Support for Transactions and Replication in the EAN Directory Service[†]

Gerald Neufeld and Barry Brachman

Dept. of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z4

Abstract

The OSI directory system manages a distributed directory information database of named objects, defining a hierarchical relationship between the objects. An object consists of a set of attributes as determined by the object's class. Attributes are tuples that include a type and one or more values.

After presenting an overview of the X.500 standard, we discuss the use of atomic transactions within the EAN X.500 implementation. Atomic transactions are used within the database component of the Directory Service Agent (DSA). Non-standard extensions to X.500 were made to provide user agents with a means of forming atomic transactions. The new interface allows any sequence of standardized X.500 operations to be executed in the context of an atomic transaction. Support is also provided for the two phase commit protocol, allowing two or more directories to atomically commit updates. This work was done in conjunction with a distributed multidatabase (MDBS) project which used the X.500 directory to store its schema information [5]. Finally, ongoing work to provide replicated master DSAs is described.

Keyword Codes: C.2.4; H.3.4

Keywords: Distributed Systems; Information Storage and Retrieval, Systems and Software; Directory Services

1. Introduction

X.500[6] 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

[†]This work was partially supported by a grant from NSERC.

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.

This paper describes the use of atomic transactions within the EAN X.500 directory and extensions to the X.500 standard to provide user agents with a means of handling atomic transactions. The new interface allows any sequence of standardized X.500 operations to be executed in the context of a transaction. Support is also provided for the two phase commit protocol, allowing two or more directories to atomically commit updates. This work was done in conjunction with a distributed multidatabase project which used the X.500 directory to store its schema information [5]. Replication, based on the two phase commit protocol, has also been implemented

The paper is structured as follows: Section 2 provides a brief overview of the ISO/CCITT X.500 directory services, Section 3 discusses the use of atomic transactions within our DSA, and Section 4 describes non-standard additions to the EAN X.500 application program interface. Replication is the subject of Section 5 and Section 6 is the conclusion.

2. The X.500 Directory Model

The X.500 directory model consists of a set of active agents called DSAs (Directory System Agents), a directory information database that is distributed among the DSAs and is called the DIB (Directory Information Base), and a set of DUAs (Directory User Agents) through which the DSAs containing the DIB are accessed. The DSAs communicate with each other using the Directory System Protocol (DSP); a DUA communicates with a DSA using the Directory Access Protocol (DAP).

The DIB is organized using a hierarchical, tree-structured object model. The DIB is usually referred to as the Directory Information Tree (DIT). In the DIT, each node or entry represents an OSI object, such as a country, organization, person, machine, document, or application. Each entry belongs to a particular class and consists of a set of attributes as defined by its class. An attribute is composed of a type and one or more values. For example, class Person can have a set of mandatory attributes, such as the person's surname, as well as a set of optional attributes, such as the person's telephone number, e-mail address, and photo (a Group 3 fax image).

Each attribute type is uniquely identified by a data structure called an *object identifier*. Object identifiers are internationally standardized or defined by national administrative authorities or private organizations. The syntax of an attribute value is determined by the attribute type. The syntax indicates how the value is represented and how it is compared. For example, the syntax of the UTCTime[2] attribute follows the ASN.1 definition of UTCTime and matches for equality if two values represent the same time. Two UTCTime attributes can also be matched for order; that is, an earlier time is considered less than a later time. Another example is the telephone attribute, which matches for equality but not order.

Within an entry, one or more attribute values are designated as distinguished. The

[2]A representation of Coordinated Universal Time (Greenwich mean time).

set of such attributes and their distinguished values form a Relative Distinguished Name (RDN). The RDN must be unique among the entry's siblings. For an entry, therefore, an RDN
uniquely identifies an immediate descendant of that entry.

Although the DIT is a tree, alternative names can be used for the same object by using aliases. An alias entry points to an object entry by storing the distinguished name of the object in the entry of the alias. As such, it is a symbolic link to the object entry. Figure 1 illustrates these concepts.

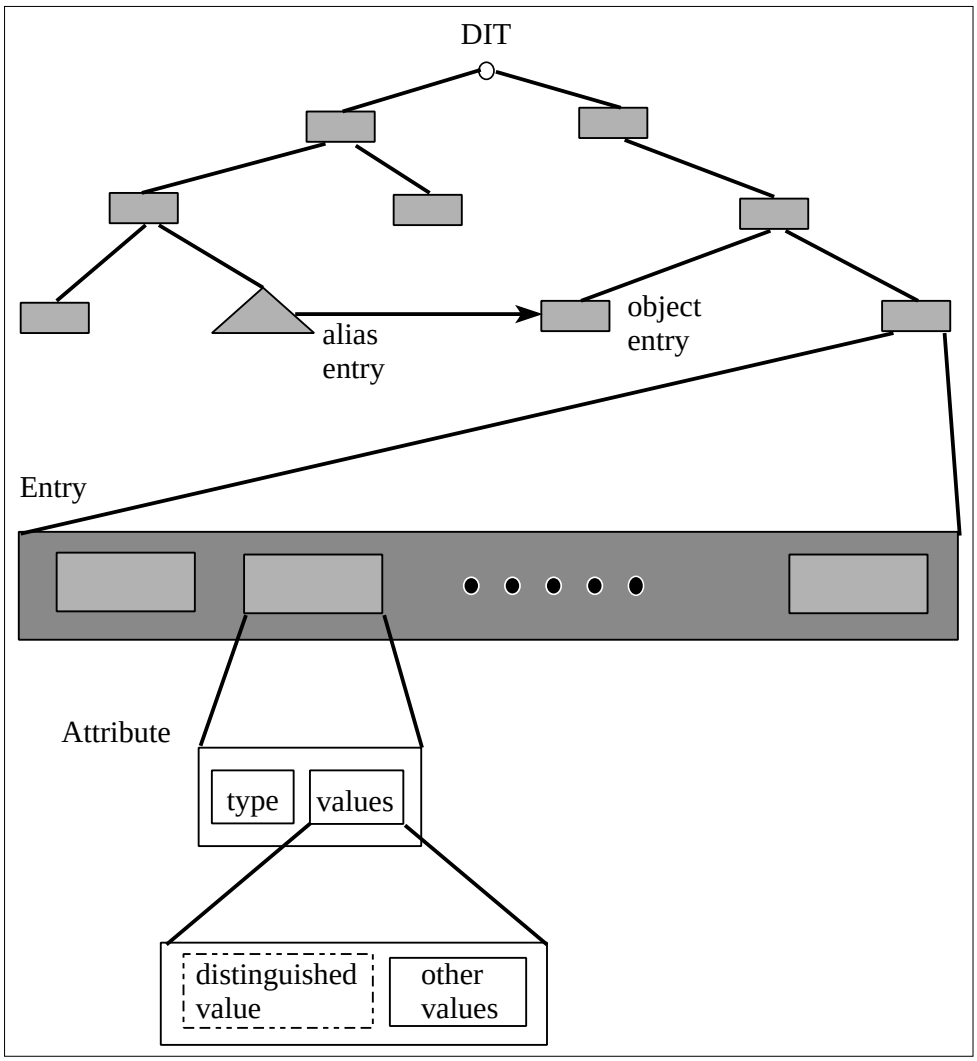

Figure 1. Structure of the DIT

A *distinguished name* for an object is the sequence of RDNs from the root of the DIT to

the object. When a user presents a possible distinguished name to the directory system, the system determines whether the purported name is valid. The purported name is presented as a sequence of RDNs, where each RDN consists of a set of Attribute Value Assertions (AVAs). An AVA is a possible attribute type and value that is purported to be a distinguished value. It is the function of the name resolution algorithm to validate the purported distinguished name and, if successful, to locate the entry with that name.

When the DIB is distributed, each DSA typically holds one or more *naming contexts*, which are fragments of the DIB. The distinguished name of the initial vertex of a naming context is the *context prefix*. For directory requests to be performed independently of the origin of the request, DSAs must be able to identify and interact with each other. A DSA accomplishes this by maintaining several kinds of knowledge references about other DSAs. The access point (presentation address) of a DSA responsible for the part of the DIT immediately below a particular entry is represented by a *subordinate reference*. Similarly, a *superior reference* represents the access point of a DSA immediately above a particular entry. A *cross reference* is a type of knowledge that improves name resolution by associating a context prefix with an access point.

Figure 2 shows a hypothetical example of a DIT. In this example there are two country objects below the root, representing Canada and Great Britain. Under each country object is an organization object. Under the Canada object is a university organization and under the Great Britain object is a business organization. Within organizations are one or more organizational units. For example, below UBC, is the Science faculty and within Science is the Computer Science department, which is CS. The leaves represent several physical objects.

If it is assumed that the attribute beside each entry in Figure 2 is the RDN for that object, valid distinguished names can be formed for the objects. For instance, the name {`C=CA, Org=UBC, OU=Science, OU=CS, CN=Peter Smith`} identifies the person Peter Smith who works in the Computer Science department at UBC.[3] The fax machine in Sales at XYZ has the name {`C=GB, Org=XYZ,`
`OU=Sales, CN=fax`}. This object can contain the fax telephone number for the physical object. There is no ambiguity because the order of the attributes in the distinguished name is significant. Distinguished names in X.500 are not very different from names in a hierarchical file system, or in other naming systems such as the Domain Name System [13] (DNS) or Clearinghouse [15]. In X.500, the DIT can be arbitrarily deep and there is only one root for all objects.

The directory operations defined by the X.500 recommendations are listed in Figure 3. With the exception of Abandon, each operation takes a distinguished name among its arguments. An assortment of service controls are available for the user to direct or constrain operations. For example, a limit can be placed on the number of entries returned by List or Search, or the use of cached information can be forbidden.

Multiple independent directories are possible. Each directory has a completely separate name space and can have a DIB distributed among several DSAs.

[3]Attribute types can be abbreviated: for example, "`C`" for `Country`, "`Org`" for `Organization`, "`OU`" or "`OrgUnit`" for `Organizational Unit`, "`CN`" for `Common Name`.

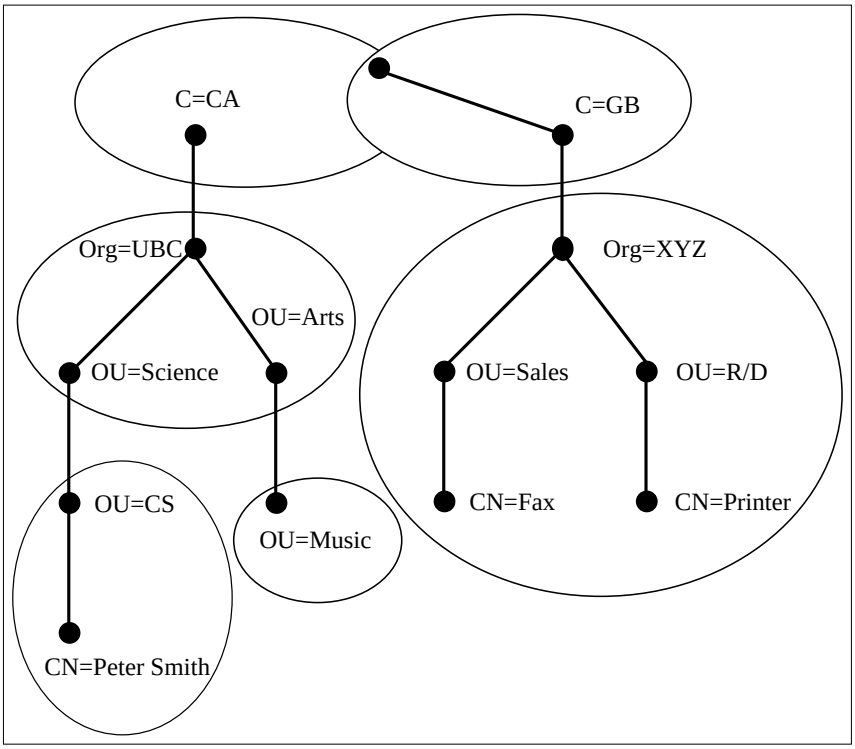

Figure 2. Example of a DIT

Operation Name	Function
Abandon	Cancel an active Read, Compare, List, or Search operation.
AddEntry	Add a leaf entry to the DIT.
Compare	Compare a value with the values of a particular attribute type in a particular entry.
List	List the immediate subordinates of an entry.
ModifyEntry	Perform a sequence of one or more modifications to an entry.
ModifyRDN	Change the RDN of a leaf entry.
Read	Extract information from an entry.
RemoveEntry	Remove a leaf entry from the DIT.
Search	Search a portion of the DIT, starting from a particular entry and returning selected information from entries of interest.

Figure 3. Directory Operations

3. Atomic Transactions for Fault Tolerance

The X.500 Recommendations specify how the Directory functions, but do not dictate how it is to be implemented. When implementing the Directory, the greatest challenge is achieving a high degree of efficiency while conforming to the standard. Towards that end, the EAN X.500 implementation [11] uses inverted indexes to reduce search time and stores some directory information redundantly to speed browsing. Also, the DSA was implemented as a multithreaded application, supporting concurrent requests from DUAs and DSAs.

As a consequence of choosing to use more complicated data structures in exchange for improved performance, a single X.500 operation that updates the directory, such as AddEntry, RemoveEntry, or ModifyEntry, ultimately causes many data structures to be modified on disk. Atomic transactions are used in the EAN X.500 DSA so that these data structures are updated in an all-or-nothing fashion [9]. In the event of a fault, such as a DUA or DSA crash or a communication failure, the data structures are never left in an inconsistent state. Also, since the DSA allows concurrent requests (such as multiple DUAs attempting updates simultaneously), atomic transactions ensure that the operations appear to be performed in some serial order (i.e., each operation appears to be executed in isolation) and the database remains consistent.

The EAN X.500 DSA's support for atomic transactions is realized by two components [11]: the persistent object store and the database. Both components use the Threads light-weight process kernel [10]. They are discussed separately in the following sections.

3.1. Persistent Object Store

Many OSI applications require some kind of persistent store. For X.500, each DSA must store its portion of the DIB. To accommodate these applications, a persistent object store was created. This generalized facility can be used for many different applications, not just X.500.

The object store provides nested atomic transactions [8]; most object store operations are executed in the context of a transaction. The object store takes an optimistic approach to concurrency control. This approach is appropriate in the context of the directory service, because update transactions are relatively infrequent and concurrent updates are unlikely. If the system crashes before a top-level transaction commits, the transaction is implicitly aborted. Likewise, attempting to commit a transaction that would leave the object store in an inconsistent state (for example, multiple simultaneous updates) causes the transaction to be aborted. If the system crashes during a commit, the system completes the commit when the application is restarted. Transactions also simplify exception handling because an explicit abort undoes a partially completed request that may involve many objects. This mechanism provides a simple and easily used facility to create fault-tolerant applications.

The `BeginTransaction` operation causes a new object store transaction to be started and associated with the requesting DUA. All subsequent operations are then executed within the context of the established transaction. The `CommitTransaction`, `PrecommitTransaction`, and `AbortTransaction` operations are likewise mapped into the corresponding object store functions.

3.2. Database

To provide consistency in the face of crashes, assistance for implementing distributed and replicated databases, and multithreaded operation, the `tdbm` [3] database (`dbm`[7] with transactions) was developed. The object store uses `tdbm` as its underlying database. The `tdbm` database provides nested atomic transactions [14], volatile and persistent databases, support for very large data, storage for the database within a single UNIX[4] file, and assistance for managing distributed databases. It can be configured to operate either as a conventional UNIX library or as part of a multi-threaded application.

The `tdbm` database uses an extensible hashing technique than can retrieve the page within the database file holding the item of interest in one or two disk operations as the database grows and shrinks. It uses a lock manager, provided by Threads, to allocate, obtain, and release a lock on behalf of a client thread. With strict two phase locking [14], locks are not released until the top-level transaction commits or an abort occurs.

Commit processing of a top-level `tdbm` transaction is done by creating a transaction file, which is an *intention list* [16] that represents the actions that must be executed to update the database. This approach is *after-image physical logging* [9].

Recovery is automatically initiated when `tdbm` is started so that incomplete transaction files can be removed and the contents of completed transaction files can be applied (or reapplied) to the database. The recovery procedure is idempotent: if the system crashes during the overwriting process, recovery can be retried until successful.

One of the original design goals of `tdbm` called for databases that could be used with an object store that supports distributed operations and replication. The distributed object store is responsible for interprocess communication and execution of an atomic commit protocol (such as the two phase commit protocol [2]), but the underlying databases must provide some assistance. The `tdbm` database does this by providing a precommit (prepare to commit) operation and a way of determining at restart time whether a distributed transaction was in progress, and if so, the databases involved and the phase the transaction was in.

4. Enhancements to X.500

There are several reasons for wanting to provide atomic transactions to DUAs. A user may simply want to add or modify two or more entries in an all-or-nothing fashion. There are typically fixed-costs associated with beginning and committing a transaction, making it more efficient to perform several updates within a single transaction rather than doing each within its own transaction. This is important, for example, when a directory is initially loaded with entries. Atomic transactions also support updates that involve two or more directories. In this situation, a protocol such as the two phase commit is used so that updates are made to all directories in the transaction or no updates are applied to any directory. A multidatabase project [1] is using transactions for this purpose. The standardized X.500 DAP does not include atomic transaction operations; we have provided new operations through our API.

Communication between a DUA and a DSA in the enhanced system can use either the standardized DAP interface or an interface based on the Open Software Foundation's

[4] UNIX is a trademark of AT&T.

Distributed Computing Environment (DCE), which provides communication and resource sharing services, including RPC. The DUA programmer is provided with an Application Program Interface (API) that communicates using the DAP
or the specialized DCE/RPC based protocol.

In the remainder of this section, an extended model of DUA-DSA interaction, details of the modifications made to the DAP, and an overview of the DSA's support for transactions are presented.

4.1. DUA-DSA Interaction

A transaction executed at a DSA may involve only that DSA or be part of a larger transaction involving several DSAs. For one DSA, the DUA can request any number of operations within the context of an atomic transaction at the DSA. Figure 4 shows three DUA commands executed within an atomic transaction. To the API must be added new operations to begin, commit, and abort a transaction associated with a particular connection to a DSA. Communication can use the DAP or DCE/RPC.

For DCE/RPC, a client executes transactions involving at least two DSAs (or other kinds of servers) and requires that either all DSAs commit their transactions or that none do. The *proxy*, a new component of the DSA, accepts DCE/RPC requests and submits them for execution within the DSA. An example of this mode of interaction is presented in Figure 5, where two DSAs are involved in a transaction; the higher-level transaction is committed only if both DSA_A and DSA_B successfully precommit (prepare to commit) each of their transactions and, therefore, vote yes as part of the two phase commit protocol. The DSAs never act as transaction managers. An operation to precommit must be added to the API for the two phase commit protocol.

In neither situation can a transaction require chaining from one DSA to another, so a DUA must execute a transaction at the DSA responsible for the distinguished name of interest. The DUA must contact the DSA directly. A DSA can return a referral to the DUA, which is a pointer to another DSA at which name resolution can be continued. The client must, therefore, use referrals to reach the appropriate DSA. As it is doing this, however, each one of the DSAs that is returning a referral is becoming part of the transaction. Aside from ignoring the problem, a subtransaction can be used for each access to a DSA. The subtransaction can be aborted if a referral is constructed. This preferable solution requires the proxy to generate the abort and forward the request. Also, the DSA that has the entry should return its access point (address) so that the DUA can cache this knowledge and go directly to it next time, improving performance. Changes to the API may be required to return the access point. Because there is no DSA to DSA communication, there is no requirement for a DSA to handle knowledge information other than that provided via a DUA.

One example of a transaction manager that could be interfaced when DCE/RPC is used is Encina[5] [17]. Encina expands the DCE foundation to include services that support distributed transaction processing and management of recoverable data. The Encina Toolkit supports the development of client/server transaction processing applications. Its Distributed Transaction Service component provides the logic for the two phase commit protocol. The Communication Service modules of the Encina Toolkit provide mechanisms

[5]Encina is a trademark of Transarc Corporation.

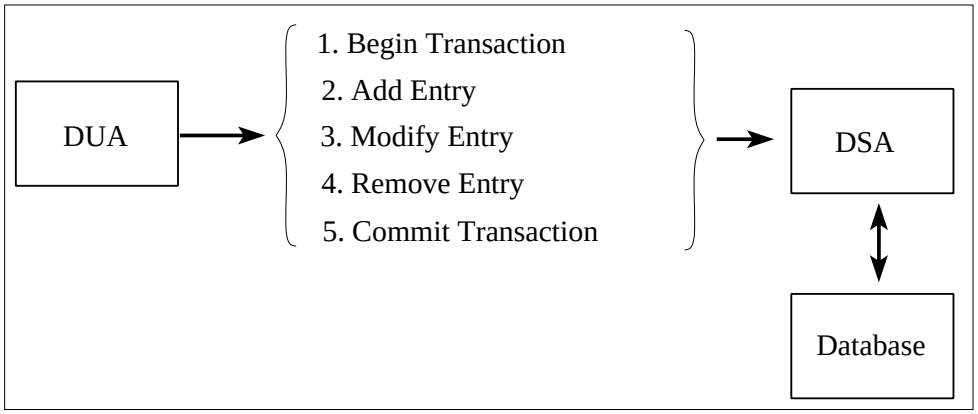

Figure 4. Transaction with a Single DSA

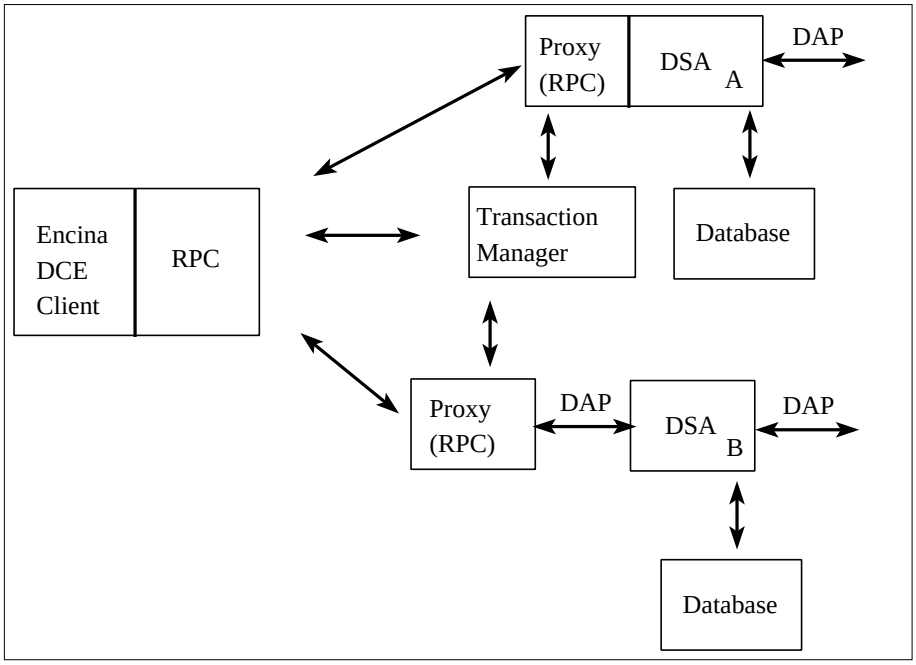

Figure 5. Transaction with Multiple DSAs

(such as RPC) for an application to make requests of other application programs.

In the example in Figure 5, a client uses the Encina transaction service. The Encina transaction service's primary responsibility is to conduct the two phase commit protocol. Both the application and the transaction service communicate with the proxy using Encina's RPC support. The proxy effectively converts a request received via RPC into the corresponding DAP API operation for execution.

4.2. Additions to the DAP

The new operations that have been added to the API to support atomic transactions are:

- Starting a new transaction
- Precommitting a transaction
- Committing a transaction
- Aborting a transaction

A traditional directory operation can be performed outside of a transaction, where the operation behaves as if it were enclosed by a `BeginTransaction`, `CommitTransaction` sequence.

Each operation takes an arbitrary character string as an argument. The string is the user agent's identifier for a transaction. The new operations do not return a result, but rather an indication of whether the operation was successful or unsuccessful. The new operations are described in more detail in [12].

A user agent starts a new transaction at a DSA by invoking `BeginTransaction`. These transactions can be nested. The first argument identifies the connection to the DSA.

The current transaction can be precommitted using `PrecommitTransaction`. If this operation is successful, the directory guarantees that a subsequent `CommitTransaction` operation also succeeds. After being precommitted, the transaction can only be committed or aborted; no other operation can be performed. This operation can be used in the first phase of the two phase commit protocol. If all directories in the distributed transaction precommit successfully, the second phase issues a `CommitTransaction` at each directory. Because a DSA acts as a server, the application or its transaction manager must track the global state of the two phase commit protocol and must reestablish contact with a DSA in the event of a crash or communication failure.

The active transaction is committed using
`CommitTransaction`. The transaction may have been previously precommitted.

The `AbortTransaction` operation aborts the current transaction. The same effect is implicitly achieved by the `Abandon` operation or if the connection between the DUA and DSA is lost while a transaction is in progress.

4.3. Multidatabase Applications

An MDBS provides an integrated view of data from multiple, autonomous, heterogeneous, distributed sources, sparing applications from having to deal with many different interfaces and protocols for retrieving and updating data. The catalog component of the CORDS MDBS [5] is required to maintain information on schemas, security, component

data sources, host sites, network links, and system performance. The prototype of the CORDS MDBS uses the EAN Directory Service to manage the catalog. An MDBS transaction management architecture has been developed and work has begun on an X/Open XA interface for the EAN Directory Service and configuration of the directory as an XA-compliant resource manager using the Encina Transaction Processing monitor. This architecture uses the transaction management enhancements discussed above.

5. Replication

Replication can provide increased performance and availability. Performance is improved when data "closest" to the requestor can be accessed. Without replication, if the sole copy of the data is inaccessible, the system will be unusable. Better availability is achieved when alternate copies of data can still be accessed after failures.

Our experience with X.500 suggests that the vast majority of directory operations are read only. Replication can avoid having to use the network for these operations. Also, it means that the probability of conflicting transactions is low.

The 1992 version of the X.500 Recommendations provides replication through a primary/shadow arrangement. Updates are applied at the primary DSA and disseminated to shadow DSAs. If the primary DSA is unavailable, directory updates cannot be performed.

Rather than doing replication at the application level, we are implementing replication within `tdbm`. This will result in an application-independent tool and will permit updates to be executed at any replica.

Our approach to replication is to add the two phase commit protocol to `tdbm`. This enforces strong consistency among the replicas, ensuring that all replicas store exactly the same data. We feel this is a good choice in a LAN environment where communication is fast and reliable or in an environment where the number of replicas is relatively small.

Two phase commit works adequately when there are no failures, but updates can not be performed if any replica fails or the network partitions. To ameliorate this problem, we plan to implement the virtual partition algorithm [2] so that normal operation can continue within a majority subset of the replicas.

We have also developed a weak consistency scheme [4] that sacrifices traditional database update semantics in exchange for greater availability than any known strong consistency protocol. The weak consistency method falls into the family of optimistic protocols. After a partitioning, execution of transactions proceeds normally. If write-write conflicts are detected when partitions later merge, transactions may be rolled back to ensure consistency. Since we have observed that directory updates are relatively rare and do not tend to conflict in any case, weak consistency appears to be particularly well suited to use with X.500. We feel, however, that the implementation of weak consistency will be significantly more difficult than that of the virtual partition algorithm.

6. Conclusions

We began using atomic transactions within the EAN X.500 DSA primarily as a means of providing all-or-nothing behaviour when updating the local portion of the DIT. We then extended `tdbm` to provide support for the two phase commit protocol by implementing a precommit function. The X.500 Directory Access Protocol was then enhanced by adding

atomic transactions to the API. The API to the DSA was extended to support requests via DCE/RPC in addition to the standardized DAP. Support was provided so that multiple directories (and other kinds of servers) could be used in a distributed update. Finally, replication functionality was added to `tdbm` so that multiple master DSAs could coexist.

REFERENCES

1. G. Attaluri and D. P. Bradshaw. "Architecture for Transaction Management in the CORDS Multidatabase Service", *Proc. of the 1993 CAS Conference*, October, 1993, pp. 873-887.
2. P. Bernstein, V. Hadzilacos, and N. Goodman. "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
3. B. Brachman and G. Neufeld. "TDBM: A DBM Library with Atomic Transactions", *Proc. USENIX Summer Technical Conference*, June 1992, pp. 63-80.
4. B. Brachman and G. Neufeld. "Weakly Consistent Transactions in ROSS", *Proc. of the 1993 CAS Conference*, October, 1993, pp. 888-894.
5. N. Coburn, P. Larson, P. Martin, and J. Slonim. "CORDS Multidatabase Project: Research and Prototype Overview", *Proc. of the 1993 CAS Conference*, October, 1993, pp. 767-778.
6. Comite Consultatif Internationale de Telegraphique et Telephonique (CCITT), Fascicle VIII.8, "Recommendation X.500: The Directory – Overview of Concepts, Models and Services", Dec. 1988.
7. Computer Systems Research Group,
   Computer Science Division, EECS. **ndbm(3)**, *4.3BSD Unix Programmer's Reference Manual (PRM)*, University of California, Berkeley, Apr. 1986.
8. R. Gruber. "Optimistic Concurrency Control for Nested Distributed Transactions", MIT/LCS/TR-453, June 1989.
9. T. Haerder and A. Reuter. "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, Vol. 15, No. 4, (Dec. 1983), pp. 287-317.
10. 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.
11. 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.
12. G. Neufeld and B. Brachman. "A Transactional API for the EAN X.500 Directory Service", *Proc. of the 1992 CAS Conference*, November, 1992, pp. 81-91.
13. P. Mockapetris. "RFC 1035: Domain Names – Implementation and Specification", USC Information Sciences Institute, Nov. 1987.
14. J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.
15. D. Oppen and Y. Dalal. "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", Technical Report OPD-T8103, Xerox Corporation, Palo Alto, CA, Oct. 1981.
16. H. Sturgis, J. Mitchell, and J. Israel. "Issues in the Design and Use of a Distributed

File System", *Operating Systems Review*, Vol. 14, No. 3, (July 1980), pp. 55-69.
17. Transarc Corp. "Encina Product Overview", Document Number TP-00-M235, 1991.