Using Transactions in the EAN X.500 Directory Service

Barry Brachman
Gerald Neufeld
Dept. of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z2

June 8, 1993

brachman@cs.ubc.ca

1 Introduction

X.500[1] is a set of ISO and CCITT recommendations for an OSI (Open Systems Interconnection) distributed directory service. The directory manages a distributed information database containing all the objects to be named. It also defines a hierarchical relationship between the objects.

The directory database is partitioned among a set of directory system agents. The collection of agents provides the directory service. The directory service incorporates distributed algorithms for name resolution and search, resulting in a network transparent service. A user agent is used to access the directory service.

The EAN X.500 implementation [3] comprises a DSA, several DUAs, an ASN.1 compiler, and assorted support tools. All applications are built using the UBC OSI Distributed Application Programming Environment (DAPE) [2].

This paper looks at the design of a remote application programming interface (API) that allows user agents to be built outside of DAPE and yet still interact with an EAN X.500 DSA. Nonstandard extensions to the EAN X.500 DSA that provide user agents with a means of handling atomic transactions are also described.

The paper is structured as follows: Section 2 provides a brief overview of the extentions added to the EAN X.500 DSA.

2 Overview

The following nonstandard transaction operations have been added to the EAN X.500 DSA:

- Start Transaction
- Commit Transaction
- Abort Transaction
- Precommit Transaction
- List Precommitted Transactions

Use of the nonstandard operations is optional; standardized operations will behave in the standardized manner. That is, if transactions are not explicitly used, each operation behaves as if inside of its own top-level transaction. The EAN X.500 Directory Access Protocol (DAP) API has been enhanced so that these new operations can be invoked from a DUA. Each of the standard X.500 operations can be invoked within the scope of an atomic transaction.

Because some applications do not wish to operate within the UBC Threads environment, a remote DAP API facility (*rapi*, rhymes with "happy") has been developed. For example, these applications may wish to use their own threads package. Rapi acts as an intermediary between an application that wishes to use the DAP API and a given DSA.

3 Rapi

The Remote API is a separate UNIX process which receives RPC calls (ASN.1 encoded) from an application via TCP/IP and forwards the request via DAP to a DSA. In addition to all standardized operations, it also supports the EAN X.500 maintenance operations and the transactional operations. Rapi supports multiple concurrent clients and multiple outstanding operations per client. The application, rapi, and the DSA can be on any machines that can establish the required connections.

An application wishing to use rapi is used and linked with the EAN DUA/DAP library in the same manner as an application not using Rapi. The application using Rapi, however, is linked with a library of stubs rather than the usual X.500 Remote Operations (ROSE).

4 Transaction Support

One difference between an application integrated with the DAP API and one using rapi is that the latter must use a new primitive, `Ds_remote()`, to indicate where rapi is running. This function must be called before the API is used. When `X500_Bind()` is called, the information specified by the most recent call to `Ds_remote()` is used to contact a Rapi server. `Ds_remote()` can be called multiple times so that different instances of the rapi server can be used sequentially.

```
int
Ds_remote(char *remoteAPIHost, char *remoteAPIPort)
```

The transactional operations are described below. Familiarity with the EAN X.500 DAP/API is assumed. (Recall that the result of a Bind operation establishes an *association* between the application and the DSA.)

4.1 Starting or switching transactions

```
DsRc
Ds_start(DsConnection *conn, OCTS *xid)
```

The DSA assumes that each transaction is known by a globally unique transaction id (the *xid*). It is up to the application to guarantee this property. If the OCTET STRING `xid` is known to the DSA, it becomes the current transaction for the association, otherwise a new transaction is initiated and made the current transaction (if there was a previously active transaction it continues to exist). Transactions cannot be nested.

```
DsRc
Ds_abort(DsConnection *conn, OCTS *xid)
```

Abort the given transaction.

```
DsRc
Ds_commit(DsConnection *conn, OCTS *xid)
```

Commit the given transaction.

```
DsRc
Ds_precommit(DsConnection *conn, OCTS *xid)
```

Precommit the given transaction. The only subsequent operations that are legal for `xid` are abort and commit.

```
DsRc
Ds_listprecommitted(DsConnection *conn, LIST **result)
```

The `xids` of *all* precommitted transactions known to the DSA are returned as a list of OCTET STRING.

5 Caching

The stubs layer that replaces the X.500 ROSE layer uses caching to improve application performance. Caching is on a per-association basis only. The result of a Read operation is cached if the full entry is being read. The cached result can be an entry (a successful read) or an error (an unsuccessful read). Results are cached whether they are part of a transaction or not.

The DSA has been extended to facilitate caching in that all local entries in the DSA are given unique "birthmarks" (version identifiers). Whenever an entry is created or modified, it is assigned a new birthmark. An entry that the DSA has cached as a result of a chained Read or Search does not have a birthmark and cannot be cached. Outside of a transaction, an application may read stale cached entries. Within a transaction, however, the system guarantees at precommit/commit time that the transaction used the "current" version of the cached entry. This is achieved by transmitting

to the DSA the distinguished name and birthmark of each entry read from the cache during the transaction. The DSA validates these entries and responds with a (possibly empty) set of entries that are stale. Stale entries are removed from the cache. If one or more stale entries were read by the transaction, it is unilaterally aborted by the DSA.

An association's cache is freed when the association is terminated. Also, an entry is removed from the cache if it is the object of a Modify, ModifyRDN, or Remove operation.

6 Trapi - A DUA to test rapi

Trapi is an example of a simple DUA that tests rapi and the transactional aspects of the EAN DSA. Besides helping to debug rapi, it is currently the only DUA that supports the transactional operations. It should be relatively easy to understand how trapi works and to extend it. While rapi is intended to be a complete and sturdy server, trapi is a simple and internally crude application that provides a minimal user interface. For example, it is not possible to set service controls.

The supported commands are listed in Figure 1. They can be abbreviated to the shortest unambiguous string. Optional arguments are designated within square brackets.

Command Name	Function
`?`	Display all command names
`abort [xid]`	Abort the current transaction or the given one
`add [<dn>]`	Add a randomly constructed entry
`bind [<dsa> <dua>]`	Connect to a DSA
`cd <dn>`	Set the current DN
`close`	Close the current connection
`commit [xid]`	Commit a transaction
`conn`	Print info about open connections
`help`	Print a list of commands
`listprecommitted`	Print a list of all precommitted xids
`precommit [xid]`	Precommit a transaction
`pwd`	Print the current DN
`quit`	Exit the program ungracefully
`rapi [rapi_host rapi_port]`	Set or see location of rapi
`read [<dn>]`	Read an entry
`start [xid]`	Start or resume a transaction
`switch <conn#>`	Switch active connections

Figure 1: Trapi Commands

Notes:

- The prompt indicates the current connection id.
- Single or double quotes can be used if a token contains white space.

- The `add` and `read` commands append their argument, if present, to the current DN (Distinguished Name). The current DN, initially null (i.e., the root), is set using the `cd` command. No braces are put around the optional argument or around the argument to the `cd` command.
- Error messages are incredibly brief. Consult the DSA log files for some additional clues about what went wrong.
- The `add` command adds a random entry of type `eanperson` under the given DN.
- The `bind` command defaults to the DSA given in the x500.config file. It prompts for a password. Doing a `bind` with no arguments and no password provides a read-only, anonymous connection. In the UBC environment, doing `bind stephen {cou=ca,cn=admin}` with a password of "grumpy" provides a read/write connection.

References

[1] Comite Consultatif Internationale de Telegraphique et Telephonique (CCITT), Fascicle VIII.8, "Recommendation X.500: The Directory – Overview of Concepts, Models and Services", Dec. 1988.

[2] G. Neufeld, M. Goldberg, and B. Brachman. "The UBC OSI Distributed Application Programming Environment – User Manual", Technical Report 90-37, Department of Computer Science, University of British Columbia, Jan. 1991.

[3] G. Neufeld, B. Brachman, M. Goldberg, and D. Stickings. "The EAN X.500 Directory Service", *Journal of Internetworking Research and Experience*, Vol. 3, No. 2, (June 1992), pp. 55-82. Also Technical Report 91-29, Department of Computer Science, University of British Columbia, November, 1991.