enTrans: A System for Flexible Consistency Maintenance in Directory Applications

Anandi Herlekar, Atul Deopujari, Krithi Ramamritham

Dept. of Computer Science and Engg., I.I.T. Bombay
{anandi, atuld, krithi}@cse.iitb.ac.in

Shyamsunder Gopale, Shridhar Shukla

Persistent Systems Pvt. Ltd., Pune
{shyam, shridhar}@persistent.co.in

1 *enTrans*: What and Why?

Directories were designed for data-intensive applications where reads are more frequent than writes and they were primarily meant for the standard white and yellow page applications. *LDAP* (Lightweight Directory Access Protocol) is an open industry standard for accessing directory-based information. Due to its natural way of representing data in hierarchical form, efficient read access, and support for heterogeneous data, *LDAP* is being used for more demanding applications, such as policy enabled networks and identity and access management. While the early *LDAP* applications mostly involved reads, as directory applications mature, the need for updates is increasing and updates typically are spread across data repositories which store user profiles in a decentralized fashion [XNS02]. For example, the information critical to the functioning of networked services is being stored in directories; the data includes DNS data, certificates, AAA services and application configuration data and a part of this information is updatable. These updates need transaction support to maintain integrity of the information stored across multiple data repositories under continuous updates.

Unfortunately, *LDAP* supports only atomic updates at the individual entry level and, today, the applications are forced to take ad-hoc measures to maintain consistency across multiple data repositories. A systematically developed transaction support for consistent updates across multiple data repositories is therefore imperative. The directory applications increasingly involve long duration activities and hence the traditional *OLTP* transaction support is inadequate. Thus the transaction support should support advanced transaction models, as well; or, at least, should have primitives to facilitate implementation of the advanced model with little effort. The success of *LDAP* can, in part, be attributed to its simplicity and ease of use and it is only natural for users to expect the transaction support for *LDAP* to be as simple to use as are *LDAP* primitives.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

With these compelling motivations, we have developed *enTrans*, a highly flexible and customizable transaction support facility that works with any off-the-shelf *LDAP* server. The *enTrans* framework provides advanced transaction support for *LDAP* applications and a mechanism to define and enforce application specific integrity constraints. The rest of this section describes the specific features of *enTrans*.

Predefined Trigger Action Protocol (PTAP)

Predefined Trigger Action Protocol (PTAP) is a customizable advanced transaction support protocol for *LDAP* applications. It works as an active functionality layer on top of *LDAP* and provides basic APIs for flexible transaction support to an *LDAP* user. Figure 1 presents an architectural overview of *PTAP*. *PTAP* includes support for the standard transaction model and some of the advanced transaction models (e.g., Nested Transactions, Sagas and its variants, and Co-Operating Transaction) for applications that are long running. A user can either choose one of the transaction models supported or can use the primitives provided by *PTAP* to implement the transaction model that best suits her application.

The features of *PTAP* include: (a) The transaction support is provided through simple APIs with minimal changes to the standard *LDAP* APIs to maintain the interface user friendly. (b) The APIs cover all the functionality required to implement an advanced transaction model efficiently. (c) The APIs allow transactions to be started on different *LDAP* servers with different dependencies (e.g., commit, abort, group, start-on-commit) between them. These dependencies are automatically maintained, without requiring the user to expend any extra effort. (d) The transaction support is provided as a plug-in component that will work

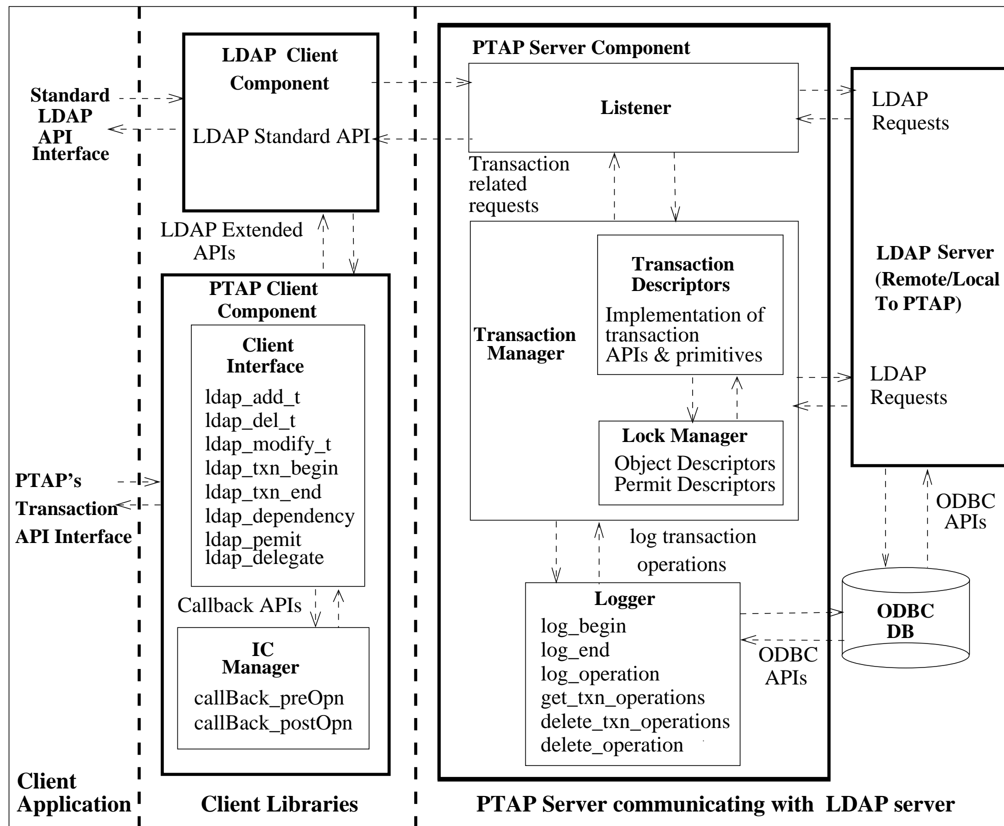

Figure 1: *PTAP* Design Overview

with any *LDAP* server.

Though advanced transaction models have been around for a long time [Elm92, BP95, BDG+94], not many robust system implementations have been deployed, perhaps due to the lack of compelling application scenarios. We believe that with directory based applications becoming more prevalent and their consistency becoming more critical, efficacy of flexible transaction systems such as *enTrans* needs to be showcased to practitioners.

Integrity Constraint Manager (ICM) of PTAP

Integrity Constraint Manager (ICM) of PTAP facilitates defining and enforcing application specific integrity constraints. It allows a user to specify user-defined functions for integrity checking, before and/or after the invocation of an *enTrans* update primitive. Any new *ICM* tailored for a specific application can be plugged-in without adding a new API specific to that application.

enTrans - Integration of *PTAP* with *enQuire*

We have integrated *PTAP* with *enQuire*, Persistent's virtual directory product [Per01]. *enQuire* Virtual Directory is an *LDAP* v3 compliant directory server providing an on-line *LDAP* view built from multiple data stores on the network. With *enQuire*, disparate user databases can be exposed as parts of a single *LDAP* hierarchy. An administrator-supplied configuration that maps data store schema to object classes and attributes is used by *enQuire* server to generate directory hierarchies on-the-fly. *PTAP*'s integration with *enQuire* enables consistent propagation of changes made to directory entries to the data stores – in a manner that conforms with the transaction needs of the application.

2 Motivating Example

In order to motivate the need for advanced transactions in *LDAP*, in this section we present a generic application involving employee information. Consider a typical organization in which the employee information directory is distributed across different departmental data repositories like HR (personal information, address, designation, *etc.*), Accounts (salary, travel allowance, *etc.*) and Security (login, password, *etc.*). The travel office of the organization accesses information about trains, flights and hotel availability from different data repositories. A typical travel plan of an employee can be viewed as a long duration transaction comprising of following activities:

i. Accessing HR repository: The transaction queries the HR data repository to determine the designation dependent travel privileges of the employee.

ii. Booking a ticket for the forward journey: The transaction tries to book an air ticket; if the expenses exceed the travel allowance, either the manager's consent is sought or a train ticket is booked. This involves updates to air and/or railway reservation repositories and to the accounts database.

iii. Booking a hotel room: If the hotel accommodation is not available, the employee either cancels her travel plan and the transaction needs to undo the forward journey reservation or the employee goes ahead with the travel plan (reaching the destination is a must and accommodation and return journey can be managed somehow).

iv. Booking a ticket for the return journey: If return journey ticket is not available, either the employee cancels her travel plan and the transaction has to undo the forward journey reservation and the hotel reservation or the employee goes ahead with the travel plan (return journey can be managed somehow).

The travel plan has different constraints which can be implemented using different transaction models; For example:

(a) Travel only if all the bookings are successfully done: This can be implemented using the Saga model.

(b) Travel even if forward reservation is done but hotel and return reservations are not done. The journey could be done either by air or by train. This can be implemented using a combination of the nested transaction model and the contingency transaction model.

This example illustrates (a) data consistency requirements across multiple repositories and (b) workflow and business logic found commonly in enterprise and service provisioning systems. Such requirements are also becoming increasingly common in the web service infrastructure, as it relies on uniform and consistent access to a user's identity which is split across multiple data repositories. Figure 2 sketches how the above travel plan can be implemented using the primitives provided by *enTrans*. The transaction APIs provided by *PTAP* are shown in bold.

3 *enTrans* Design: Issues and Approach

This section describes some of the design issues from users' point of view and justifies our decisions.

- One way to realize *PTAP* is to implement it as a plug-in library that provides transaction functionality. This needs the *LDAP* server to make calls to the functions in this library for transaction related requests. Other approach is to implement *PTAP* as a server that intercepts the calls to *LDAP* server and serves the transaction related requests; for normal *LDAP* requests, it calls *LDAP* functions. As the later solution does not need any changes to the backend *LDAP* server and can work with any *LDAP* server, we have implemented *PTAP* as a server.

- The callback functions implemented by the *ICM* component of the *PTAP* make it easy for the end user to implement complex operations like those in the travel plan in a single *LDAP* update call (like **ldap_add_t/ldap_delete_t**) and the transaction properties of the updates are ensured by the callback functions of the *ICM*. This reduces efforts of the end user in writing her own transaction model.

- One way to enable a user to start dependent transactions across multiple *LDAP* servers is to allow her to establish two different connections with two different *LDAP* servers, start the transactions on these servers and then form dependencies among them. But this requires a lot of work from the user's part. Also, since the user can start transactions in any order on any server, all transactions involved in the dependency need to be validated. The validity information gathered may not be consistent at a given point due to network delays in the distributed transactions and distributed checking. We avoid this validation by adding the host and port as parameters to **ldap_txn_begin** API and making a restriction that all the dependent transactions should be initiated through a single *LDAP* server to which the client connects for the first time.

PTAP has server and client side components. The server side component of *PTAP* implements the transaction APIs. All the requests to *LDAP* server go through the *PTAP* server component and it consists of *Transaction Manager (TM)* that uses *Logger* functions to log the operations executed in a transaction. The logs are used to bring the directory in a consistent state by undoing appropriate transactions after a failure.

The client side component of the *PTAP* provides APIs for transaction initiation and termination and for specifying dependencies among them. It also includes the *ICM* that defines the callback functions; the *ICM* is a plug-in component with the default implementation returning *LDAP_SUCCESS* for each callback function.

```
travel_plan = ldap_txn_begin(0) // travel_plan is ID of type LDAP_TXNID for a new transaction
    forward_res = ldap_txn_begin(travel_plan) // forward_res executes as a sub-transaction of travel_plan
    ldap_formDependency(travel_plan, forward_res, GC)
    // group commit (GC) dependency is formed between the two that
    // aborts both of the transactions even if one of them fails
    ldap_permit(travel_plan, forward_res)
    // travel_plan permits forward_res to perform conflicting operations on objects that are locked by it
        ldap_search(travel_allowance)
        // search the accounts directory for the travel_allowance of the employee
        if(expensesForAirRes < travel_allowance)
            ldap_add_t(forward_res, flightDetails)
            // This adds an entry with details of the flight reservation to the directory that stores
            // the reservation information. This is done as a part of the forward_res transaction
            ldap_modify_t(forward_res, expDetails)
            // modify the account information of the employee in the accounts directory as part of forward_res
        else
            ldap_add_t(forward_res, trainDetails)
            // This adds an entry with details of the train reservation to the directory entry that stores
            // the reservation information. This is done as a part of the forward_res transaction
            ldap_modify_t(forward_res, expDetails)
    ldap_delegate(forward_res, travel_plan)
        // forward_res delegates to travel_plan the responsibility for operations performed by it
    ldap_txn_end(forward_res)
    hotel_res = ldap_txn_begin(travel_plan)
    ldap_formDependency(travel_plan, hotel_res, AD)
    // forms an Abort Dependency (AD) with travel_plan that aborts hotel_res if travel_plan fails
        ldap_add_t(hotel_res, hotelDetails)
        // This adds an entry with details of the hotel booking
        // This is done as a part of the hotel_res transaction
        ldap_modify_t(hotel_res, expDetails)
    ldap_txn_end(hotel_res)
    return_res = ldap_txn_begin(travel_plan)
    ldap_formDependency(travel_plan, return_res, AD)
        // the AD aborts return_res if travel_plan fails
        // similar code as the one for forward_res
    ldap_txn_end(return_res)
ldap_txn_end(travel_plan)
```

Figure 2: Implementation of travel plan using primitives provided by *enTrans*

4 Summary

The support provided by *enTrans* can be employed by users to realize long running activities with transaction properties – where the activities access one or more *LDAP* servers. The philosophy underlying *enTrans* allows us to use any standard *LDAP* server. It also provides for a pluggable and hence customizable integrity constraint manager. A detailed description of *PTAP* implementation can be found in [HDR02].

References

[BDG+94] A. Biliris, S. Dar, N. Gehani, H. Jagadish, and K. Ramamritham. ASSET: A system for supporting extended transactions. In *Proc. of the SIGMOD*, pages 44–54, 1994.

[BP95] Roger S. Barga and Calton Pu. A practical and modular implementation of extended transaction models. In *Proc. of the VLDB*, pages 206–217, 1995.

[Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[HDR02] Anandi Herlekar, Atul Deopujari, and K. Ramamritham. Advanced transactions in LDAP. Technical report, Indian Institute of Technology, Bombay, Janaury 2002. Available at www.cse.iitb.ac.in/krithi/papers/enTrans.ps.gz.

[Per01] Persistent Systems Private Limited. *Enquire Directory Server*, 2001. Available at http://www.persistentdata.com/documents/EnquireWhitePaper.pdf.

[XNS02] XNS Public Trust Organization. *An Introduction to XNS, the eXtensible Name Service*, January 2002.