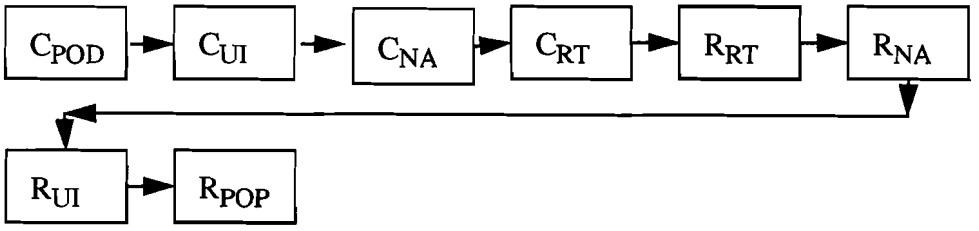


Note all the transactions dependent on GT_{RT} are compensated for in the reverse order and are then redone in the forward order. This sequence of operations could cause undesirable results. For example, after compensating for the reserve transportation, we redo the transaction by confirming a new reservation from a new company. However, in the meantime since we have released control over the inventory database, any other transaction could update the inventory (taking the goods) thereby disabling the Redo of the update inventory transaction. Also, we can notice that unnecessary work is being done in undoing and redoing the transactions when we get the reservation for a new vehicle.

To overcome the above problems Interaction provides an alternate algorithm for recovery - Replacement recovery. The algorithm is described in detail in [NZ 92], however I shall present a brief overview of it.

According to [NZ 92], the fundamental concept of replacement recovery algorithm is to replace any work in an invalid transaction without modifying the work done by the dependents of the invalid transaction.

Falling back to our example, since the cancellation and reservation of the transport vehicle has no effect on the other transactions (if re-reserved on the same day), then the sequence shown below would be equivalent to the one above.



As seen from the diagram the compensate and redo transactions are moved up in the execution sequence. Furthermore if the compensate and redo are inverse of each other then they could be eliminated totally without execution.

Continuing with the example, the transaction notify accounting is moved up the sequence and since C_{NA} and R_{NA} are inverse, they are eliminated. Same is the case with C_{UI} and C_{POP}.

Implementation

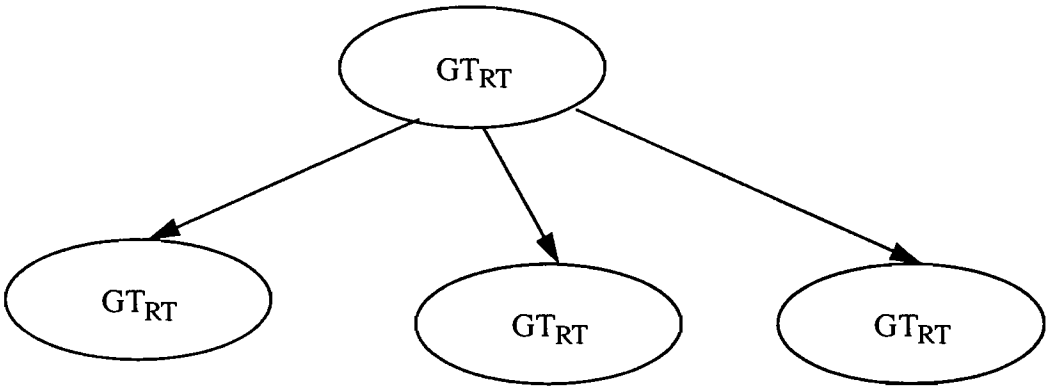
In the current version of Mongrel, we have the compensation mode completely running. Initial design has to be done for the replacement recovery mode, however, it has yet to be implemented.

The recovery process is initiated when the IM requests the abort a retry of a global transaction or interaction. The request for the abort (or retry) may be for any of the following reasons:

1) The underlying state on which the transaction depends has been violated (note:- this is implemented in the form of events and weak conflicts. For more details refer to [Nod 92])

2) User specifies an abort due to external malfunctions.

Upon getting the request for Abort (retry), the IRS creates a DAG (Directed Acyclic Graph) to store the invalid transaction and its dependents. For example, the DAG would be like



The DAG is created using information stored in the dependency information file. The dependency information is stored in the following format

TO-GT Dep-Type Num_Parents From-GT's.....

where
TO-GT is the GTID of the dependent transaction
Dep-Type is the type of dependency (S/E)
Num-Parents is the number of parents
FROM-GT is the list of GT's "TO-GT" is dependent on.

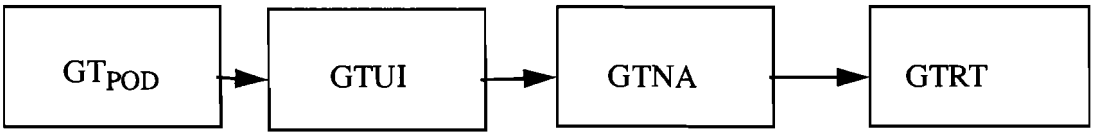
where the dependency could be a state or execution dependency (S/E).

A typical entry would be

GT_{POD} E 1 GT_{RT}
GT_{UI} E 1 GT_{RT}
GT_{NA} E 1 GT_{RT}

The DAG representing the complete set of transactions and their partial order is to be given to the RRD (Rollback and Recovery Daemon) for recovery.

The RRD first topologically sorts the DAG to give a list of transactions. For example the list in our case would be like



The generated list of GT's now needs to be compensated for.

Compensation for any transaction commences by requesting a begin of global transaction to the IM. After the new GT has begun the list of global subtransactions (LSN) and their corresponding host names are obtained. This information is obtained by reading their log and selecting the desired entries.

Next compensating steps are fired at the IM for individual global subtransactions. The IM request the individual agents to perform the compensating steps. The above procedure is repeateed for all the transactions in the list. The compensating for the final transaction terminates the recovery pro cess.

4. Overview of Imlementation

As is the case with all other projects, there were problems, solutions, and trade-off made to realize it practically. Some of the problems I encountered are -

1. Communication with the IM

During the recovery of the committed transactions the RRD needs to communicate with the IM to accomplish compensation. Since the IM initiates the recovery, hence the IM process remains blocked thereby preventing input requests. A complete solution here would have been to have an

asynchronous protocol between the IM and the RRD. However, due to the time constraints and an already jeopardized communication problem, we simplified it by using synchronous protocols. The solution is by no way incorrect because in the proposed model, it is the communication between the TaSL and RRD that is significant. I would write this off as a trade-off rather than an abnormality.

2. Creating the DAG

Another difficult problem was that of creating the DAG from the dependency information stored in the global level. This is because the dependency graph is stored in an inverse format at the global level. The format could have been reversed, but since the data is dynamic, storing it in the form of "From_Egde To_Edge" seemed to be a viable solution. However, this causes multiple scans of the files and hence a very inefficient algorithm. I could not find a suitable solution for this.

Further Work

There are a couple of areas where, work needs to be done

1. Integrating the RRD with the TaSL - This would capture true reactivity
2. Integrating the IM and IRS as a single process - This would reduce the RPC traffic and make the system more robust due to lesser RPC communication.
3. Adding complete recovery features to enable total failure safe system.

Bibliography

[KLS90] - "A Formal Approach to Recovery by Compensating Transactions" - Henry Korth, Elizier Levy, Abraham Silberschatz. Proceedings of the 16th VLDB Conference - 1990.

[KL91] - "Database Systems Concepts" - Henry Korth and Abraham Silberschatz. Database Text Book, 2nd Edition.

[Nod92] - "Supporting Long running tasks on an Evolving Multidatabase using Interactions and Events" - Marian Nodine. PDIS 1992

[NZ92] - "Supporting Reactive Planning Tasks on An Evolving Multidatabase" - Marian Nodine and Stan Zdonik. Tech Report 92-59, Department of Computer Science Brown University,

Appendix I

The following section describes in detail, the various data structures, modules and classes used by the Logging and Recovery system. The Logging and Recovery System consists of 2 entities, Global and Local level and are described in terms of their modules, data structures and classes.

Global Level Logging

Modules

The are 2 modules associated with the global level logging.

1. IRS server

This module is invoked as a separate process by the IM. The module acts as a server for global level logging. It creates the communication sockets, performs the initialization/registration and listens for requests from other entities of the system.

Upon getting a request, the server transfers control to the IRS dispatcher which unmarshals the command using the system specified rpc routines. The dispatcher then makes the appropriate calls to perform the desired action. The results of the actions are marshalled and returned to the calling process.

Note - The server blocks the calling process and communication is via RPC using customized routines. (A set of standardized routines are available for communicating with entities within the Mongrel MDBS)

2. Recovery Daemon

The recovery daemon is invoked by the ILD if an Interaction or Global Transaction needs to be retried or aborted. The daemon is spawned off as a separate process as it needs to communicate with the IM. The daemon essentially calls the right function in the RRD, which is in charge of the recovery.

Arguments - IAID if an Interaction, and IAID + GTID if a Global Transaction, needs to be retried or aborted

Data Structures

The following data structures are used by the Global level.

1. Log_rec

It stores the all the log records of a Global Transaction.

```
struct Log_rec                          // Log record format
{
 int reccnt;                            // Record Count
 int record_time[MAXREC];               // Timestamps
 int IAID;
 int GTID;
 int record_type[MAXREC];
 char *log_entry[MAXREC];               // Log Entry
 int count[MAXREC];
}
```

2. Active_rec

Active_rec stores information of all the active records

```
struct Active_rec
{
 int record_time;
 int IAID;
 int GTID;
 char *text;
}
```

Classes

The global level logging consists of the following relevant classes which are described in the Appendix

1. ild
2. iaLog
3. gtLog
4. RRD
5. RRDDAG

For each class, important instance variables and member functions are examined in detail. The classes are examined on their functionality basis i.e. the member functions are grouped into logical components.

1. Class ild (Interaction Logging Daemon)

This class performs all the duties of Logging system. It's main functionality is

1. Provide an interface to the IM (Interaction Manager) for logging the various phases of Interaction execution.
2. Maintain the global log and the temporary data.
3. Invoke the recovery daemon when an Interaction (partial or complete) needs to be recovered or retried.

1.1 Interface to the IM.

The IM has the maximum interaction with this class. Specifically, it requests logging and recovery of the Interaction execution. The main functions through which it can access the ILD are

1.1.1 Status ild :: abortIa(int IAID)

Semantics - This routine aborts an InterAction. It first generates a DAG, which is passed to the RRD. The RRD aborts/compensates all the GT"s present in the DAG. The result is then logged into the MASTER_LOG and finally the subdirectory of the IAID is removed.

Called By - Routines in the IM.

Calls - RRD::rollBckGtRrd()
iaLog::logIAAbort()
ild::deleteDir()
ild::getRRDDAG()

Parameters - IAID - InterAction ID

Returns - Status OK/NOT_OK

Effects - Aborts an InterAction

1.1.2 Status ild :: abortGt(int IAID, int GTID)

Semantics - This routine aborts the specified Global Transaction and all those that are dependent on it.

Called By - Routines in the IM.

Calls - RRD::rollBckGtRrd()
ild::getRRDDAG()

Parameters - IAID - InterAction ID
GTID - Global Transaction ID

Returns - Status OK/NOT_OK

Effects - Aborts a GT

1.1.3 Status ild :: retryIa(int IAID)

Semantics - This routine retries the specified IA. The DAG representing the Interaction in terms of the GT's is first generated. The DAG is then passed to the RRD for retrying.

Called By - Routines in the IM

Calls - ild::getRRDDAG()
RRD::rollBckGtRrd()

Returns - Status OK/NOT_OK

Effects - Retries the entire Interaction.

1.1.4 Status ild :: retryGt(int IAID, int GTID)

Semantics - This routine retries the specified Global Transaction and all it's dependents. The DAG representing the GT is first generated. The DAG is then passed to the RRD for retrying.

Called By - Routines in the IM

Calls - ild::getRRDDAG()
RRD::rollBckGtRrd()

Returns - Status OK/NOT_OK

Effects - Retries the Global Transaction and it's dependents.

1.1.5 Status ild :: logIMException (int, int, REC, Err_Code)

Semantics - This routine logs Exceptions to the MASTER LOG. Exceptions could be of various types and are identified by an exception identifier.

Called By - Routines in the IM, ild and the RRD.

Calls - ild::writeLog()

Parameters - int IAID
int GTID
REC rec_type - Identifies the caller
Err_Code err_code - Identifies the exception

Returns - Status OK/NOT_OK

Effects - Logs an exception record.

1.1.6 Status ild :: logDEpendency (int IAID, int GTID, char*)

Semantics - This routine writes dependency information to the IAID.depinfo file.
To see the format of the IAID.depinfo file see the project report.

Called By - Routines in the IM.

Calls - ild::checkFile()

Parameters - IAID, GTID, Dependency Info

Returns - Status OK/NOT_OK

Effects - Adds an entry in the /pro/mdb/ild/global_log/IAID/IAID.depinfo

1.2 Interface to the RRD

The RRD (Rollback and Recovery Daemon) also has a high degree of communication with the ILD. It needs to read the MASTER LOG and make log entries during the process of recovery and retrying.

1.2.1 Log_rec* ild :: readLog(int IAID, int GTID)

Semantics - This routine reads all the records for the Global Transaction specified by the GTID. It scans the entire log and filters those records that are required.

Called By - RRD::compTrans()

Calls - File IO routines to scan the log.

Parameters - IAID, GTID.

Returns - Log_rec* - Structure to store the log information.
(The structure is described in detail earlier)

Effects - Allocates memory for the Log_rec and returns the pointer after reading the information.

1.2.2 TransStatus ild :: verifyGTstatus(int IAID, int GTID)

Semantics - This routine returns the current status of an Interaction or a Global Transaction. It does by checking the temporary data storage structure.

Called By - RRD::rollBckGtRrd()

Calls - Directory maintenance routines

Parameters - IAID, GTID

Returns - Transaction Status - TS_NOT_OK
TS_ACTIVE
TS_COMMITTED

Effects - Returns the status.

1.2.3 Status ild :: deleteDir(int IAID, int GTID)

Semantics - This routine deletes the sub directory that stores the temporary data for a given Interaction or a Global transaction.

Called By - ild::abortIa()
gtLog::logGTCommit()
iaLog::logIACommit()

Calls - File IO routines

Parameters - IAID, GTID

Returns - Status OK/NOT_OK

Effects - deletes either of the following subdirectories
pro/mdb/ild/global_log/IAID or
/pro/mdb/ild/global_log/IAID/GTID

1.2.4 Status ild :: delActiveEntry(int IAID, int GTID)

Semantics - This routine removes the specified Global Transaction from the dependency information of the file of the Interaction.

Called By - RRD::rollBckGtRrd()
gtLog::logGTCommit()

Calls - File I/O routines

Parameters - IAID, GTID

Returns - Status OK/NOT_OK

Effects - Removes an entry corresponding to the GTID from the IAID.active file.

1.3 Protected Member Functions

The protected member functions are used only by the ild and it's derived classes (iaLog and the gtLog).

1.3.1 Status ild :: writeLog(int, int, REC, char*)

Semantics - This routine writes a log record to the MASTER_LOG. The log record is described below.

Called By - Logging routines in the iaLog, gtLog and the ild.

Calls - File IO.

Parameters - IAID, GTID,
Rec_Type - Identifies the type of record.
Log_entry - String of characters to store information

Returns - Status OK/NOT_OK

Effects - Adds an entry to the MASTER_LOG thereby updating it.

1.3.2 int ild :: checkFile(char*)

Semantics - This routine checks whether a file with the specified name exists. It does so by trying to open the iostream with certain flags.

Called By - Routines in the iaLog, gtLog, and the ild.

Calls - File I/O

Parameters - char *name

Returns - TRUE if file exists
FALSE if it does not exist.

Effects - None

1.3.3 Status ild :: createDir(int IAID, int GTID)

Semantics - This routine creates a directory for the specified Interaction or the Global Transaction. If the GTID is NULL we create an IAID directory. If the GT is a integer we create the directory we always check to insure that there is an IAID directory before creating the GTID directory.

Called By - gtLog::logGTBegin()
iaLog::logIABegin()

Calls - File and directory routines

Parameters - IAID, GTID

Returns - Status OK/NOT_OK

Effects - A directory (IAID or GTID) is created at the path specified as
path for IAID = /pro/mdb/ild/global_log/IAID
path for GTID = /pro/mdb/ild/global_log/IAID/GTID

1.4 Private Member Functions

The private member functions are used only by the ild and are generally helper or initialization functions.

1.4.1 RRDDAG* ild :: getRRDDAG(int IAID, int GTID)

Semantics - This routine returns the RRDDAG for the specified Interaction or Global Transaction. If the GTID is NULL then the entire Interaction is read into the DAG (Directed Acyclic Graph). otherwise the specified Global Transaction and it's descendents are read into the DAG.

Called By - ild::abortIa()
ild::abortGt()
ild::retryIa()
ild::retryGt()

Calls - ild::readDepInfo()
RRDDAG::insertNode()
Set::addInt()
Set::getCurrentInt()

Parameters - IAID, GTID

Returns - RRDDAG * - DAG corresponding to the GTID, IAID

Effects - Allocates memory for RRDDAG*. This memory is freed in * RRD::rollBckGtRrd(). In case a GTID needs to be rolled back, the GTID node is added to the DAG.

1.4.2 Status ild :: readDepInfo(int IAID, RRDDAG* dag_ptr)

Semantics - This routine reads the dependency information for the given Interaction. It reads the "TO" field on each line and inserts it as a NODE in the DAG.

Called By - ild::getRRDDAG()

Calls - RRDDAG::insertNode()

Parameters - IAID - desired IAID
RRDDAG* - DAG to read the dep info

Returns - Status OK/NOT_OK

Effects - Modifies the DAG structure by adding nodes

1.4.3 Status ild :: readDepInfo(int IAID, int GTID, RRDDAG* dag_ptr, Set* set_ptr)

Semantics - This routine reads the dependency information for the given Global transaction. It first reads the "TO" field in each line. Then it scans through the "FROM" fields for the "TO" field. If the "FROM" field matches the GTID then an edge from "FROM" to "TO" is added to the DAG and "TO" is added to the set of NODES

Called By - ild::getRRDDAG()

Calls - RRDDAG::insertEdge()

Parameters - IAID, GTID - Desired GT"s
RRDDAG * - DAG to read dependency info
Set * - Set of GTID's to examine

Returns - Status OK/NOT_OK

Effects - Modifies the DAG structure by adding Edges. Also, adds integers to the Set.

2. Class iaLog (Interaction Log)

The iaLog is responsible for logging actions pertaining to the Interaction. It is derived from the class ild and an interface to the IM only. The interface consists of the following functions

2.1 int iaLog :: logIABegin(int IAID)

Semantics - This routine first creates the directory for the IAID.

Called By - Routines in the IM

Calls - ild::createDir()
ild::writeLog()

Parameters - IAID

Returns - Status OK/NOT_OK

Effects - Creates a new subdirectory GLOBAL_LOG_DIR/IAID. It then writes a log record for the BEGIN operation.

2.2 int iaLog::logIACommit (int IAID)

Semantics - This routine logs a IA commit record. It first checks to see if the IA is still active. If the IA is active, it then checks for active GT's. If all GT's have been committed, the IAID directory is deleted and the record is logged.

Called By - Routines in the IM

Calls - ild::deleteDir()
ild::writeLog()
ild::checkFile()

Parameters - IAID

Returns - Status OK/NOT_OK/BUG

Effects - Deletes the GLOBAL_LOG_DIR/IAID subdir, and adds a record to the MASTER_LOG.

2.3 int iaLog :: logIAAbort (int IAID)

Semantics - This routine writes an Abort record to the MASTER_LOG.

Called By - Routines in the IM

Calls - ild::writeLog()

Parameters - IAID

Returns - Status OK/NOT_OK

Effects - Adds a record to the MASTER_LOG

3. Class gtLog (Global Transaction log)

The gtLog is responsible for logging actions pertaining to the Global Transactions. It is derived from the class ild and an interface to the IM only. The interface consists of the following functions

3.1 Status gtLog :: logGTBegin (int IAID, int GTID, char *depinfo)

Semantics - This routine logs a GT Begin Record. It first creates the GT directory and then updates the IAID.active file. The IAID.depinfo file is updated to reflect the dependencies.

Called By - Routines in the IM

Calls - ild::writeLog()

Parameters - IAID, GTID
Depinfo - The depinfo is added to the file GLOBAL_LOG_DIR/IAID/IAID.depinfo

Returns - Status OK/NOT_OK

Effects - Modifies the following files
GLOBAL_LOG_DIR/IAID/IAID.active and
GLOBAL_LOG_DIR/IAID/IAID.depinfo.

3.2 Status gtLog :: logGTCommit (int IAID, int GTID, char *gst_info)

Semantics - This routine logs a GT commit record. It first removes the entry from the IAID.active file. It then deletes the directory of the GT.

Called By - Routines in the IM

Calls - ild::writeLog()
ild::deleteDir()
ild::delActiveDir()

Parameters - IAID, GTID
gst_info - Information regarding global sub transactions

Returns - Status OK/NOT_OK

Effects - Modifies GLOBAL_LOG_DIR/IAID/IAID.active and removes the GLOBAL_LOG_DIR/IAID/GTID subdirectory.

3.3 Status gtLog :: logGTAbort (int IAID, int GTID)

Semantics - This routine writes an ABORT record to the MASTER_LOG.

Called By - Routines in the IM

Calls - ild::writeLog()

Parameters - IAID, GTID

Returns - Status OK/NOT_OK

Effects - Adds a record to the MASTER_LOG

4. Class RRD (Rollback and Recovery Daemon)

This class is responsible for the recovery of Interactions and Global Transactions. It is spawned as a separate process by the ILD when it is required to abort/retry an Interaction/Global Transaction. It communicates with 2 external entities, the ILD and the IM. With the ILD it is an input communication through the class interface. With the IM it is via RPC calls as the IM is a separate process. It has a proposed interface with the TaSL.

4.1 Interface with the ILD

The ILD invokes the rollback and recovery procedure for an Interaction by calling the RRD with the desired information.

4.1.1 Status RRD :: rollBckGtRrd(RRDDAG *dag_ptr)

Semantics - This routine is the critical one in the RRD and is the only interface to the RRD. The roll back routine takes as input a DAG. The DAG represents the transactions that need to be aborted/compensated. The DAG is first topologically sorted, then each transaction is checked its current status.

The ACTIVE GT"s are aborted immediately by removing the sub directories for the given GT, removing the entries from the IAID.active file and lastly logging the ABORT GT record.

The COMMITED transactions are more complicated to rollback. Essentially, they need to be compensated for.

Called By - ild::abortIa()
ild::abortGt()
ild::retryIa()
ild::retryGt()

Calls - RRD::toposort()
RRD::compTrans()
ild::verifyGTstatus()
ild::deleteDir()
ild::delActiveEntry()
gtLog::logGTAbort()

Parameters - RRDDAG* - DAG corresponding to the IA or GT.

Returns - Status OK/NOT_OK

Effects - Deletes the RRDDAG* which was allocated in ild::getRRDDAG

4.2 Connection to the IM (Interaction Manager)

The RRD communicates to the IM via RPC to compensate for a Global Transaction.

4.2.1 int RRD :: IM_beginGT(int IAID, int* pred_list)

Semantics - This routine begins a new GT by making a RPC call to the IM. It first creates a client handle for the IM if it does not exist. This handle is later destroyed by the destructor.

Called By - RRD::compTrans()

Calls - RPC calls to the IM to begin a new Global Transaction.

Parameters - int IAID,
int *predlist - Predecessor list of the GT.

Returns - GTID of the new transaction.

Effects - Initializes the data member IM_cl when called first time.

4.2.2 Status RRD :: IM_doCsStep(int GTID, char *hostname, int LSN)

Semantics - This routine performs the compensating steps for a GT, by making RPC call to the IM.

Called By - RRD::compTrans()

Calls - RPC calls to the IM.

Parameters - int GTID
char *hostname - Local Database Name
int LSN - Log Sequence Number on the Local Database

Returns - Status OK/NOT_OK

Effects - None

4.2.3 Status RRD :: IM_commitGT(int IAID, int GTID)

This routine commits a GT by making a RPC call to the IM.

Called By - RRD::compTrans()

Calls - RPC calls to the IM to commit GT.

Parameters - int IAID
int GTID

Returns - Status OK/NOT_OK

Effects - None

4.3 Private Member Functions

The private member functions are used only by the ild and are generally helper or initialization functions.

4.3.1 Status RRD :: compTrans(int IAID, int GTID)

Semantics - This routine compensates the specified Global Transaction. It first begins a new GT by communicating with the IM. Then it reads the MASTER_LOG for the GST information. Lastly, the steps for each of the GST are executed to complete the compensation and the new GT is then COMMITED.

Called By - RRD::rollBckGtRrd()

Calls - RRD::IM_beginGT()
RRD::IM_doCsStep()
RRD::IM_commitGT()
ild::readLog()

Parameters - IAID
GTID

Returns - Status OK/NOT_OK

Effects - None

4.3.2 CST_info* RRD :: getCSTInfo(char *str)

Semantics - This routine converts a string containing the GST information into the CST_info structure. The format of the string is "Num_of_GST LDB_Name LSN....."

Called By - RRD::compTrans()

Calls - RRD::parseString()

Parameters - char *str - GST information

Returns - CST_info structure (The last element is packed with an INVALID_LSN)

Effects - None

4.3.3 void RRD :: parse_string(char *str, int& count, char *arg_str)

Semantics - This is an internal routine which parses a string delimited by a blank character.

Called By - RRD::getCSTInfo()

Calls - None

Parameters - char *str - String to be parsed
int& count - Character to start counting at.
char *arg_str - Argument string to store the field

Returns - None

Effects - Updates "count" and arg_str.

5. Class RRDDAG

This class stores the DAG (Directed Acyclic Graph) and provides operations to modify it. The DAG is internally stored as an adjacency list and is consists of other classes such as the vertices and vertex list which are not described in depth. It interfaces with the ILD and the RRD Since there is an overlap

5.1 Interface with the ILD

The ILD mainly builds the DAG and hence the interface consists of messages to insert edges and vertices.

5.1.1 void RRDDAG :: insertEdge (int from_vertex, int to_vertex)

Semantics - This routine inserts an edge in the DAG. It first inserts the "TO" and "FROM" nodes in the vertex list. It then inserts the "TO" node in the vertex list of the "FROM" node. * (The DAG is represented internally by an adjacency list.

Called By - ild::readDepInfo()

Calls - RRDvertexList::insertVertex()
RRDvertexList::getVertex();

Parameters - from_vertex - "FROM" node
to_vertex - "TO" node

Returns - None

Effects - Adds elements to the vertex list of the DAG as well as the "TO" vertex.

5.1.2 void RRDDAG :: insertNode(int node)

Semantics - This routine inserts a NODE in the DAG. It does so by inserting a node in it's vertex list.

Called By - ild::getRRDDAG()
ild::readDepInfo()

Calls - RRDvertexList::insertVertex()

Parameters - Node - NODE to be inserted

Returns - None

Effects - Adds a NODE to the vertex list.

5.2 Interface to the RRD

The RRD uses the topologically sorted DAG for compensating transaction.

5.2.1 void RRDDAG :: toposort(int *GTID_list)

Semantics - This routine sorts the DAG topologically and returns the elements as an integer list. It uses the DFS to do the topologically sort.

Called By - RRD::rollBckGtRrd()

Calls - RRDDAG::initializeVertices()
RRDDAG::DFSvisit()

Parameters - int *GTID_list - List of integers to store the final set of elements.

Returns - None

Effects - Modifies the input GTID_list.

5.3 Private Member Functions

The private member functions are used only by the RRDDAG and are generally helper or initialization functions.

5.3.1 void RRDDAG :: DFSvisit(RRDvertex *vertex_ptr, int *GTID_list)

Semantics - This routine calls the DFS on the DAG and stores the elements in integer list in the order of it's "FINISH TIME". It is a recursive routine and recurses on the members of the GTID_list.

Called By - RRDDAG::toposort()

Calls - RRDDAG::DFSvisit()
RRDvertex::startTime()
RRDvertex::finishTime()
RRDvertex::color()

Parameters - RRDvertex *ptr - Node on which to recurse.

Returns - None

Effects - This routine updates the data members "time" and "topo_count". It also stores the NODES in the GTID_list.

5.3.2 void RRDDAG :: initializeVertices()

Semantics - This routine initializes all the NODES in the DAG for the DFSvisit.

Called By - RRDDAG::toposort()

Calls - RRDvertex::color()
RRDvertex::startTime()
RRDvertex::finishTime()
RRDvertex::parent()

Parameters - None

Returns - None

Effects - Initializes some of the data members of the vertices. Also sets topo_count.

Local Logging

Module

The Local Level contains a single module, the LRS server

LRS server

The LRS server is created as new process by the Agent Manager (AM). It accepts requests from the GST (Global Sub Transaction) to perform the local level logging.

The server first creates a communication port using it's newly acquired program number from the Pnum server. The details of the port (program number and name) are then passed to the Agent for future communication. The Agent when creating the GST, gives the LRS server location in order to communicate with the server.

Finally, the server is ready to accept requests from the GST. The requests are send to the dispatcher. The dispatcher unmarshals the request, makes the desired function call, and returns the results to the calling process.

Note - The server blocks the calling process and communication is via RPC using customized routines. (A set of standardized routines are available for communicating with entities within the Mongrel MDBS)

Data Structures

1. return_info

It stores the return information of a function call.

```
typedef struct                          // Return value structure
{
 int stat;
 int argc;
 char **argv;
} return_info;
```

2. step_info

It stores information about steps, their compensating steps and the return values.

```
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;
```

3. cstep_info

It stores details of a compensating step.

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

4. log_rec

It is used to store all log entries of a specific LSN.

```
typedef struct                          // Log record structure
{
 int LSN;
 int num_rec;
 int time_stamp[MAX_REC];
 rec_type rec[MAX_REC];
 char *log_entry[MAX_REC];
} log_rec;
```

5. comp_info

It is used to return the compensating information for a specified LSN.

```
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;
```

Classes

The local level logging consists of the following relevant classes which are described in the Appendix

6. LRS
7. LocalLog

For each class, important instance variables and member functions are examined in detail. The classes are examined on their functionality basis i.e. the member functions are grouped into logical components.

6 Class LRS (Local Recovery System)

This class is responsible for the logging and recovery at the local level of the two level log. It contains a reference to a LocalLog and routines read/write to/from the log. It's only external interface is the GST (Global Sub Transaction).

6.1 Interface to the GST

The GST invokes the LRS for logging it's complete execution. During compensation it requires the compensation information that it gets from the LRS.

6.1.1 LRS :: LRS()

Semantics - This is the constructor of the class LRS. It first creates instantiates the LocalLog, and then checks for the following files

1. /pro/mdb/ild/local_log/LOCAL_LOG.
2. /pro/mdb/ild/local_log/LSN_FILE.

If they do not exist, it creates them. It also, initializes the LSN to the initial count of 1.

Called By - The LRS dispatcher

Calls - LocalLog::LocalLog()
LocalLog::writeInitRecord()
File I/O routines

Parameters - None

Returns - None

Effects - Creates a new LocalLog and writes the initialization record if the LOCAL LOG does not exist.

6.1.2 Status LRS :: logGSTAbort(int LSN)

Semantics - This routine logs the abort GST record to the LOCAL_LOG. After logging it successfully, it deletes the temporary LSN file.

Called By - Routines in the GST

Calls - unlink()
LocalLog::logGSTAbort()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - Adds an Abort record and then deletes the local LSN file if one is present.

6.1.3 int LRS :: logGSTBegin()

Semantics - This routine logs a GST Begin Record. First it assigns a LSN to the GST.
The LSN - Log Sequence Number is a unique identifier for the GST and is used for all the logging purposes. Next the BEGIN record is entered into the LOCAL_LOG.

Called By - Routines in the GST

Calls - LRS::getLSN()
LocalLog::logGSTBegin()

Parameters - None

Returns - LSN - Log Sequence Number
or INVALID_LSN if it fails.

Effects - Creates a file /pro/mdb/ild/local_log/LSN_files/LSN if successful.

6.1.4 Status LRS :: logGSTCommit(int LSN)

Semantics - This routine logs the GST commit record. It does so by delegating the call to the Local Log.

Called By - Routines in the GST.

Calls - LocalLog::logGSTCommit()

Parameters - LSN.

Returns - Status OK/NOT_OK

Effects - None

6.1.5 Status LRS :: logGSTPrepared(int LSN)

Semantics - This routine prepares the Local Database for the Two Phase Commit Algorithm. It corresponds to the Phase I of the 2 phase commit.

It first commits the transaction locally by flushing the data in the checkpoint file system to the LOCAL_LOG (on the stable storage).

It then adds the Log Entry for the PREPARED state. Lastly, it deletes the LSN file in the checkpoint file system.

Called By - Routines in the GST

Calls - unlink()
LocalLog::logGSTPrepared()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - Copies the contents of the LSN file to LOG. Adds entry to the LOG and deletes the LSN file.

6.1.6 Status LRS :: logGSTAbortDecision(int LSN)

Semantics - This routine logs the ABORT DECISION in the LOCAL_LOG. After the 1st phase of the 2PC GST is either Committed or Aborted. This message is used to signify the ABORT of the GST.

Called By - Routines in the GST

Calls - LocalLog::logGSTAbortDecision()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

6.1.7 Status LRS :: logStepInfo(LSN, step_info)

Semantics - This function logs the step information. It does so by converting the Step information to a log entry (a string) and then asks the Local Log to do the logging.

Called By - Routines in the GST

Calls - LRS::stepInfoToLogEntry()
LocalLog::logStepInfo()

Parameters - int LSN
struct step_info s_info

Returns - Status OK/NOT_OK

Effects - Deletes the char *log_entry which is allocated in LRS::stepInfoToLogEntry()

6.1.8 Status LRS :: logCStepInfo(LSN, cstep_info)

Semantics- This function logs the CStep information. It does so by converting the Step information to a log entry (a string) and then asks the Local Log to do the logging.

Called By - Routines in the GST

Calls - LRS::cstepInfoToLogEntry()
LocalLog::logCstepInfo()

Parameters - int LSN
struct cstep_info cs_info

Returns - Status OK/NOT_OK

Effects - Deletes the char *log_entry which is allocated in LRS::cstepInfoToLogEntry()

6.1.9 comp_info* LRS :: readCompInfo(int LSN)

Semantics - This routine get the compensating step information for the given LSN. It first reads the local log and gets all the records for the LSN.
It then filters out the compensating information and returns to the calling function.

Note:- This routine allocates storage for the comp_info. The calling function is responsible for deleting the storage

Called By - Routines in the GST

Calls - LocalLog::readLog()
LRS::logEntrytoCompInfo()

Parameters - LSN

Returns - struct comp_info* if successful, NULL otherwise

Effects - Deletes the struct log_rec* allocated in the routine LocalLog::readLog().

6.2 Private Member Functions

The private member functions are used only by the LRS and are generally helper or initialization functions.

6.2.1 int LRS :: getLSN()

Semantics - This routine generates a new LSN and returns it to the calling function. It reads in the current LSN from the /pro/mdb/ild/local_log/LSN_FILE. and is later updated by incrementing the currently stored LSN. If for any reason it is unsuccessful then an INVALID_LSN is returned to the user.

Called By - LRS::logGSTBegin()

Calls - File IO routines

Parameters - None

Returns - New LSN if successful, INVALID_LSN otherwise.

Effects - Updates the /pro/mdb/ild/local_log/LSN_FILE.

6.2.2 char* LRS :: stepInfoToLogEntry(step_info)

Semantics - This function converts contents of a Step Information structure to a log entry (string of characters). It allocates storage for the string, it then stores the individual fields of step_info in the string using the DELIMITER and returns it to the calling function which is responsible for clearing the string storage.

Called By - LRS::logStepInfo()

Calls - String Manipulation functions

Parameters - struct step_info

Returns - char *log_entry if successful, NULL otherwise.

Effects - None

6.2.3 char* LRS :: cstepInfoToLogEntry(cstep_info)

Semantics - This function converts contents of a CStep Information structure to a log entry (string of characters). It allocates storage for the string, it then stores the individual fields of cstep_info in the string using the DELIMITER and returns it to the calling function which is responsible for
clearing the string storage.

Called By - LRS::logCStepInfo()

Calls - String Manipulation functions

Parameters - struct step_info

Returns - char *log_entry if successful, NULL otherwise.

Effects - None

6.2.4 Boolean LRS :: logEntrytoCompInfo(char *log_entry, comp_info* cinfo)

Semantics - This routine reads a log entry and extracts the compensation step information from the log entry. This routine is tightly coupled with the format in which the STEP and the corresponding CSTEP information is stored in the LOG.

Called By - LRS::readCompInfo()

Calls - LRS::getField()

Parameters - char *log_entry
struct comp_info *cinfo

Returns - TRUE if successful, FALSE otherwise

Effects - This function modifies the following fields of the comp info structure

1. Num rec
2. cstep_id
3. argc - storage is allocated for this field and is hence
   for the calling function to free the storage.
4. argv

It also deletes the temporary strings that are allocated by LRS::getField();

6.2.5 char* LRS :: getField(char *log_entry, int field)

Semantics - This routine returns the contents of specified field. It allocates storage for the string and the calling function is responsible for freeing it.

Called By - LRS::logEntrytoCompInfo()

Calls - None

Parameters - char *log_entry
int field

Returns - char* - contents of the specified field

Effects - Allocates storage for the returning string

6.2.6 Boolean LRS :: checkFile(char *filename)

Semantics - This routine checks for the existence of the specified file.

Called By - LRS::LRS()
LRS::logGSTAbort()

Calls - File IO routines

Parameters - char *filename

Returns - TRUE if file exists
FALSE if it does not exist

Effects - None

7. Class LocalLog

This class does the actual physical read/write from/to the log. It's only interface is the LRS (of which it is a component)

7.1 Interface to the LRS

The LRS communicates with the LocalLog for the physical read/write of the Logs.

7.1.1 Status LocalLog :: logGSTAbort(int LSN)

Semantics - This routine logs a GST Abort Message for the specified LSN.

Called By - LRS::logGSTAbort(int LSN)

Calls - LocalLog::writeLog()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

7.1.2 Status LocalLog :: logGSTPrepared(int LSN)

Semantics - This routine logs a GST Prepared Message for the specified LSN.

Called By - LRS::logGSTPrepared(int LSN)

Calls - LocalLog::writeLog()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

7.1.3 Status LocalLog :: logGSTAbortDecision(int LSN)

Semantics - This routine logs a GST Abort Decision Message for the specified LSN.

Called By - LRS::logGSTAbortDecision(int LSN)

Calls - LocalLog::writeLog()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

7.1.4 Status LocalLog :: logGSTCommit(int LSN)

Semantics - This routine logs a GST Commit Message for the specified LSN.

Called By - LRS::logGSTCommit()

Calls - LocalLog::writeLog()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

7.1.5 Status LocalLog :: logGSTBegin(int LSN)

Semantics - This routine logs a GST Begin Message for the specified LSN.

Called By - LRS::logGSTBegin(int LSN)

Calls - LocalLog::writeLog()

Parameters - LSN

Returns - Status OK/NOT_OK

Effects - None

7.1.6 Status LocalLog :: logStepInfo(LSN, log_entry)

Semantics - This routine logs the STEP info for the specified LSN.

Called By - LRS::logStepInfo()

Calls - LocalLog::writeLog()

Parameters - LSN
char *log_entry

Returns - Status OK/NOT_OK

Effects - None

7.1.7 Status LocalLog :: logCstepInfo(LSN, log_entry)

Semantics - This routine logs the CSTEP info for the specified LSN.

Called By - LRS::logCstepInfo()

Calls - LocalLog::writeLog()

Parameters - LSN
char *log_entry

Returns - Status OK/NOT_OK

Effects - None

7.1.8 log_rec* LocalLog :: readLog(int LSN)

Semantics - This routine returns all the log records for a given LSN. Storage is first allocated for the log record and the log entries it contains. The data fields are then initialized. The log entries are scanned sequentially till all the records of the LSN have been read or the EOF is reached. The log record structure is updated if the scanned LSN is the specified one.

Called By - LRS::readCompInfo()

Calls - File IO and string manipulation routines.

Parameters - LSN

Returns - Log record structure if successful (see the file LRS_global.H for details
NULL otherwise.

Effects - Returns the pointer to a structure, so please delete the structure as soon as you are done with. It is a memory intensive structure and hence can cause memory problems if left unattended.

7.2 Private member Functions

The private member functions are used only by the LocalLog and are generally helper or initialization functions.

7.2.1 Status LocalLog :: writeLog(LSN, rec_type, log_entry)

Semantics - This function writes a LOG entry to the LOCAL LOG. The entry consists 4 fields

1. Timestamp
2. LSN
3. Record Type
4. Log Entry - String

Called By - Other LocalLog functions

Calls - File IO routines

Parameters - int LSN
enum Record Type
char *log_entry

Returns - Status OK/NOT_OK

Effects - Adds a Log Entry to the file /pro/mdb/ild/local_log/LOCAL_LOG.

7.2.2 Status LocalLog :: writeLSN(LSN, rec_type, log_entry)

Semantics - This function writes a log entry to the temporary LSN file. The entry consists 4 fields

1. Timestamp
2. LSN
3. Record Type
4. Log Entry - String

Called By - Many LocalLog functions

Calls - File IO routines

Parameters - int LSN
enum Record Type
char *log_entry

Returns - Status OK/NOT_OK

Effects - Adds a Log Entry to the file /pro/mdb/ild/local_log/LSN_files/LSN.