M.Tech. Dissertation

Advanced Transactions in LDAP

submitted in partial fulfillment of the requirements for the degree of
Master of Technology

By

Anandi D. Herlekar
Atul Deopujari

Prof. K. Ramamritham

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
January 29, 2002

Abstract

Lightweight Directory Access Protocol(LDAP) is an open industry standard for accessing common directory information. It is being used extensively in many network applications. While single LDAP operation is atomic, such assurance cannot be made for multiple LDAP operations. A transaction permit an application to ensure consistency of the information it stores in the directory. For example, should an application need to maintain consistent information across more than one directory entry, without transactions the potential exists for a failure to occur after one entry is modified but before the second one is. Moving an entry from one directory to another would require the entry to be deleted from one LDAP directory and adding to another. The consistency requirement is that the entry should be present in only one directory at a time. This is possible only with transactions. However, typical directory applications are long duration activities, where the traditional transaction semantics of locking and logging may not prove to be efficient, use of advanced transactions support become imperative.

We have developed a framework to provide advanced transaction support in LDAP and means to enforce application specific integrity constraints. We have added active functionality to LDAP by introducing a layer on top of LDAP. We call this layer, Predefined Trigger Action Protocol (PTAP). This layer provides to the naive user basic APIs for transaction support. The transaction support provided by PTAP includes the standard transaction model and some of the advanced transaction models (Nested Transaction, Nested Saga, Co-Operating Transaction) for applications that are long running. An advanced user can use the primitives provided by PTAP and implement the advanced transaction model that will best suit his/her application.

Contents

List of Figures

Chapter 1

Introduction

LDAP directories were designed keeping in mind applications where reads are more frequent than writes. So strict consistency was not a requirement for LDAP databases. For example, in a user information directory, it might be acceptable if information such as telephone number is inconsistent. For maintaining consistency of data, standard relational databases with strict transaction functionality can be used. However, LDAP offers many useful properties such as a natural way of representing data in hierarchical form, efficient read access, scalability, support for heterogenous data etc. Therefore LDAP is being used for purposes other than the traditional white pages application. In particular, information critical to the functioning of networks and their services is being stored in the directory. Examples are DNS data, certificates, and application configuration data. When used in this "Network Repository" role, it becomes increasingly important to ensure integrity of the information stored within the directory. The lack of support for transactions in the LDAP protocol is seen as a barrier to the more widespread use of the directory for such purposes.

Transactions [GR92] permit an application to ensure consistency in the information it stores in the directory. For example, should an application need to maintain consistent information across more than one directory entry, without transactions the potential exists for a failure to occur after one entry has been modified but before the second one has. Moving an entry from one directory to another would require the entry to be deleted from one LDAP directory and adding to another. The consistency requirement is that the entry should be present in only one directory at a time. This is possible only with transactions. However, typical directory applications are long duration activities, where the traditional transaction semantics of locking the entries for the duration of the transaction [SKS97] may not prove to be efficient. Use of advanced transactions [RC96] become imperative.

So we provide support for building different advanced transaction models to be used with LDAP. These models relax some of the consistency requirements of the traditional transaction semantics, keeping in view the semantics of the application.

We have developed an executable transaction modeling language for defining existing and new extended transaction models in LDAP based on the primitives discussed in [BDG+94]. We have added active functionality to LDAP by introducing a layer on top of LDAP. We call this layer, Predefined Trigger Action Protocol (PTAP). This layer provides to the naive user basic APIs for transaction support. The transaction support provided by PTAP allows standard transaction model and some advanced transaction modeling support e.g. Nested Transaction, Nested Saga, Co-Operating Transaction for the applications that are long running. An advanced user can use the APIs provided by PTAP and implement the advanced transaction model that will best suit his/her application.

The report is organized as follows:
Chapter 2 discusses a motivating example where LDAP can be used to organize the data. The example focus on the integrity constraints for making a travel plan and how these can be implemented using advanced transaction semantics. Chapter 3 gives the general overview of the design of the PTAP module and the user interface to PTAP. Chapter 4 explains the important data structures and algorithms in PTAP. Chapter 5 explains how a test client is to be set. chapter 6 gives the summary and the possible future work. The appendix A gives the configuration and usage options for PTAP.

Chapter 2

Motivating Example

This chapter discusses a motivating example. It involves different scenarios of making a travel plan and how one can model these scenarios using specific advanced transaction models.

2.1 Travel Plan

We discuss an example where a person has to make a travel plan to attend a meeting at some place. These are the possible steps that need to be done to do the travel plan:

1 Making a forward flight reservation to the destination.

2 Making a hotel reservation at the destination city.

3 Making a return flight reservation from the destination.

These activities are performed by transactions `forward_res`, `return_res` and `hotel_res` respectively. The global transaction of the travel plan is referred to as `travel_plan`. We note the following integrity constraints for the application:

1 `forward_res` and `return_res` work co-operatively on the data

2 success of `hotel_res` depends on the success of `forward_res`

Taking the integrity constraints into account, the success of `travel_plan` can be under the following scenarios:

1 `forward_res`, `hotel_res` and `return_res` must commit successfully

2 `forward_res` must commit successfully but `hotel_res` may fail (reaching the destination city has top priority, accommodation can be done somehow), no constraint on `return_res`

3 `return_res` may fail while `forward_res` and `hotel_res` must commit successfully (will get back somehow)

Before we describe how each of the scenarios can be implemented using advanced transaction models, a brief digression on the semantics of some of the advanced transaction models and the semantics of the primitives defined in [BDG$^+$94] seems to be in order.

2.2 Advanced Transaction models and Transaction primitives

We describe the semantics of some of the advanced transaction models and describe how they can be realized using the primitives.

2.2.1 The Nested Transaction model

The Nested Transaction model permits transactions to be nested within transactions to form a tree of subtransactions. The root of the tree is called the top-level. The tree as a whole is failure atomic, serializable and therefore, isolated from the other trees/transaction. Subtransactions execute atomically with respect to their siblings and to other, nonrelated transactions, and are atomic with respect to their parent. They can abort independently without causing the whole transaction to abort. A subtransaction can potentially access any object that is currently accessed by one of its ancestor transactions. When a subtransaction commits, the objects modified by it are made accessible to its parent transaction. However, the effects on the objects are made permanent in the database only when the root transaction commits. Failure of the top-level transaction results in the abort of each of the subtransactions.

2.2.2 The Nested Saga Transaction model

The Nested Saga model permits a transaction to be broken down into component subtransactions. Each component subtransaction has a corresponding compensating transaction that semantically undoes the effects of its committed associate. The component subtransactions of a SAGA execute in a serial order and it is also permitted that subtransactions of other sagas can be interleave in their execution. When a component subtransaction of a saga aborts, its results are undone and a chain reaction results that executes the compensating transactions of each of the predecessor subtransactions of the same saga.

2.2.3 The Split Transaction model

In Split transaction model, a (long)transaction can be split into two or more smaller transactions in such a manner that the new transactions are serializable. Partial results can be committed or aborted by committing the transaction that has been split off – even before the splitting transaction is committed. This makes some changes visible to other transactions, even though the transaction that made the changes is still in progress. Splitting also allows other short-duration transaction, which are waiting for objects released as a result of partial commitment, to proceed.

2.2.4 The Co-operating Transaction model

In this model, a (long)transaction can be split into two smaller transactions that work co-operatively on the data. However, in this model the commitment of the split transaction depends on the commitment of the splitting transaction.

The paper [BDG$^+$94] defines the following transaction primitives to realize the advanced transaction models

- **permit**(t_i, t_j, ob_set) **:** Transaction t_i permits transaction t_j to perform conflicting operations on objects in ob_set that are locked by it
- **delegate**(t_i, t_j, ob_set) **:** Transaction t_i delegates the responsibility of the operations performed by it on objects in ob_set to transaction t_j. The operations can commit only if t_j commits.

- **form_dependency**($type,\ t_i,\ t_j$) **:** Transaction t_i forms a dependency of $type$ with transaction t_j governing its commit/abort semantics with respect to t_j. The dependencies can be one of

  commit : If both commit then t_i cannot commit before t_j, however if t_j aborts then t_i can still commit
  abort : If t_j aborts then t_i aborts
  group : Either both commit or neither commit

2.2.5 Realization of Advanced Transaction models

The nested transaction model can be realized using the above primitives as follows:

> The parent transaction *permit*s the child transaction to perform conflicting operations.
> The child transaction *delegate*s to the parent transaction after it has finished execution.
> *form_dependency* `group commit` forces the commit of both or none transactions,
> if such a behaviour is desired

The co-operating transaction can be realized as follows:

> The splitting transaction *permit*s the transaction that splits to perform conflicting operations
> The transaction that split forms a *commit* dependency with the splitting transaction forcing it
> to commit only after the splitting transaction commits.

In a similar manner other advanced transactions and custom transaction behaviour can be realized using the primitives, [BDG+94] contains the details.

Following sections describe how each of the scenarios can be implemented using advanced transaction models and the primitives.

2.3 Scenario 1

In this scenario, the success of `travel_plan` depends on the success of all the three transactions. Therefore they are modeled in the nested transaction model with `group commit` dependency among them.

2.4 Scenario 2

In this scenario, the success of `travel_plan` depends on the success of `forward_res`. Therefore these are modeled using the Nested Transaction model with parent as `travel_plan`. Failure of `forward_res` results in the failure of `travel_plan`. There is no constraint on `hotel_res` and `return_res`. However, these transactions should abort if `travel_plan` aborts. Therefore, `hotel_res` and `return_res` form an `abort` dependency with `travel_plan`.

2.5 Scenario 3

In this scenario, `travel_plan` depends on the joint commit of `forward_res` and `hotel_res`. Therefore `travel_plan` and `forward_res` are modeled in the Nested Transaction model. The only constraint on `return_res` is that it should abort if `travel_plan` aborts which can be imposed by forming an `abort` dependency between `return_res` and `travel_plan`.

Chapter 3

Advanced transactions across multiple LDAP servers

The increasing use of LDAP in Directory Enabled Networking (DEN) applications has made it imperative to support advanced transaction in LDAP. Advanced transactions can be implemented by adding active functionality to LDAP. Lightweight Trigger Action Protocol (LTAP)[AFG+99] is such a protocol that is designed to provide active functionality to LDAP. LTAP works as a gateway that pretends to be an LDAP server - LDAP commands intended for the LDAP server are intercepted by LTAP which does trigger processing in addition to servicing the original LDAP command. The LTAP gateway provides a portable way to add triggers to an LDAP directory. LTAP also provides locking facilities, forbidding updates to an entry while trigger processing is being performed on that entry.

The design of PTAP is based on the above guidelines of LTAP. This chapter gives a design overview of PTAP and the APIs that are exposed to the user. We have kept the implementation generic enough to be plugged in into any LDAP server and to plug in any integrity enforcer depending upon the application. Trigger actions are defined by pre-defined callback APIs. These APIs can be implemented according to application specific needs.

The APIs exposed by PTAP are of two types.

- Client side APIs: These include APIs for initiating and terminating transactions and the primitive APIs that form dependency among them.
- Callback APIs: The client side of PTAP defines callback functions that are called for the predefined trigger events like pre-add, post-add, pre-delete and post-delete etc. These functions need to be implemented to provide application specific Integrity Constraint Manager(ICM).

PTAP assumes that the underlying LDAP server provides single LDAP operation as an atomic operation. (In case of OpenLDAP, which does not satisfy this condition, we have modified the back-sql backend to support this. We have used Oracle as the backend for this purpose.)

3.1 Design overview

Figure 3.1 shows the different components of PTAP and the interaction between them.

PTAP has the client side component and the server side component.

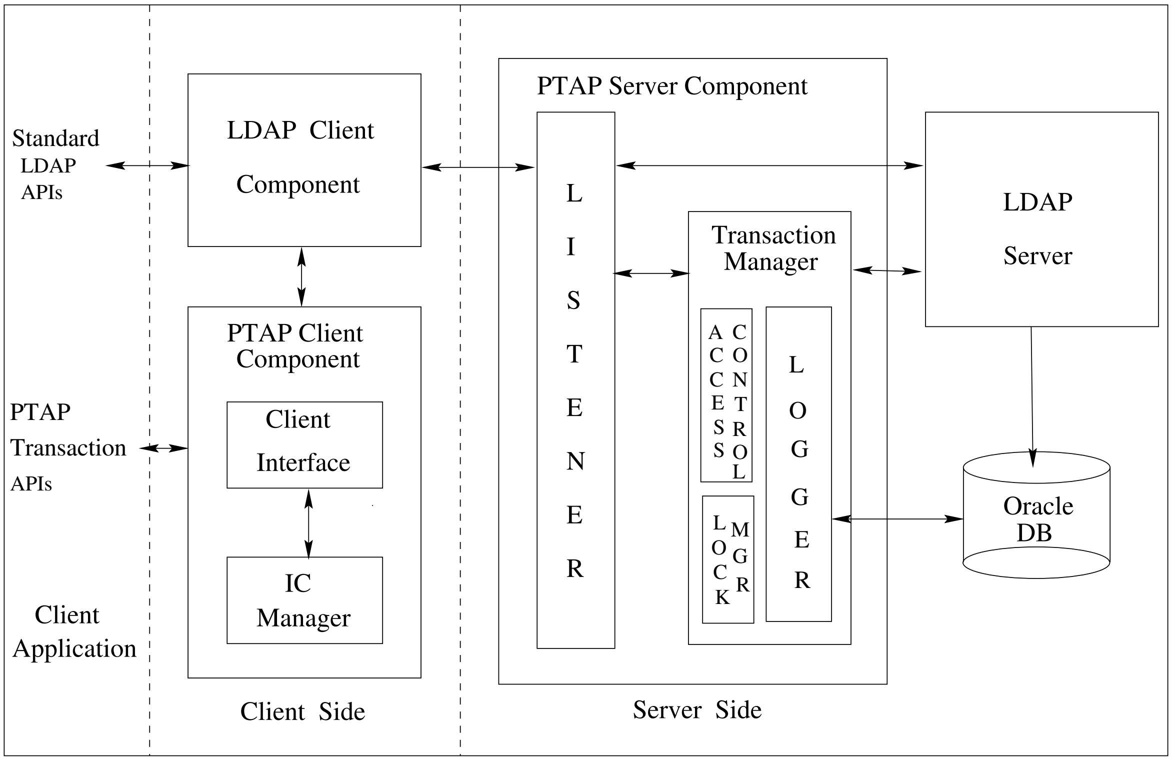


Figure 3.1: Design Overview

3.1.1 PTAP Client component

The client side component of PTAP provides APIs for transaction initiation and termination and for forming dependency among them. It also defines the callback functions that need to be implemented for enforcing application specific integrity constraints. The Integrity Constraint Manager (ICM) defines these callback functions. These functions should be implemented for application specific integrity manager. The Client Interface component provides an interface for the transaction APIs and the implementation of advanced transaction semantics.

3.1.2 PTAP Server component

The server side component of PTAP consists of `Transaction Manager (TM)` that uses `Logger` functions to log the operations in a transaction. It acts as a gateway to LDAP server. Client requests coming to LDAP server pass through this component. It redirects transaction related requests to TM and other requests are passed to the backend LDAP server. Return result from TM and LDAP server are returned back to the client. TM does some preprocessing (locking and logging) before executing LDAP operations on the backend LDAP server. In the post processing of these LDAP functions, TM checks the return code of these functions. If the return code is not LDAP_SUCCESS and the operation is part of a transaction, then it initiates undo of the transaction. The LDAP server has an option to either enable or disable the PTAP server. If PTAP is disabled, all the requests will be passed to the LDAP server without any pre or post processing.

The update requests(i.e. requests for add and delete) from the client are executed depending on their type as follows:

I **Update request not part of a transaction** In this case, the PTAP server will perform following actions:

a Before passing the request to LDAP

* Locks the entry involved in the update. (This is done by a dummy transaction.)
* Checks if the entry can really be updated. This is done by checking for preupdate conditions like, existence/non-existence of the entry before deleting/adding the entry.

b If [a] return SUCCESS, then the same request is passed to the LDAP server.

c After LDAP server returns

* Unlocks the entry involved in the update
* Returns success or error code to the client.

II **Update request is part of a local transaction** In this case, the update request is a part of a local transaction and involves updating entries on the local LDAP server. The PTAP server will perform following actions upon receiving such requests:

a Before passing the request to LDAP

* Locks the entry involved in the update.
* Checks if the entry can really be updated. This is done by checking for preupdate conditions like, existence/non-existence of the entry before deleting/adding the entry.
* If the checks succeed, then the operation is logged.

b If [a] returns LDAP_SUCCESS, then the operation is passed to the LDAP server.

c After LDAP server returns

* If LDAP server returns error code, then aborts the transaction to which it belongs to and aborts the dependent transactions.
* Returns the success or error code to the client.

Once the transaction is over (committed or aborted), the logs corresponding to the operations in the transaction are deleted by calling Logger functions.

III **Update request is part of a transaction on remote server** In this case, the update request from the client involves updates to entries on remote LDAP server as a part of remote transaction. The actions performed by the PTAP server on receiving such request are as follows:

– The request is passed to the remote LDAP server. It will be processed as in case II on the remote LDAP server.

– After LDAP server returns

* If LDAP server returns error code, then aborts the transaction to which it belongs to and aborts the dependent transactions.
* Returns the success or error code to the client.

3.2 PTAP configuration

The PTAP can be setup under the following configurations

- PTAP server with local LDAP server

- PTAP server without local LDAP server

These configurations are shown in fig 3.2.

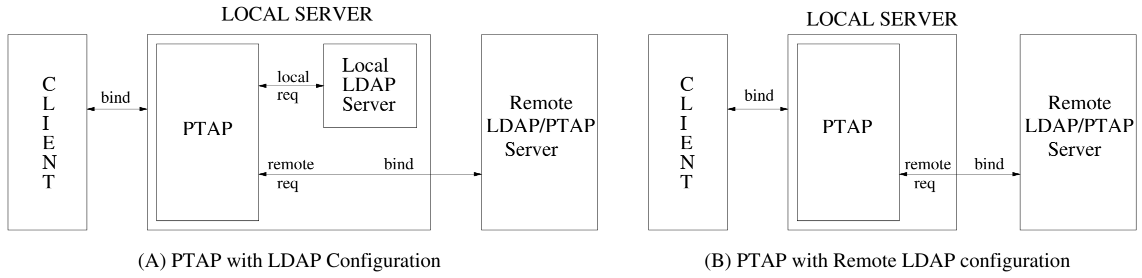


Figure 3.2: PTAP Setup

Client application binds itself to a PTAP server. This server is referred to as the local server. The local server returns to the client a connection handle (ld) which is subsequently used by the client for passing requests/operations to the server. The client does not bind to each of the servers on which it has to start a transaction. The client sends the transaction requests only to the local server. In the latter configuration, operations cannot be executed on the local server. The local server is used for maintaining only the transaction related information and actual transaction/operation being executed on remote server. For example, in the travel plan example, the local PTAP server can be used for starting the `travel_plan` transaction that co-ordinates other transactions i.e., `forward_airRes, return_airRes, hotel_res`. Operations are actually executed by these transactions on remote servers. If the client requests an operation/transaction on the local server (only in the former case) then the PTAP forwards the request to the local LDAP server that in turn executes the operations locally, otherwise it forwards the operation to the PTAP server that the client intends (remote server). The local server acts as if it is a client to the remote server and executes the operation on the remote server on behalf of the client.

We now explain each of the components of PTAP.

3.3 The Client Interface

The Client Interface exposes transaction interface to the user for initiating and terminating transactions and forming dependencies between them. The APIs that are exposed and their semantics are described below.

3.3.1 ldap_txn_begin

Synopsis

```
#include "transact.h"
returnCode
ldap_txn_begin (
        LDAP *ld,
        LDAP_CONST LDAP_TXNID *parentTxnId,
        txnDetails * newTxnDetails);
```

Description
This API starts a new LDAP transaction. `ld` is the connection handle with the LDAP server to which the request will be passed (called local LDAP server or local server hereafter). parentTxnId is a pointer to LDAP_TXNID structure giving the details of parent transaction. The LDAP_TXNID structure stores details of the transaction identifier. This is an opaque structure and the internal fields are not accessible to the client. The server uses it for identification and authorization. The server initializes this structure. The `txnDetails` structure stores the details of a transaction. It is defined as:

```
typedef struct TXNDETAILS
{
    LDAP_TXNID * ldapTxnId;
    ldapAddr srvrAddr;
    advTxnType txnType;
}txnDetails;
```

ldapAddr is defined as follows:

```
typedef struct LDAPADDR
{
    char hostName[MAX_HOSTNAME_LENGTH];
    int portNo;
}ldapAddr;
```

If parentDetails is NULL or the ldapTxnId field in NULL, the new transaction is treated as top level transaction. The top level transaction should always be initiated on local LDAP server(i.e. the ldap server for which the connection handle is obtained). If parentDetails is non-null, then it is checked if it has been initiated by the same client on the same connection as `ld`. newTxnDetails gives the details of the LDAP server (host and port No) on which it is to be initiated. Either the parent transaction or the new transaction should be on the local LDAP server. If the new transaction is on remote LDAP server, a new transaction is initiated on the remote LDAP server by the local LDAP server and a mapping is created from remote transaction to local transaction. The local transaction ID is returned to the client. All the operations that are part of this local transaction are passed to the remote server through the mapping. On successful return from ldap_txn_begin the ldapTxnId field of newTxnDetails will be pointing to LDAP_TXNID of the new transaction, which is to be passed as a parameter to the successive operations in the transaction. The parent child relationship is used to form dependencies and for permit and delegate operations if the transaction type of the new transaction is one of the standard transactions. The parent child relationship is also used to verify the validity of dependencies. This is explained in detail in the section for ldap_form_dependency.

Return value
Returns LDAP_SUCCESS if the new transaction could be started successfully, else returns one of the following error codes defined in transact.h

LDAP_INVALID_TXNID	: server does not have record of the passed txnId. It could have been already aborted/committed or it might not have been returned by a successful call to ldap_begin_txn.
LDAP_INVALID_PARENT_TXNID	: parent TxnId was non-null and invalid
LDAP_INVALID_CONNID	: the LDAP_CONNID is invalid

LDAP_NO_ENTRY_INFO	: could not read entry information from a local server for logging, before executing the operation
LDAP_INVALID_PARENT_TXN_ID	: invalid TXNID of parent
LDAP_NO_AUTHORIZATION	: a client trying to initiate a new transaction with parent not initiated by it
LDAP_TXN_WAITING	: txn waiting for other txns to commit
LDAP_ERR_STARTING_REMOTE_TXN	: could not start a new transaction on remote server
LDAP_ERROR_REMOTE_ADD	: error occurred while adding an entry on the remote server
LDAP_ERROR_REMOTE_DELETE	: error occurred while deleting an entry on the remote server
LDAP_ERR_COMMITING_REMOTE_TXN	: error occurred while committing a remote transaction
LDAP_CANNOT_DELEGATE	: trying to delegate to an incompatible transaction
LDAP_CANNOT_PERMIT	: trying to permit to an incompatible transaction
LDAP_ERR_READING_ENTRY_INFO	: could not read entry information from a local server for logging, before executing the operation
LDAP_INVALID_STATUS	: the expected status passed in ldap_txn_end is invalid
LDAP_ERR_ENTRY_TYPE_NOT_SUPPORTED	: addition or deletion of entries of type ref not supported
LDAP_ERR_CLOSING_CONNECTION	: error while closing connection
LDAP_ERR_OBJ_EXISTS	: entry to be added already exists
LDAP_ERR_LOGGING	: error occurred while logging an operation

3.3.2 ldap_txn_end

Synopsis

```
#include "transact.h"
returnCode ldap_txn_end (
        LDAP *ld,
        LDAP_CONST LDAP_TXNID * ldapTxnId,
        txnState state);
```

Description
This API is used to commit or abort a transaction. ldapTxnId is the txnId of the transaction to be aborted or committed. `state` holds one of the constants of the enumeration txnState defined in transact.h as:

```
typedef enum TXNSTATE
{
    COMMIT = 1,
    ABORT,
    END
}txnState;
```

The transaction and all its dependent transactions are aborted when the state specified is ABORT. If state specified is COMMIT, the transaction dependency is checked if it can be committed. If it can be committed, then two phase commit protocol is used to commit the transaction and its dependent transactions. Otherwise, the state of the transaction is set to READY_TO_COMMIT and error code returned is

LDAP_TXN_WAITING. The txnState can also be passed as END in which case the txn is not committed even if it can be committed. Its status is set to READY_TO_COMMIT and it waits for COMMIT or END status. Once the status of the transaction is set to READY_TO_COMMIT, no operation is allowed to be executed as part of the transaction.

Return value
If the state of the transaction could be successfully set to the required state, then LDAP_SUCCESS is returned. Otherwise, appropriate error code is returned. The details of return value code can be found in the section 3.3.1

3.3.3 ldap_add_t

Synopsis

```
#include "transact.h"
returnCode
ldap_add_t(
        LDAP *ld,
        LDAP_CONST char *dn,
        LDAPMod **attrs,
        LDAP_CONST LDAP_TXNID * txnId);
```

Description
This API is similar to the LDAP's standard ldap_add API except for the last parameter. The last parameter specifies the transaction id. The add operation is executed as a part of this transaction. It is checked that the transaction is initiated by the same client session as defined by the ldap session handle `ld`. If the transaction id is a local transaction that maps to a remote transaction, the add request is passed on to the remote server. If this operation fails, the transaction defined by txnId and all its dependent transactions are aborted. The callback functions are called before and after executing this API. If the txnId field is NULL, then it works as the standard ldap_add API with callback support.

Return value
If the add operation is executed successfully, as part of the specified transaction, then LDAP_SUCCESS is returned, otherwise appropriate error code is returned. The details of return value code can be found in the section for ldap_txn_begin.

3.3.4 ldap_delete_t

Synopsis

```
#include "transact.h"
returnCode
ldap_delete_t(
        LDAP *ld,
        LDAP_CONST char *dn,
        LDAP_CONST LDAP_TXNID * txnId);
```

Description
This API is similar to the LDAP's standard ldap_delete API except for the last parameter. The last parameter

specifies the transaction id. The delete operation is executed as a part of this transaction. It is checked that the transaction is initiated by the same client session as defined by the ldap session handle `ld`. If the transaction id is a local transaction that maps to a remote transaction, the delete request is passed on to the remote server. If this operation fails, the transaction defined by txnId and all its dependent transactions are aborted. The callback functions are called before and after executing this API. If the txnId field is NULL, then it works as the standard ldap_delete API, with callback support.

Return value
If the delete operation is executed successfully, as part of the specified transaction, then LDAP_SUCCESS is returned, otherwise appropriate error code is returned. The details of return value code can be found in the section for ldap_txn_begin.

3.3.5 ldap_modify_t

Synopsis

```
#include "transact.h"
returnCode
ldap_modify_t(
        LDAP *ld,
        LDAP_CONST char *dn,
        LDAPMod **attrs,
        LDAP_CONST LDAP_TXNID * txnId);
```

Description
This API is similar to the LDAP's standard ldap_modify API except for the last parameter. The last parameter specifies the transaction id. The modify operation is executed as a part of this transaction. It is checked that the transaction is initiated by the same client session as defined by the ldap session handle `ld`. If the transaction id is a local transaction that maps to a remote transaction, the delete request is passed on to the remote server. If this operation fails, the transaction defined by txnId and all its dependent transactions are aborted. The callback functions are called before and after executing this API. If the txnId field is NULL, then it works as the standard ldap_modify API, with callback support.

Return value
If the modify operation is executed successfully, as part of the specified transaction, then LDAP_SUCCESS is returned, otherwise appropriate error code is returned. The details of return value code can be found in the section for ldap_txn_begin.

3.3.6 ldap_permit

Synopsis

```
#include "transact.h"
returnCode
ldap_permit(
        LDAP *ld,
        LDAP_CONST txnDetails * txnFrom,
```

```
            LDAP_CONST txnDetails * txnTo );
```

Description
This API allows transaction `txnFrom` to access objects that are locked by transaction `txnTo`. This permits transaction `txnTo` to perform operations without waiting for transaction `txnFrom` to release the objects. Since entries on a server can be accessed by only the transactions on local server, both the transaction in ldap_permit have to be local transactions, or they can be mapping to transactions on same remote server.

Return value
LDAP_SUCCESS is returned if permit can be performed without any error. Otherwise, appropriate error code is returned.

3.3.7 ldap_delegate

Synopsis

```
#include "transact.h"
returnCode
ldap_delegate (
        LDAP *ld,
        LDAP_CONST txnDetails * txnIdFrom,
        LDAP_CONST txnDetails * txnIdTo );
```

Description
This API transfers responsibility for the operations performed by `txnFrom` to transaction `txnTo`. So, once `txnFrom` delegates to `txnTo`, it will be as if `txnTo` and not `txnFrom` has performed the operations for which `txnFrom` was responsible prior to the delegation. Again, since a local transaction can perform operations on only the local LDAP server, delegation cannot be done across transactions on different LDAP servers. i.e. both the transactions should be transactions on the same LDAP server or they should be mapping to transactions on same remote server.

Return value
LDAP_SUCCESS is returned if permit can be performed without any error. Otherwise, appropriate error code is returned.

3.3.8 ldap_formDependency

Synopsis

```
#include "transact.h"
returnCode
ldap_formDependency(
        LDAP *ld,
        LDAP_CONST txnDetails * txnIdFrom,
        LDAP_CONST txnDetails * txnIdTo,
        dependencyType type);
```

Description
This API forms dependency of specified type between transactions `txnIdFrom` and `txnIdTo`. The dependency among the transactions can be one of

CD (commit dependency)	: If both commit, `txnIdTo` cannot commit before `txnIdFrom` commits, but if `txnIdFrom` aborts `txnIdTo` may still commit.
AD (abort dependency)	: If `txnIdFrom` aborts `txnIdTo` also aborts.
GC (group commit)	: either both `txnIdTo` and `txnIdFrom` commit or neither commits.

We allow dependency formation between parent and child transactions and between sibling transactions. Dependencies can be formed only by the LDAP server of the parent transaction. All these dependencies are logged at the parent site. If a sub-transaction finishes execution, it sets its status as READY_TO_COMMIT and waits for ABORT or COMMIT message from the parent transaction.

Return value
If the dependency can be formed successfully, this returns LDAP_SUCCCESS. Otherwise, appropriate error code is returned.

3.4 The Integrity Constraint Manager

The ICM defines callback functions that are called before and after any operation is passed to the server side. These functions need to be implemented by ICM to provide application specific integrity enforcer. For example, in the travel plan example, if the user requests for addition of an entry of type `DoRes` then the ICM starts the transactions `travel_plan, forward_airRes, return_airRes, hotel_res` with the appropriate semantics, taking parameters from the entry that is requested for add. The PTAP provides default implementation of ICM by implementing these callback functions that return success. Signatures of the callback functions are given below:

```
returnCode
callback_preAdd (
        LDAP * ld,
        LDAP_CONST char *dn,
        LDAPMod ** attrs,
        txnDetails * parentTxnDetails);

returnCode
callback_preDelete (
        LDAP * ld,
        LDAP_CONST char *dn,
        txnDetails * parentTxnDetails);

returnCode
callback_postAdd (
        LDAP * ld,
        LDAP_CONST char *postDn,
        txnDetails * parentTxnDetails,
        returnCode rcRet);
```

```
returnCode
callback_postDelete(
        LDAP * ld,
        LDAP_CONST char *postDn,
        txnDetails * parentTxnDetails,
        returnCode rcRet);
```

The pre operations should return LDAP_SUCCESS_STOP to stop the further processing by the PTAP. On receiving LDAP_SUCCESS_STOP, PTAP does not execute the operation on the local server and returns LDAP_SUCCESS to the client. The return value of operation on the local server are passed to the post operations. The callback operations can take appropriate action depending on this value.

3.5 The Listener

The listener component of the PTAP server accepts connections/operations from the clients. If the operation is to be executed as a part of a transaction then it passes the request to the transaction manager. If it is a nontransactional request then it encloses the operation in a dummy transaction and makes it pass through the transaction manager component. If the operation is not enclosed in dummy transaction then that may lead to inconsistency in data. This can be explained by the following example. A transaction has added an entry to the directory and holds a lock on the entry. This lock information will be kept in the transaction manager. Another client requests for deletion of the entry (the request not being part of transaction). If we allow the request to directly go to the directory then it can delete the entry and this will lead to inconsistency in data. Therefore, we force the request to pass through the transaction manager that will detect the possible conflicts in the operations. The assumption that we make here is that clients do not directly talk to the underlying LDAP server.

3.6 The Transaction Manager

The figure 3.3 shows the logical view of the transaction manager component of the PTAP. The transaction manager consists of the following components:

Multi-Server Transaction	: The component that handles the transactions across multiple servers
Lock Manager	: The component that maintains information about the locked entries (the object descriptor list)
Access Control	: The component that maintains information about the transactions that permit other transactions to perform conflicting operations (the permit descriptor list)
Logger	: The component that does the logging of the operations

3.6.1 The Multi-Server Transaction Component

This component handles the transaction request across multiple servers. It handles how remote transactions must be started and how validation of the remote transaction requests is done. The details of this component are discussed in chapter 4

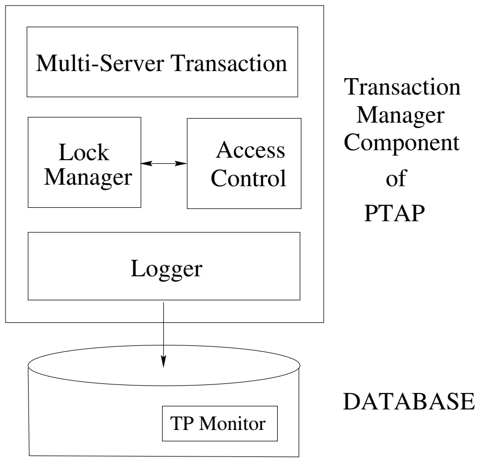


Figure 3.3: Components of Transaction Manager

3.6.2 The Lock Manager and the Access Control

These two component work in tandem. Any request for an operation on an entry first goes to the lock manager. Each entry is represented as a logical object identified by the `dn` of the entry. If the requested object does not exist then the lock manager creates a new object for the transaction and grants the lock on the object to the transaction. If the object exist then the lock manager finds from the access control component whether the transaction is permitted to lock the entry, grants the object to the transaction if permitted. Thus, the access control component provides the advanced transaction semantics. Strict two phase locking is used here, i.e. locks are released only at transaction abort or commit. The locking of the objects can lead to deadlocks under the following cases

- The requesting transaction and the transaction that holds the lock on the object are from the same connection and the transaction that holds the lock does not permit the requesting transaction
- The requesting transaction is from a different connection as the one of the holding transaction and waiting for the lock to be released leads to a cyclic wait

A deadlock avoidance approach is used here that declares a deadlock and returns error that leads to the abortion of the transaction that requested the lock.

3.6.3 The logger component

The logger component ensures transaction level atomicity. The operation level atomicity is handled by the underlying directory, i.e. if there is a failure in the midst of performing an operation, then the underlying directory will undo the partial results of the operation. The logger ensures the semantic undo of the transactions in accordance with the advanced transaction model semantic. The logger provides functions to perform operation level logging, for deleting logs of operations of a transaction, for deleting logs of an operation and for reading the log records. The logger provides following interface for logging log_begin, log_operation, log_end, get_operations, deleteLogForTxn, deleteLogForOpn. Write Ahead logging is used, the operations being logged before they are executed on the directory. The **Logger** component only logs the UNDO log records. The REDO log records are not required for the assumption that the atomicity of the operation is taken care by the underlying database. Only log records that are required to undo a transaction are logged by the logger. An operation is logged and actually executed only if it satisfies the precondition i.e. existence/non-existence for a delete/add operation. The current implementation of logger does not provide full support for crash recovery. After a failure of a transaction (failure of an operation) the undo of the transaction is performed. The server reads the log records and constructs a logical inverse operation. This operation is then executed as a part of the aborting transaction. An undo operation can fail only if the log record for the operation exist but the operation is not actually performed on the database. This is possible only if

[a] there was a server crash after logging the operation and just before the operation was actually executed

[b] there was a server crash in the midst of performing the operation

In both these cases we can neglect the error of the undo operation since in case [a] undo is not actually required and in case [b] the underlying database will undo the operation.

The next chapter discusses the implementation details of the PTAP server side component.

Chapter 4

Implementation

This chapter describes the data structures and algorithms used to implement the transaction primitives. The client side data structures mainly maintain mapping between the LDAP_TXNID returned by the server after successful execution of ldap_txn_begin and the details of the transaction (like host name and port number). This information is used to check the validity of LDAP_TXNID passed to the subsequent operations in the transaction. We focus our discussion here on the data structures and algorithms used by the `TM`.

4.1 PTAP Design considerations

The main consideration of the design is to make the PTAP server transparent to the client application. For the client application, PTAP server is an LDAP server providing transaction support. The client application talks with the PTAP server as it does with any other LDAP server. The PTAP server in turn talks to the backend LDAP server. PTAP maintains a unique connection handle for each client connection and maps this connection handle to a connection handle with the backend LDAP server. The `TM` component of PTAP maintains status of all active transactions in a client connection. An LDAP entry is the basic unit that can be locked by a transaction. The distinguished name of the entry is used as unique identifier of the entry. All the operations in a transaction are executed atomically maintaining strict consistency irrespective of the type of the transaction. Failure of a single operation in the transaction causes abortion of all the operations in the transaction. But there exists weak consistency among transactions and subtransactions. The form of consistency depends on the type of the transaction and on dependencies among transactions and subtransactions formed using the transaction primitives.

4.2 Data Structures

Figure 4.1 shows important data structures that are maintained at server side. Each data structure is explained in detail in the following subsections.

4.2.1 Connection descriptor (connDesc)

This structure maintains the connection status of a client with the server. Each client connection is uniquely identified by this connection descriptor. The structure has following fields:

connId : The identifier uniquely identifying the client connection to the PTAP server. The PTAP

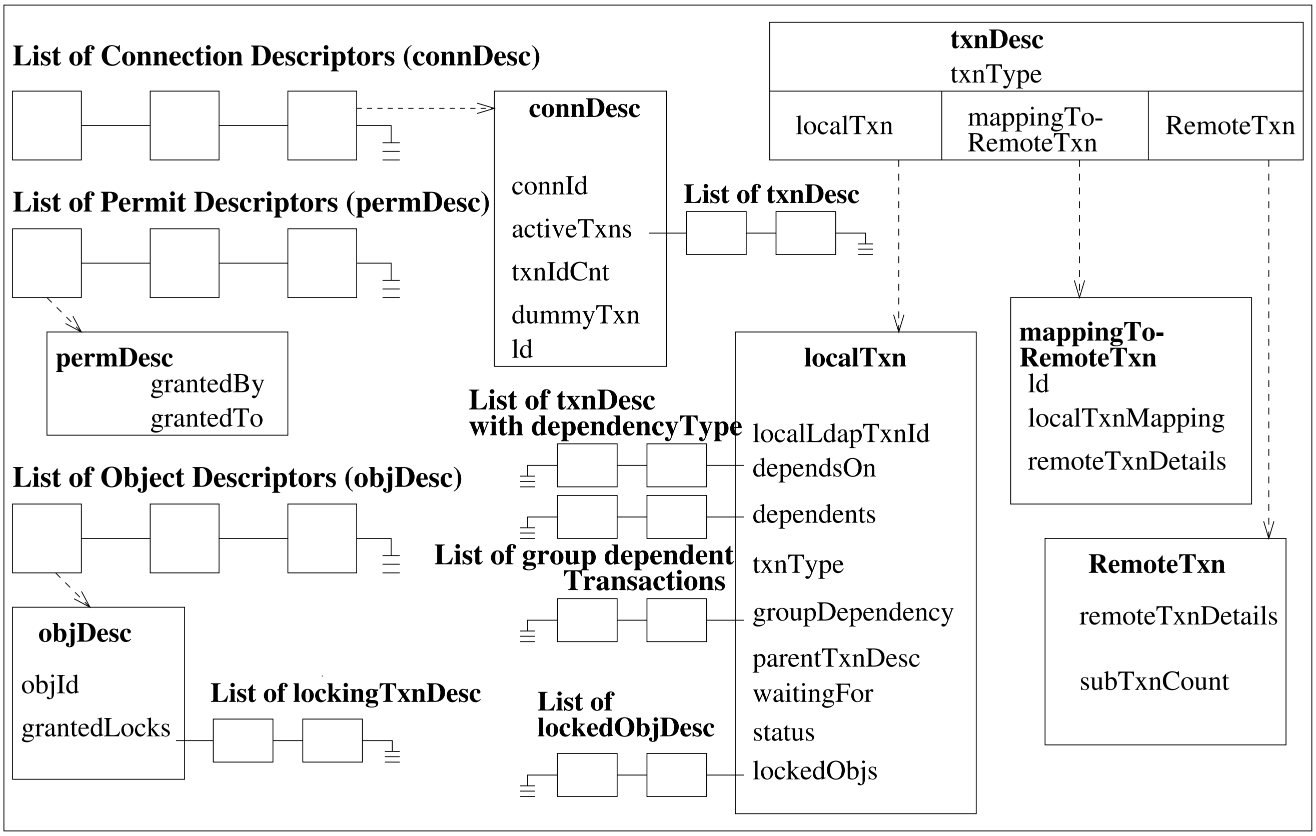


Figure 4.1: Server side data structures

server assigns a unique connection identifier to each connection request from clients and passes it to the `TM`. The `TM` stores this identifier in connId field of the connDesc.

activeTxns : This maintains the list of `transaction descriptors` of active transactions of this client connection.

txnIdCnt : Count of number of successful begin_txn calls made by this client connection to the PTAP server. It is incremented for each successful begin_txn call made by this client and used as a transaction identifier for the new transaction.

dummyTxn : This is the `transaction descriptor` of a dummy transaction used for executing an operation that is not part of any transaction. As the dummy transaction consists of a single operation that is assumed to be an atomic operation, logging is not needed for operation in dummy transaction. Dummy transaction locks an entry that the operation in it is going to operate on to maintain serializability between operations in other transactions.

ld : This is the connection handle with the backend LDAP server. It is used to query the backend server for pre-update conditions like existence of an entry before delete operation.

4.2.2 Transaction Descriptor (txnDesc)

This structure maintains information about a transaction. Depending on the location of a transaction with respect to the local server, a transaction descriptor can be any one of the three types: LOCAL_TXN, REMOTE_TXN or REMOTE_TXN_MAPPING. This structure represents one of the three types of transaction descriptors and has a type tag that gives the type of the transaction.

The three types of transaction descriptors are described below.

Local transaction (localTxn)

This structure maintains information about a local transaction i.e. transaction started by the PTAP on its local site. Only local transaction can access entries on the local server. This transaction descriptor is equivalent to the transaction descriptor of the single server case described in [Her01]. The fields in this descriptors are as follows:

localLdapTxnId	: This is the LDAP_TXNID of a local transaction uniquely identifying a transaction on the local server.
txnType	: This is an enumeration that gives type of the transaction. It holds one of the constants as NESTED_TXN, NESTED_SAGA or COOPERATING_TXN.
parentTxnDesc	: This is the `transaction descriptor` of the parent transaction. It could be one of REMOTE_TXN, LOCAL_TXN or REMOTE_TXN_MAPPING.
lockedObjs	: This is a list of `locked object descriptors` of objects locked by this transaction. It also stores the mode in which the object is locked. This list is searched to check if an object is already locked by a transaction.
status	: This is an enumeration that gives internal state of the transaction. It can be one of INITIALIZED, READY_TO_COMMIT (the transaction is ready to commit but may commit or abort depending on the status of its `dependsOn` transactions), ABORTING, COMMITTING.
dependsOn	: This is the list of `transaction descriptors` on which the transaction depends with dependency type CD or AD. This list should be empty for the transaction to be able to commit.
dependents	: This is a list of `transaction descriptors` which are dependent on this transaction for abort or commit. Whenever a transaction is about to commit, it traverses its dependent list and removes itself from the `dependsOn` list of the dependents. If this makes the `dependsOn` list of the dependent empty and its `groupDependency` list is also empty, then dependent transaction is also committed.
groupDependency	: This is the list of `transaction descriptors` that form group dependency among themselves. All the transactions in this list should be READY_TO_COMMIT for all the transactions to be able to commit. Even if one of the transactions in the group aborts, all the transactions in the group are aborted.
waitingFor	: This is the transaction descriptor of the local transaction that has held a lock on an entry needed by this transaction. This field is used to maintain the `wait-for` graph and check for deadlocks.

Mapping to remote transaction (mappingToRemoteTxn)

This structure is a descriptor of a local transaction that maps to a remote transaction initiated by the local PTAP server. The local transaction ID is returned to the client application. Whenever an operation is requested as a part of this local transaction, the request is passed to the remote server using this mapping descriptor. As a remote transaction cannot access the local LDAP entries, the lockedObjs fields of this descriptor is not used and it is always empty. The fields in this structure are as follows:

ld	: This is the connection handle with the remote LDAP server. Client requests that are part of this transaction are passed to the remote LDAP server using this connection handle.
localTxnMapping	: This is the `localTxn` descriptor that maps to the remote transaction. LDAP_TXNID of this localTxn is returned to the client.
remoteTxnDetails	: This field gives the details of the remote transaction such as host name and port number of the remote LDAP server, LDAP_TXNID of the transaction started on the remote server. Whenever a request for an operation as a part of local transaction is made, the request is passed to the remote server as a part of the remote transaction. `ldap_txn_begin` involving a remote transaction, always logs the remote transaction id. The local transaction IDs are used only for mapping.

Remote transaction (remoteTxn)

This structure is a descriptor of a remote transaction which is not initiated by the local PTAP. `remoteTxn` descriptor can only act as a parent descriptor to local transaction descriptors and cannot lock an entry or take part in any of the dependencies. The fields in this transaction descriptor are as follows:

remoteTxnDetails	: This field gives the details of the remote transaction such as host name and port number of the remote LDAP server, LDAP_TXNID of the transaction on the remote server.
subTxnCount	: This gives count of the sub-transactions of this remote transaction on local server. Whenever a subtransaction of the remote transaction ends, the subTxnCount of the remote transaction is decremented. If the count becomes zero, it is removed from the active transaction list.

4.2.3 Object Descriptor (objDesc)

Each local transaction descriptor maintains list of object descriptors of objects that it has locked. Each object descriptor has following fields:

objId	: This is the distinguished name of an LDAP entry that uniquely identifies the entry.
grantedLocks	: This is the list of `locking transaction descriptors` that provide information about transactions that have locked this object.

4.2.4 Locking transaction descriptors (lockingTxnDesc)

This data structure describes transactions that lock an object. An objDesc maintains a list `lockingTxnDesc` of the object. It includes following fields:

tD : Transaction descriptor of transaction that has locked this object.
mode : Mode of locking. It could be READ/MODIFY.
next : Next transaction locking the object.

4.2.5 Locked object descriptors (lockedObjDesc)

This data structure describes objects locked by a transaction. A local transaction descriptor `localTxn` maintains a list of `lockedObjDesc` for the objects it has locked. It includes following fields:

oD : Object descriptor of object that has been locked by this transaction.
mode : Mode of locking. It could be READ/MODIFY.
next : Next object locked by the transaction.

4.2.6 Permit descriptors (permDesc)

The TM maintains list of permission descriptors. This data structure provides information about permissions granted so that transactions can perform conflicting operations on objects. Since only local transactions can access objects on local LDAP server, only local transactions can permit to other local transactions. Mapping transactions that map to remote transactions on same remote server can also permit to each other.

A permit descriptor contains following fields:

grantedTo : Transaction descriptor of a local transaction that is granted the permission.
grantedBy : Transaction descriptor of transaction that grants the permission.
operation : The operation for which the permission is granted. It can be READ, MODIFY or ALL. Default value is ALL.

4.3 Implementation of transaction primitives

This section describes server side implementation of the transaction primitives.

4.3.1 ldap_txn_begin (ld, patentTxnId, newTxnDetails)

Following conditions are checked before initiating a new transaction:

- If parent transaction ID is `null`, then the new transaction must be started on the local server as a topmost transaction.
- If the parent transaction ID is `non-null` and parent transaction is on the local server then a local transaction descriptor must exist for this transaction in the active transaction list of the current connection. This check assures that the parent transaction is a valid transaction already started on the local server.

- If the parent transaction ID is `non-null` and parent transaction is on the remote server, then a remote or mapping transaction descriptor must exist for the parent transaction. If the parent transaction descriptor is a remote transaction descriptor, then the new transaction must be on the local server. This condition checks validity of the remote parent transaction.
- If all the above conditions are checked and the parent transaction descriptor is a `non-null` valid transaction descriptor, then its status should be INITIALIZED. This check assures that the transaction is not terminated yet.

If the parent transaction satisfies all the above conditions, the new transaction is initiated as follows:

- Increment the txnIdCnt of the connection descriptor.
- If the new transaction is to be started on a remote server, connect to the remote PTAP server and send request for a new transaction on the remote server. Initialize a new MAPPING_TXN_DESC with the local and remote transaction IDs.
- If the new transaction is to be started on local server, initialize a new LOCAL_TXN.
- If the new transaction descriptor could be initialized successfully on remote or local transaction descriptor, log the operation.
- Add the new descriptor to the `activeTxns` list of current connection descriptor and return the local transaction ID.

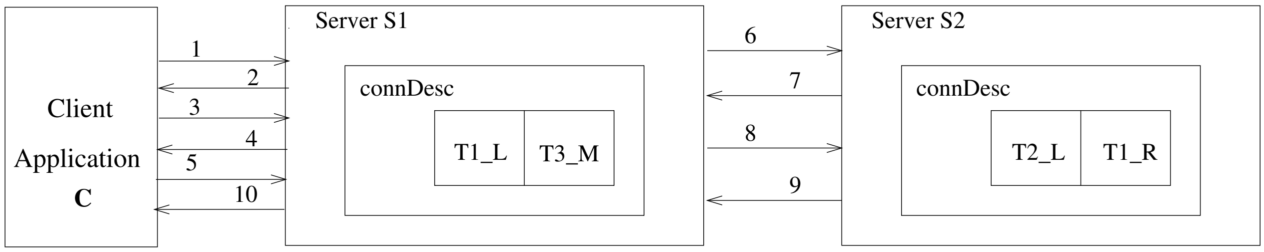
Figure 4.2 shows steps executed for initiating a remote transaction as a subtransaction of a local transaction using PTAP It also shows the data structures that are initiated at each PTAP server. Suffix `_L` indicates transaction descriptor of type LOCAL_TXN. Suffix `_M` indicates transaction descriptor of type REMOTE_TXN_MAPPING. Suffix `_R` indicates transaction descriptor of type REMOTE_TXN.

4.3.2 ldap_txn_end(ld, ldapTxnId, expectedStatus)

Different actions are performed depending on the value passed for `expectedStatus`. `expectedStatus` can hold one of END, COMMIT or ABORT. The actions performed on different `expectedStatus` value are described below.

END

If the expected status is END, then the status of the transaction is set to READY_TO_COMMIT. This status indicates that all the operations in this transaction are logged and the transaction can be committed or aborted. The transaction waits for a COMMIT or ABORT request. No more operations are allowed as a part of this transaction once its status is set to READY_TO_COMMIT. If `ldapTxnId` is transaction ID of a transaction of type `REMOTE_TXN_MAPPING`, `ldap_txn_end` request is sent to the remote LDAP server with expectedStatus as END. `expectedStatus` is passed as `END` to execute the first phase of the `two phase commit protocol`.



1 Bind request from C to S1
2 Connection handle returned to C by S1
3 ldap_txn_begin request from C to S1 for a txn on S1
4 Transaction ID T1_L returned to C by S1
5 ldap_txn_begin request from C to S1 for a txn on S2 with T1_L as the parent
10 Transaction ID T3_M returned to C by S1

6 Bind request from S1 to S2
7 Connection handle returned to S1 by S2
8 ldap_txn_begin request from S1 to S2 with T1_L as the parent
9 Transaction ID T2_L returned to S1 by S2

Figure 4.2: Steps for initiating a remote transaction

Commit

If the expected status is COMMIT, dependencies of the transaction are checked to see if the transaction can actually commit. Transactions are committed using `two phase commit protocol` [SKS97]. If the transaction can not `actually commit`, status of a transaction is set to READY_TO_COMMIT and LDAP_TXN_WAITING is returned. If `ldapTxnId` is transaction ID of a transaction of type `REMOTE_TXN_MAPPING`, `ldap_txn_end` request is sent to the remote LDAP server with `expectedStatus` set to END. This is the first phase of the `two phase commit protocol`. When a transaction and all it's dependents can `actually commit`, it sends a COMMIT request to all its dependent transactions that are remote transactions and executes the commit operation on all dependents that are local transactions. This is the second phase of the two phase commit protocol.
A transaction can `actually commit` only if all of the following conditions are satisfied.

- Status of the transaction is READY_TO_COMMIT.
- If the transaction does not depend on any other transaction for COMMIT and ABORT i.e. its `dependsOn` list is empty or all of its `dependsOn` transactions can `actually commit`.
- If the group dependency list of the transaction is empty or the group status is READY_TO_COMMIT. (A group status is READY_TO_COMMIT only if each member of the group can `actually commit`)

The definition of an `actually committing` transaction is a recursive definition. The recursion ends when, a transaction with dependsOn and group dependency list as empty is found. The recursion may not end due to cyclic dependencies among transactions. For e.g. if transaction `t1` depends on transaction `t2` and transactions `t2 and t3` are group dependent. If transaction `t3` depends on transaction `t1` with ABORT dependency, then all the three transactions will become READY_TO_COMMIT but won't commit as they

are waiting for each other in cycle for commit. If such a cycle is detected, then all the transactions in the cycle and their dependents are committed. Figure 4.3 gives the pseudocode for a recursive function that checks if a transaction can actually commit or it is involved in a dependency cycle, each of which has status as READY_TO_COMMIT, in which case it returns TRUE, otherwise, it returns FALSE.
It takes two parameters. The first parameter is the descriptor of the transaction that is to be checked if it can `actually commit`. The second parameter is the the list of transactions that depend on each other and there exist a path from the first transaction to the last transaction in dependency graph(The graph is maintained implicitly by the data structures). The recursion terminates if the checknode is not READY_TO_COMMIT (in which case it returns FALSE) or can `actually commit` (in which case it returns TRUE). If any of the `dependsOn` transaction or `group dependency transaction` of the checknode along with a transaction in the initiator list forms a cycle, then that transaction is not checked for further dependency. This is because, detection of a cycle means that the transaction can `actually commit`.
The cycle detection algorithm is not executed when the dependency is formed, because the dependency cycle may not lead to deadlock at the time of actually execution of the transactions as some of them may abort not causing all of them to abort but breaking the cycle.

If a transaction can actually commit then it performs following cleanup operations while committing.

- Removes it's trace from all txnDesc in the list of active transactions i.e. removes it's entry from any transaction which depends on it so that the dependent transaction may commit.
- Deletes it's log records from the log table and remove it's entry from the active transactions.
- Unlock all the entries that it had locked.

For a group commit, the last transaction in the group which can commit sets the group status to READY_TO_COMMIT and performs the above operations for each member of the group. Figure 4.4 gives the pseudocode for commit.

Abort

A transaction is aborted when ldap_txn_end is called with expected state as ABORT. It is also aborted by the server internally, when an operation in a transaction fails. Also, when a transaction is aborted, all its dependents that have dependency type AD (abort dependency) are aborted. If the aborting transaction is part of a group dependency, then all transactions in the group are aborted. The aborting transaction is removed from the dependsOn list of all dependents that have dependency type CD (Commit dependency). If its dependsOn list becomes empty after this removal, the dependent transaction is committed. Figure 4.5 gives the pseudocode for abort.

Unlike the [MHL^{+}92] system, aborting of transactions does not require to do redo operations because all the log operations are written to the disk immediately. So, whenever an operation is completed, its log is on the disk.

To abort a transaction, the server reads the log records and constructs a logical inverse operation. This operation is then executed as a part of the aborting transaction. An undo operation can fail only if the log record for the operation exist but the operation is not actually performed on the database. This is possible only if

```
boolean canActuallyCommit(checknode: txnDesc,
    initiators: list of txnDesc)
        if checknode.status != READY_TO_COMMIT
            return FALSE
        if all in the dependsOn list and group dependency list have
            status READY_TO_COMMIT
            if dependsOn list of checknode not empty
                add checknode to the initiators list
                for each txnDesc t in dependsOn list
                    if t does not exists in the initiator
                    i.e. it does not form a cycle
                        if canActuallyCommit(t, initiators) = FALSE
                            return FALSE;
            if (checknode part of group dependency and the group is not
            yet checked)
                for each txnDesc t in the group dependency
                    if dependsOn list of t is not empty
                        if t does not exist in the initiator
                            if canActuallyCommit(t, initiators) = FALSE
                            /* A transaction depends on a member in group dependency
                            * means that the transaction depends on each on the group.
                            * so check for each member in the group */
                            return FALSE;
        else
            return FALSE;
        return TRUE;
```

Figure 4.3: Pseudocode to check if all `dependsOn` transactions can commit

[a] there was a server crash after logging the operation and just before the operation was actually executed

[b] there was a server crash in the midst of performing the operation

In both these cases we can neglect the error of the undo operation since in case [a] undo is not actually required as the operation is not performed at all. In case [b] the operation will not be completed as an LDAP operation is assumed to be atomic.

4.3.3 ldap_form_dependency(ld, txnIdFrom, txnIdTo, dependencyType)

Dependencies can be formed between sibling transactions and parent and child transactions. Also, the dependencies can be formed only on the server where both the transactions are initiated. These conditions

```
commit(txnToBeCommited)
    if canActuallyCommit(txnToBeCommited) == FALSE
        return LDAP_TXN_WAITING
    if(txnToBeCommited is part of group and the group status is not
        READY_TO_COMMIT)
            set group status to READY_TO_COMMIT
            for each txnDesc t in group
                doCleanup(t)
    else
        doCleanup(txnToBeCommited) /* txn not part of a group. */

doCleanup(txnToBeCleanedup)
    remove log records of txnToBeCleanedup
    unlock all objects locked by txnToBeCleanedup
    for each dependent t of txnToBeCleanedup
        remove txnToBeCleanedup from dependsOn list of t
        if dependsOn list of t becomes empty
            commit(t)
    remove txnToBeCleanedup from list of activeTxns
```

Figure 4.4: Pseudocode for committing a transaction and its dependent transactions

can be satisfied by ensuring that transaction descriptors of both the transactions taking part in the dependency are LOCAL_TXN or REMOTE_TXN_MAPPING. Following modifications to the data structure are done:

- If the dependency type is CD or AD, then `txnIdFrom` is added to the `dependent` list of `txnIdTo`. `txnIdTo` is added to the `dependsOn` list of txnIdFrom list.
- If the dependency type is GD, then the group dependencies of both the transactions are merged. If they are not part of any group dependency, new group is formed with these two transactions in the group.

4.3.4 writeLock(txnDesc, objDesc)

The locking model used is the `Multilevel locking model`[Wei91] with LDAP entry level locking. This API is not exposed to the client. It is called internally by the server for all update operations. If the operation is not part of a transaction, then a dummy transaction is started that locks the object. Following changes are made in the data structure.

- Add `lockingTxnDesc` for the transaction that requested a lock (i.e. `txnDesc`) in the `granted-Locks` list of the object descriptor.
- Add `lockedObjDesc` for the object on which lock was requested (i.e. `objDesc`) in the `locke-dObjs` list of the requesting transaction descriptor.

```
abort(txnToBeAborted)
    if(txnToBeAborted is part of group and the group status is not
        ABORTING)
            set group status to ABORTING
            for each txnDesc t in group
                doCleanupAbort(t)
    else
        doCleanupAbort(txnToBeAborted)
doCleanupAbort(txnToBeCleanedup)
    remove log records of txnToBeCleanedup
    unlock all objects locked by txnToBeCleanedup
    for each dependent t of txnToBeCleanedup
        remove txnToBeCleanedup from dependsOn list of t
        if dependencyType of t on txnToBeCleanedup is CD
            if dependsOn list of t becomes empty
                commit(t)
        else
            abort(t)
    remove txnToBeCleanedup from list of activeTxns
```

Figure 4.5: Pseudocode for aborting a transaction and its dependent transactions

If a transaction cannot obtain a lock on an object due to conflicting lock held on the object by another transaction, then it is checked if waiting of the requesting transaction can cause a deadlock. This check is made by searching for a cycle in the wait-for graph maintained by `waitingFor` field in `localTxn`. If a cycle is found, then error code is returned that aborts the transaction and prevents deadlock.

4.3.5 ldap_permit(ld, txnIdFrom, txnIdTo)

Since only local transaction can access the entries on the local server, both the transactions involved in permit operation have to be LOCAL_TXN. A permit descriptor (permDesc) that includes (txnIdFrom, txnIdTo, WRITE_MODE) is inserted into the permit descriptors list. This permits the `txnIdto` to access in WRITE_MODE all objects that `txnIdFrom` has locked without waiting for `txnIdFrom` to release the object.

4.3.6 ldap_delegate(ld, txnIdFrom, txnIdTo)

Since a local transaction can perform operations on only the local LDAP server, delegation cannot be done across transactions on different LDAP servers. i.e. both the transaction descriptors should be LOCAL_TXN or REMOTE_TXN_MAPPING mapping to transactions on same remote server. Following modifications to the data structure are done:

- Each entry in the `lockedObjs` list of `txnIdFrom` is removed and added to the `lockedObjs` list of `txnIdTo`. Also, for all these objects, the `grantedLocks` list is modified to include `txnIdTo` instead of `txnIdFrom`.
- Any entry in the permission descriptors list of the form (txnIdFrom, txnIdOther, WRITE_LOCK) is changed to (txnIdTo, txnIdOther, WRITE_LOCK).

Chapter 5

Testing

This chapter describes the test client and the example used for testing. The test cases mainly include transactions with subtransactions on remote servers having different dependencies among the transactions and subtransactions. Abort of transactions is tested by having some operations in the transactions that will fail, for e.g., addition of an existing entry or deletion of a non-existing entry.

5.1 The template test client

To make the testing process easy and to cover maximum test cases without rewriting the client, we have written a library `libreadop.so` that exports the following function.

void read_config(LDAP * ld, char filename[])

This function reads a file with name `filename` containing commands in a fixed format. The commands are interpreted one at a time and the corresponding transaction APIs are executed on LDAP server using the connection handle `ld`. The template client uses this function to execute commands in a file. It takes following command line options:

-H : LDAP url of the server to be connected
-D : The bind DN to be used to bind to the LDAP server
-f : The file name containing the commands to be executed

5.1.1 Sample command file

Contents of a sample command file are shown in figure 5.1. The lines that start with a '#' are comments. First word in the line defines the command to be executed. The first parameter to a begin command is the transaction number. The client does not need to know the LDAP_TXNID that is returned by ldap_txn_begin. All the transactions are referred by their transaction numbers. If invalid constants are passed in place of constants like transaction type or dependency type, '0' is passed. Meaning of each parameter is defined in the sample file in the comment before the command. The sample file uses example explained in next section. It cancels a travel reservation and does a hotel booking by adding an entry into the hotel directory.

```
#begin <newTxnNo><parentTxnNo><newTxnHost><newTxnPort><newTxnType>
begin 1 0 localhost 1030 NESTED_TXN
begin 2 1 localhost 1030 NESTED_TXN
#formDependency <txnFrom><txnTo><dependencyType>
formDependency 1 2 GD
#delete <dnOf entry to be deleted><txnOfOpn>
delete "name=passengerName,fn=4820,ou=Delta Airlines,o=XYZ Agency" 2
#end <txnNo><expected status>
end 2 COMMIT
begin 3 1 localhost 1030 NESTED_TXN
formDependency 1 3 GD
#add <ldifFile><txnOfOpn>
add /home1/anandi/sqlIC.ldif 1
end 3 COMMIT
end 1 COMMIT
```

Figure 5.1: Sample client command file

5.2 The test example

We have used example of the tourist agency discussed in chapter 2 which maintains directory with referrals to different hotel directories and travel agency directories. Figure 5.2 shows the simplified view of part of this directory.

A travel plan involves making a forward journey reservation, hotel reservation and return journey reservation. The travel plan is implemented as the main transaction. Forward journey reservation, hotel booking and return journey reservation are subtransactions of the top level transaction with different dependencies among them depending on the scenario being executed. For journey reservation, an entry is added to the travel agency directory and for hotel booking, an entry is added into the hotel directory. Undo of a transaction involves cancellation of a reservation i.e. deletion of the added entry.

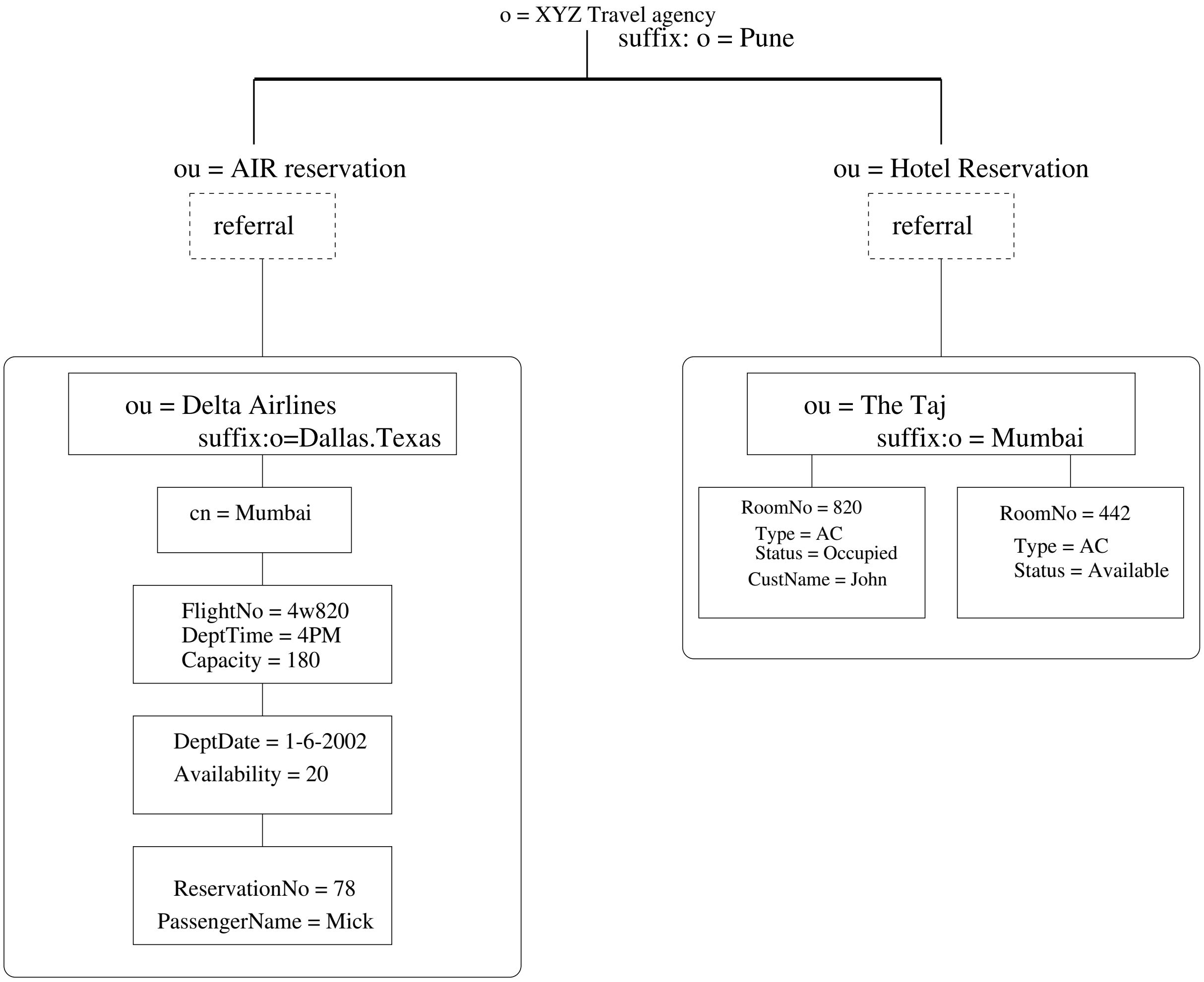


Figure 5.2: Example directory for travel plan

Chapter 6

Conclusions and future work

6.1 Conclusion

Our system provides basic framework for implementing an integrity constraint manager for LDAP servers using advanced transaction modeling support. We have implemented a component, Predefined Trigger Action Protocol (PTAP) that can be easily plugged in on top of any LDAP server. PTAP server is transparent to the client application i.e., for the client application, PTAP server is an LDAP server providing transaction support. The client application talks with the PTAP server as it does with any other LDAP server. The PTAP server in turn talks to the backend LDAP server for LDAP operations. PTAP provides the basic APIs for transaction support. The transaction support provided by PTAP includes standard advanced transaction models (Nested Transaction, Nested Saga, Co-Operating Transaction) and primitives that can be used to implement customized transaction model. The primitives are defined to support transactions across multiple LDAP servers. It provides operation level logging for each operation in a transaction. These logs are used to undo operations in a transaction in case of failure of a transaction, to maintain atomicity. The logger component provides APIs to log operations, delete an operation, delete all operations in a transaction, read operations in a transaction.

6.2 Future work

Possible future work includes making the system more robust by providing crash recovery, making it efficient and enhancing the IC manager to make it more intelligent and application independent and enhancing the PTAP server with more flexibilities in the primitives. This section describes the enhancements to be done.

6.2.1 Crash recovery

The current system does not provide crash recovery of the server. Logging the dependencies can provide easy recovery when used with the current logs. The following extra APIs need to be added to provide recovery.

- The dependencies need to be logged. Since the dependencies are formed only at the parent PTAP site, the dependencies need to be logged only at the parent site.

- In case of dependencies involving REMOTE_TXN_MAPPING, logging of only the remote transaction IDs need to be done as the local transaction IDs are used only for mapping.
- If a remote server crashes that had started a subtransaction and has status as READY_TO_COMMIT in the log (i.e. it was waiting for either ABORT or COMMIT from the parent transaction) it needs to decide if it can commit or abort depending on the dependencies formed for it on the parent site. The following API needs to be supported.
  status getFinalStatus(LDAP * ld, LDAP_TXNID * txnId);
  The parent PTAP server should check if this transaction depends on any transaction and if it can abort or commit.
- The delegate primitive should modified to change the history logs [MR97].

6.2.2 Efficiency

Currently, all the transaction APIs are synchronous APIs, i.e. they do not return until the execution is finished. This allows only one subtransaction of a transaction to be executed at a time. This can be changed and the subtransactions can be made to execute in parallel.

6.2.3 IC Manager

The IC manager implements the callback functions to implement application specific integrity constraints. The IC manager is different for different applications having different integrity constraints. The IC manager can be made more intelligent that understands integrity constraint rules specified in some language and implements appropriate transaction model to satisfy those integrity constraints. The following issues need to be addressed for this:

- A formal method to specify integrity constraints in LDAP need to be designed. LDAP applications need to be explored to find possible types of integrity constraints in LDAP.
- A formal method to map an integrity constraint to appropriate transaction model and decide the dependencies between transactional steps needs to be defined. Different applications of LDAP need to be studied to explore how the primitives can be used to implement the integrity constraints efficiently.

If PTAP is to be used with universal directory server that handles databases as well, the PTAP component needs to be more generic to handle transactions across databases in addition to the transactions in LDAP.

6.2.4 Enhancements

The PTAP server can be enhanced to provide more flexibility in primitives. The `form_dependency` API can support more dependency types like `precedence`. It can be made generic to accept the dependency type and the implementation for the semantics of the dependency.

The current assumption that all the requests that go to the LDAP server go through PTAP may not be practical. We need to find a way that assures that PTAP is notified of all the requests that come directly to the LDAP server. This will allow the PTAP to take appropriate actions on such requests too and will make sure that the directory is consistent.

References

[AFG+99] R. Arlein, J. Freire, N. Gehani, D. Lieuwen, and J. Ordille. Making ldap active with the ltap gateway: Case study in providing telecom integration and enhanced services. In Proc. Workshop on Databases in Telecommunication, September 1999.

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. Asset: A system for supporting extended transactions. ACM SIGMOD International Conference on the Management of Data, p. 44-54, December 1994.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, Calif, 1992.

[Her01] Anandi D. Herlekar. Enforcement of integrity constraints in ldap using advanced transactions. M.Tech. Stage - II report, August 2001.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Transactions on Database Systems, 17(1), March 1992.

[MR97] Cris Pedregal Martin and Krithi Ramamritham. Delegation: Efficiently rewriting history. In Proceedings of the Thirteenth International Conf. on Data Engineering, p. 266-275, Birmingham, UK, April 1997.

[RC96] K. Ramamritham and P. Chrysanthis. Advances in concurrency control and transaction processing. IEEE Computer Society Press, September 1996.

[SKS97] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw Hill, International Edition, 1997.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. ACM Transactions on Database Systems, March 1991.

Appendix A

User manual

This provides information for configuring and running the PTAP server. There are two types of options for configuring PTAP - Compile time options and runtime options.

A.1 Compile time options

These option are to be set before building PTAP sources. There are two compile time options that need to be set.

A.1.1 ICM_OPTION

The PTAP server can be installed in two modes - ICM_ENABLED mode and ICM_DISABLED mode. In ICM_DISABLED mode, the call-back functions are not compiled. When an application does not need any integrity constraints to be defined by the call-back functions, ICM_DISABLED mode is to be used. It avoids the overhead of calling the call-back function.
If PTAP is installed with ICM_OPTION set to ICM_ENABLED, then the library `libicmgr.so` that implements the call back functions needs to be provided at runtime, by setting the `LD_LIBRARY_PATH` variable to path of `libicmgr.so`.

A.1.2 BACKEND

The PTAP server uses the `logger` component to log operations in relational database. The current implementation of `logger` supports mySQL and Oracle as backend. Depending on the backend to be used, the BACKEND variable is to be set to either ORCL_BACKEND or MYSQL_BACKEND.

A.2 Runtime options

These options are to be passed to the PTAP server as command line parameters or to be set in the config file. These are read at the time of starting the server.

A.2.1 Command line options

Along with the command line options that LDAP server supports, PTAP also supports following options:

-S	: Status of PTAP server, 1 - indicates enabled and 0 - indicates disabled.
-L	: Data source name of database that is used by the `logger`.
-U	: User name for the logger database.
-P	: Password for the logger database.
-f	: Config file that is used to specify other runtime options.

A.2.2 Options in config file

The config file can contain following commands along with the normal LDAP server commands:

uri	: This is the LDAP url of the back-ldap server. PTAP uses this as the backend LDAP server.
backLdapSrvr	: This is the command to start the back-ldap server. PTAP does not check for validity of back-ldap server command and the consistency between uri and backLdapSrvr. It is responsibility of the administrator to give valid entries. If this option is not specified in the log file, the back-ldap server is not started. In this case the PTAP server serves only the transaction related requests. All the local transactions serve only as parent transactions to REMOTE_TXN_MAPPING transactions. Actual operations are performed by REMOTE_TXN_MAPPING transactions that are passed to the remote servers.
remoteSrvrBindParams	: This entry gives the bind parameters to be used by PTAP to connect to remote server for starting remote transactions. PTAP must know these parameters to be able to connect to a remote server. these are the only servers on which PTAP can start new transactions. The format of this option is: remoteSrvrBindParams <hostName><port no><bind dn><Authentication method>
backLdapLogFile	: This is the file to which all logs and error messages of the back-ldap server are redirected to. If this option is not specified then the default file name is `backLdap.log`.

Figure A.1 shows a sample config file.

```
# ldap database definitions
database              ldap
suffix                "o=sql,c=RU"
rootdn                "cn=root,o=sql,c=RU"
# this is the url of the back-ldap LDAP server
uri                   ldap://localhost:792
binddn                "cn=root,o=sql,c=RU"
bindpw                secret
# This command is used by the PTAP server to start the back_ldap server.
backLdapSrvr          ./servers/slapd/slapd -h ldap://localhost:792
                      -d 5 -f servers/slapd/back-sql/rdbms_depend/oracle/slapdORCL.conf
# remoteSrvrBindParams <hostName><port no><bind dn><Authentication method>
remoteSrvrBindParams localhost 1031 cn=root,o=sql,c=ru LDAP_AUTH_SIMPLE
# This is the file to which all logs and error messages on the back-ldap
# server are redirected to.
backLdapLogFile abc
```

Figure A.1: A sample LDAP config file