The EAN X.500 Directory Service[†]

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

Duncan Stickings
OSIware Inc.
4370 Dominion Street, Suite 200
Burnaby, B.C. V5G 4L7

March 15, 1994

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 a particular class. Attributes are tuples that include a type and one or more values. The directory database is partitioned among a set of directory system agents. The directory service is provided by a collection of agents and incorporates distributed algorithms for name resolution and search, resulting in a network transparent service. The objects can represent many real-world entities. The service is intended to serve a very large and diverse user community. This paper describes experiences gained in implementing the directory service. It also points out a number of areas in the current OSI directory design that require further work and then describes how the EAN directory system has addressed these difficulties.

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 relationship between the objects.

[†]This work was partially supported by a grant from OSIware, Inc.

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.

The service is intended to serve a very large and diverse user community spanning many countries and organizations. It is also expected to support multiple applications. The kinds of applications supported determines the kinds of objects maintained. An application such as electronic mail (X.400) is an obvious one. Others, such as library systems[29], however, can also be supported. Since the directory supports multiple applications, it is possible to take advantage of the commonality between them. That is, objects in common among applications can be shared.

This paper describes experiences gained during the design and implementation of the X.500 directory service. It also points out a number of areas in the current OSI directory design that are inadequate and hence require further work. We describe how the EAN directory service has provided solutions to some of these problems.

The paper is structured as follows. The next section provides a brief overview of the ISO/CCITT X.500 directory services and concludes with a set of unresolved issues that a production directory service must handle. Section 3 describes the OSI development environment used in the EAN directory service. The process structure of the EAN Directory System Agent is given in Section 4 and is followed by a section on how the directory information tree is represented and implemented, including a discussion on how knowledge information is represented. Section 6 describes the distributed search algorithm used by the EAN directory service. Section 7 discusses three naming methods: distinguished names, descriptive names, and typeless names. Sections 8 and 9 describe how the EAN system has implemented security and directory system agent maintenance operations, respectively. Neither access control nor maintenance have been addressed by the current standard. Some performance figures are presented in Section 10. Finally, we conclude with a summary of the current state of the system.

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 which is distributed among the DSAs, called the DIB (Directory Information Base); and a set of DUAs (Directory User Agents) through which the DSAs containing the DIB are accessed.

The DIB is organized using a hierarchical, tree-structured object model. In fact, the DIB is normally referred to as the Directory Information Tree, or 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, a class "Person" may 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 may 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 set of such attributes and their distinguished values form a Relative Distinguished Name, or RDN. It is required that the RDN be unique among the entry's siblings. Therefore, given some entry, an RDN uniquely identifies an immediate descendant of that entry.

Although the DIT is a tree, it is possible to have alternative names for the same object by using

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

aliases. An alias entry points to an object entry. It does this by storing the distinguished name of the object in the alias' entry. As such, it is a symbolic link to the object entry. Figure 1 illustrates these concepts.

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 must first determine whether the purported name is indeed valid. The purported name is presented as a sequence of RDNs, where each RDN consists of a set of AVAs (Attribute Value Assertions). 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 fragments of the DIB called *naming contexts*. The distinguished name of the initial vertex of a naming context is called the *context prefix*. For directory requests to be performed independently of where the request originates, 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 located immediately beneath 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 gives 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 exists an organization object. Under the Canada object is a university organization and under the Great Britain object is a business organization. Within organizations we can have one or more organizational units. For example, under UBC we have the Science faculty and within Science is the Computer Science department, abbreviated as "CS". The leaves represent several physical objects.

If we assume that each attribute beside the entry in Figure 2 is the RDN for that object, then we can form valid distinguished names for the objects. For instance the name {`C=CA,`

Figure 1: Structure of the DIT

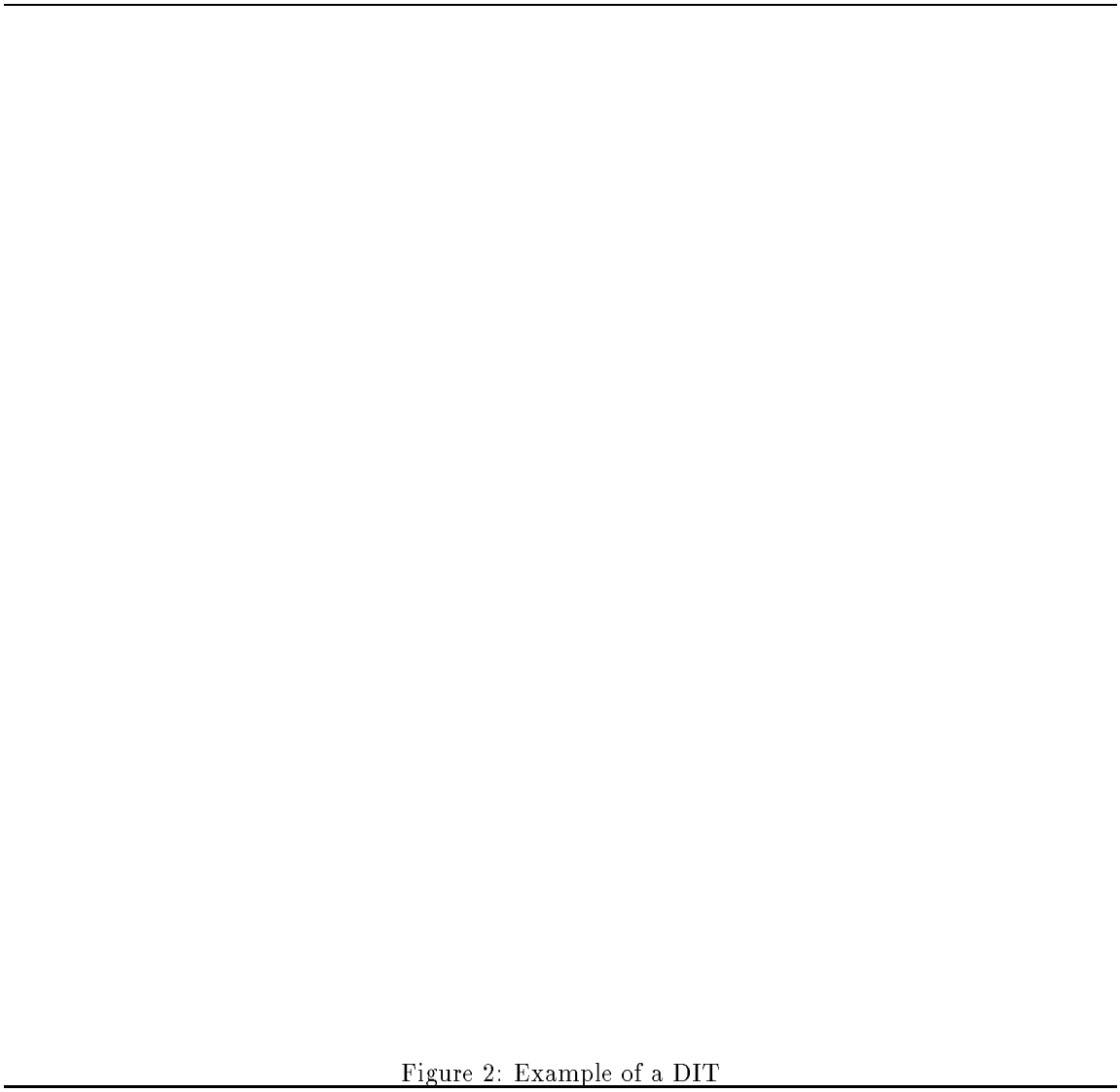

Figure 2: Example of a DIT

`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 would have the name {`C=GB, Org=XYZ, OU=Sales, CN=fax`}. This object would contain the fax telephone number for the physical object. There is no ambiguity since the order of the attributes in the distinguished name is significant. As we have seen, distinguished names in X.500 are not very different from names in a hierarchical file system, or in other naming systems such as DNS (the Domain Name System [15]) or Clearinghouse [22]. A DNS address for Peter Smith would look like `'peter-smith.cs.science.ubc.ca'` and `'peter smith@cs@ubc'` might be a Clearinghouse name for him. In X.500, the DIT may 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, it is possible to place a limit on the number of entries returned by List or Search or forbid the use of cached information.

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

Figure 3: Directory Operations

Distinguished names are considerably more complex than simple hierarchical names such as in DNS. The reason is that an RDN can be a *set* of AVAs, where each AVA includes the type name

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

as well as the value. This complexity often gets in the way of a simple, clean naming structure. It is still an open question whether this complexity provides sufficient functionality to warrant its disadvantages. Some work has been done to alleviate this problem (see Sections 7.2 and 7.3).

There are a considerable number of issues that have not been resolved in the 1988 version of the X.500 directory. Fortunately, many are currently being addressed for the 1992 version. These issues include replication (for both performance and reliability), dynamic schema management, maintenance and network management, access control, and alternative name forms such as descriptive naming. The EAN X.500 implementation provides solutions to several of these concerns. These solutions are presented in the following sections.

3 The EAN OSI Development Environment

The EAN implementation of the X.500 directory is based on the EAN DAPE (Distributed Application Programming Environment) [20]. This is a generalized environment for building OSI applications. The DSA, an interactive terminal-oriented DUA, a non-interactive command-line oriented DUA, and a multi-threaded DUA for the X Window System were built using DAPE.

DAPE is composed of five tools: a light-weight threads kernel, an ASN.1 compiler, a communications service, a persistent object store, and a database. Many of these tools are interrelated and therefore provide a coherent development environment. The tools were built with two major design principles in mind: portability and efficiency. These principles helped guide the design decisions from the many different possibilities. We describe each of these tools in greater detail in the remainder of this section.

3.1 Light-weight threads kernel

To gain the greatest efficiency and portability, it was felt that we should not rely heavily on the operating system's process management and interprocess communication mechanisms. When building interactive OSI servers such as a DSA, however, it is clearly necessary to have multiple concurrent threads of control. The solution that we have adopted is to build our own user-level process kernel,

called Threads. Threads resides within a single host operating system user process; hence, there are no changes required to the host system. This technique is not a new concept and many such sub-kernels have been built.

Using our own threads kernel rather than the host system's is efficient for a variety of reasons. First, thread creation is fast and simple. This is mainly because a thread does not contain a large amount of context. Second, all threads operate in a single, shared address space. This results in efficient interprocess communication since pointers can be passed between threads, reducing the need to copy data. Also, code can be shared by all threads. Third, threads scheduling is non-preemptive, resulting in easier and more efficient concurrency control. Fourth, calls to Threads are efficient since they are simple procedure calls rather than traps. And finally, context switching is inexpensive since the amount of threads state is minimal.

Threads communicate using blocking Send/Receive/Reply primitives [7]. Since these threads share one memory space, no copying of data is done. Concurrency control is provided both through traditional semaphores and a lock manager that supports operations on shared and exclusive locks.

Two kinds of dynamic memory management are provided by Threads: global memory and frame-based memory. Global memory is similar to memory obtained by UNIX's `malloc()` except that requests are rounded up to the nearest "chuck size". For each chunk size, a cache of chunks is maintained so that allocation and deallocation are fast.

In frame-based memory, all memory allocations are associated with a "memory frame". Frames can be linked together to form a stack, one stack per thread. Frames can be created, popped, and pushed; allocations are implicitly associated with the top frame of the requesting thread's stack. The chief advantage of this is that the programmer can explicitly free all allocations associated with a frame in one operation; a thread's frame stack is implicitly freed when it exits. Also, frame memory allocation is typically very fast since it can be obtained from a single global memory chunk simply by advancing a pointer. For the same reason, deallocation of frame memory is also efficient. This memory management scheme is particularly useful when decoding ASN.1 data. Such decoding can perform many memory allocation requests as the internal form of the data structure is constructed.

Often, the internal form together with all of its components can be deleted simultaneously. After a server decodes a request and executes it, for example, the internal form of the request can be discarded.

Portability is facilitated through the sub-kernel technique because instead of making system calls to the host kernel directly, applications must call the sub-kernel's versions. As a result, there is little application code that directly relies on the host operating system. Hence, porting the application basically amounts to porting Threads, a task that can be done once for many different OSI applications. Threads has been ported to several different flavours of UNIX as well as several different machine architectures. Early versions have also been ported to VMS and MS-DOS.

3.2 ASN.1 compiler

A considerable amount of work has been done in implementing ISO BER (Basic Encoding Rules) and ASN.1 [19, 20, 24, 27].

The EAN directory service uses the CASN1 tool [19]. This tool takes as input the ASN.1 description of the APDUs and produces the equivalent C language data structures. It also produces an encoding function and a decoding function, also in C, for each data type. As a result, the programmer can use the data structures directly. To encode an APDU, the programmer can simply call the encoding routine for the "top-level" data element of the APDU. This encoding routine calls other encoding routines for all the other data types in the APDU. When the top-level routine returns, the APDU is returned in BER transfer syntax ready to be passed to the presentation layer. Decoding an incoming APDU is very similar. On input, the top-level decoding routine is called, which calls all other descendant decoding routines. Each decoding routine is responsible for allocating the associated C data structure and storing the input value. The top-level decoding routine returns a pointer to the top-level C data structure. Frame-based memory is used by decoders and encoders so that the constructed data structures can be allocated and freed efficiently.

Converting ASN.1 types directly into C or Pascal data structures is not without problems. Difficulties occur in the translation process because ASN.1 contains data types and facilities not

available in these programming languages. Even simple data types such as `INTEGER` are problematic since BER does not restrict the size of an integer to the usual 16 or 32 bits.

Two data types that are supported by ASN.1 but which do not exist as built-in types in C or Pascal are `SET OF <type>` and `SEQUENCE OF <type>`. It is therefore necessary to create user-defined types that match those in ASN.1. Facilities such as `OPTIONAL` are also problematic. In both Pascal and C there is no counterpart for indicating whether or not a variable exists. Fortunately, there is no data type or facility that can not be represented in C or Pascal, albeit somewhat clumsily at times.

As described in Section 3.4, the object store uses CASN1 internally to write data to disk in BER format.

3.3 Communication service

In order for OSI applications to communicate, mechanisms must be provided for establishing connections and invoking operations. Using Threads and CASN1, DAPE provides the presentation, session, and transport layers, ACSE (Association Control Service Elements), and remote operations support for the application programmer. The network layer is assumed to be provided by the host operating system, whether in the system kernel or as a user process. The communication framework, protocol service, and user interface will be briefly discussed in this section; more detailed descriptions appear in [20].

The framework provides a set of services specific to protocol implementation. This module standardizes communication methods between protocol layers as well as communication between cooperating threads. This standardization allows relatively simple replacement of individual protocol layers, greatly simplifies code readability, and provides a set of libraries that can be reused by each protocol layer.

The communication service is implemented as a group of cooperating threads. The communication thread runs the protocol state machines and acts on requests from user threads. This protocol server uses worker threads comprising readers (to read from the network), writers (to write to the

network), and timers (to provide interval timers). Two forms of communication are employed. Interprocess communication between the protocol server, its workers, and users is standardized by the framework module. All inter-layer communication between the protocol layers and the protocol server is event-driven. These events correspond to protocol state machine events.

The protocol server contains a dispatcher routine that acts as a translator between the two forms of communication. Workers and users send events to the dispatcher thread using Threads IPC. The dispatcher delivers the event to the appropriate protocol layer. For example, the interval timer service uses timer processes that suspend themselves for the required interval and then send a message to the dispatcher, which then posts an event to the appropriate protocol layer. Events are delivered to protocol layers in a uniform way, regardless of their origin, simplifying protocol implementation.

The DAPE protocols provide the remote operations service defined by the CCITT standard X.219 using the X.229 protocol [4, 5]. To support this protocol, the ISO transport, session, presentation, and association control layers are also provided. The transport layer can make use of several network services; Sun X.25, UBC X.25, and the ISO TP0 protocol on top of TCP/IP[25] are currently supported. All of these protocol layers run in the same thread.

Many application threads can exist within a single Threads environment (i.e., a single UNIX process), with each application running as an individual thread. One thread can be designated as the recipient of all connection requests directed at the UNIX process. This thread may handle a connection itself or transfer the connection to another thread. In the latter case, the connection would typically be passed to a worker thread, causing the worker to receive all subsequent operation requests for the association. The worker may perform incoming requests or create other workers to deal with them, making multiple concurrent operations over the same association possible. The general design philosophy is that the application programmer does not have to deal directly with concurrency. Instead, concurrency is obtained by creating multiple threads, resulting in a simpler interface.

It is possible to set up a server thread on one or both ends of a connection so that remote

operation requests may flow in both directions. Each client and server may have zero or more connections with application threads in the same or different UNIX processes. It is also possible for multiple threads to share a connection, directing multiple outstanding requests at the same server.

The user interface makes the runtime communication services available to applications. It is implemented as an upper-half, which provides access to the ROSE (Remote Operations Service Elements) and ACSE routines and a way to block until a request is received, and a lower-half consisting of a state machine. The upper-half comprises stub routines that are called by the application to marshal requests into the form required by the framework module and forward them to the dispatcher. The lower-half of the user interface runs in the communication thread.

The process structure of the communication service is illustrated in Figure 4. The communication service has been tested against other ISO protocol implementations and there are no known interoperation bugs.

3.4 Persistent object store

Many OSI applications require some kind of persistent store. In the case of X.500, each DSA must store its portion of the DIB. To accommodate these applications, a persistent object store was created. This is a generalized facility that can be used for many different applications, not just X.500. The database used by the object store is discussed in the next section.

The object store permits an application programmer to create, delete, read, and store objects within the context of nested atomic transactions. It is capable of operating in a multi-threaded environment. Objects are instances of some programmer defined type, with each type identified by a unique type number. Each object is assigned a system-wide unique identifier, the `ObjOid`. It is certainly possible to store an object's `ObjOid` inside another object. Such an `ObjOid` would act as a kind of pointer to another object. It is therefore possible to build arbitrarily complex structures of objects.

For each object type, routines are specified to serialize (encode) and deserialize (decode) the object, and optionally, to generate keys for the object (index). The encode routine is typically

Figure 4: Organization of the Communication Service

used to flatten a linked list of data structures representing the object into a contiguous buffer; the decode routine restores the data structure. It is possible, for example, to use the encoding and decoding functions generated by CASN1. Because the object is simply being written to disk and not to another open system, however, it is not necessary to use an internationally standardized encoding. Specialized functions can be used to exploit the type information to achieve greater efficiency. For example, C-based encoding rules could be used to increase encoding and decoding speed in exchange for a less space efficient encoding and loss of portability.

The index routine is used to generate a set of keys for a given object. The keys are used to construct inverted indexes, allowing efficient retrieval of objects containing the key. As an example of how this might be used, suppose objects of a given type are structured as attribute-value pairs. If "Colour" is an indexed attribute and if some object has a value of "Blue" for this attribute, then the indexing routine might generate a key such as "Colour=Blue" from this object. The system will automatically maintain an index object that consists of the `ObjOids` of all objects having this key. The precise structure of the key is up to the indexing routine.

Typically, an object is first created using the system's memory management routines. Before the object can be stored, `ObjNew()` must be called. This function is passed the type of the object and returns a new `ObjOid` for subsequent operations. The `ObjOid` has the type information encoded within it. Hence, subsequent operations do not need to pass the type. The object can then be stored via the `ObjSecure()` operation, passing the `ObjOid` and a pointer to the "top-level" data structure of the object. The appropriate encoding and indexing functions are called by `ObjSecure()`. To read an object, the `ObjMap()` operation is used. This operation invokes the decoder to build the appropriate C language data structures and returns the "top-level" structure pointer. The application can then modify the data structures in memory and call `ObjSecure()` to replace the copy in the object store. The `ObjDelete()` operation is used to remove an object from the object store. `ObjLookup()` is passed an object type and key and returns the set of `ObjOids` associated with the key.

In many applications, including X.500, a single remote operation may result in several objects

being updated, created, or deleted. If the host system crashes while processing the remote operation, the object store must not be left in an inconsistent state. Also, if concurrent reading and writing or concurrent update operations are allowed, the object store must guard against inconsistencies that may result.

To deal with these problems, the object store provides nested atomic transactions [10]; most object store operations are executed in the context of a transaction. The object store takes an optimistic approach to concurrency control. This is appropriate in the context of the directory service since 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 (e.g., 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 since 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.

3.5 Database

Many distributed applications, including X.500, have a server component that can handle many client requests simultaneously. In the case of the X.500, the DSA is most naturally implemented as a multi-threaded application, with one or more threads servicing each client request. To maximize the level of concurrency, the database should permit simultaneous read-only and update operations while guarding the database against inconsistencies.

To provide consistency in the face of crashes, assistance for implementing distributed and replicated databases, and multi-threaded operation, the `tdbm` [3] database (`dbm`[9] with transactions) was developed. `Tdbm` provides nested atomic transactions [16], volatile and persistent databases, support for very large data, stores the database within a single UNIX file, and provides 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.

`Tdbm` uses Thompson's extensible hashing technique[30]. In extensible hashing algorithms, a database or hash file is composed of some number of (usually) equal-length pages, with each page holding zero or more items. Most of these schemes aim to retrieve the page holding the item of interest in one or two disk operations as the hash file grows and shrinks. The goal is to maintain this performance without having to do a costly rehashing of all of the items as the size of the hash file changes. While the various approaches vary in their complexity, space overhead, and ease of implementation, they all tend to depend on a secondary data structure, such as a directory or index, to assist in locating the page containing the item. When the occupancy or load factor of a page falls outside its allowed range, a reorganization takes place; e.g., an overfilled page will be split into two partially filled pages and a record of the page split is made in the directory structure.

The `tdbm` library is implemented as three independent layers: the item layer, page layer, and transaction layer. The item layer deals with the layout of key/value pairs in a page. There are two kinds of pages. The first kind is similar to that used by `dbm` and contains a directory for the items stored in the page and zero or more items. All of these pages are the same length. The second kind, indirect pages, are a variable number of physical pages long and simply contain data values that are too large to fit in any normal page.

The page layer deals with management of logical pages, allocation of physical pages, page caching, and the mappings from keys to logical pages and logical to physical pages. It is not concerned with the page contents.

The transaction layer provides nested transactions over logical pages. The transaction layer is responsible for locating the appropriate version of a page for a transaction, concurrency control, and commit and recovery processing.

Every page that is read from the database is cached as a top-level or base copy [4]. All transactions that read the page share this base copy. If a subtransaction updates or creates a page, it retains a private copy of the page that may be later accessed by it and its subtransactions. The correct

[4] Currently, this copy is kept in memory only as long as some transaction needs it.

instance of a page for a particular subtransaction is quickly located by associating a simple hash table with each transaction identifier. The search process involves examining the transaction's hash table, proceeding up the hierarchy through its superiors' hash tables, and finally reading the database, if necessary, until the page is found. As subtransactions commit, their state is merged with that of their parent; when a top-level transaction commits, the new pages are propagated back to the hash file.

Apart from reading base pages from the database during transactions, writing pages to the transaction file while preparing to commit, and copying pages from the transaction file to the database during the commit, no other file I/O is performed. When dynamically allocated pages are no longer required, they are freed for general use by the application.

Threads provides a lock manager that allocates, obtains, and releases a lock on behalf of a client thread. The `tdbm` library uses Threads semaphores to protect critical sections. As a tradeoff between overhead and the level of concurrency, concurrency control is performed at the page level rather than at the hash file level or the item level. Before a page can be read, `tdbm` obtains a read lock for the page. `Tdbm` must obtain a write lock for a page, perhaps by upgrading a read lock that the transaction already holds, before a page can be written. In keeping with strict two phase locking [16], locks are not released until the top-level transaction commits or an abort occurs. When a subtransaction commits, the parent transaction inherits any locks.

Commit processing of a top-level `tdbm` transaction is done by creating a transaction file (also called an *intention list* [28]) that represents the actions that must be executed to update the database. This approach is known as *after-image physical logging* [11].

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. This recovery procedure is idempotent; if the system crashes during the overwriting process, recovery can be retried until successful.

One of the original design goals called for databases capable of being 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. `Tdbm` does this by providing a "precommit" (prepare to commit) operation and a way of determining at restart time whether there was a distributed transaction in progress, and if so, which databases were involved and which phase the transaction was in.

4 DSA Operation

When the DSA is started, it reads a configuration file. This file tells the DSA what its name and address are, where the object store is located, and specifies various administrative limits, such as the time limit for an operation. It is possible to run multiple DSAs on the same host by having each instance consult a different configuration file on start up.

Next, the ADT (Attribute Definition Table) file is loaded. The ADT describes each recognized attribute name, giving its syntax type, standard object identifier, and flags for special handling (e.g., whether the attribute should be indexed).

The DSA proceeds to read the ODT (Object Definition Table) file. The ODT defines, for each object class, its standard object identifier and the kinds of attributes objects belonging to the object class are permitted to have.

The DSA then waits for a DUA or another DSA to bind to it, starting a new dispatcher thread to handle the association. After authenticating the requestor, the dispatcher awaits requests. A new thread is created to execute successive requests. The dispatcher exits when the association with the requestor is terminated. Figure 5 illustrates this structuring.

5 The Local DIT

The local DIT is built out of object entries written to the object store. Every entry contains its own `ObjOid`, distinguished name, RDN, attribute set, and flags indicating if the entry is an alias or if it refers to a remote naming context. An entry contains a number of optional pointers to other

Figure 5: DSA Process Structuring

entry objects in the DIT: its parent, left-most descendent, and right sibling. An entry optionally contains information about each of its children, making the List operation more efficient since it avoids having to read each child object. Figure 6 shows the ASN.1 definition for an Entry.

```
Entry ::= SEQUENCE {
flags           [0] BIT STRING {
       entryAlias(0),
       entryRemotePartition(1) } OPTIONAL,
me              [1] ObjOid,
superior        [2] ObjOid OPTIONAL,    -- root has no superior
leftDescendant  [3] ObjOid OPTIONAL,
rightSibling    [4] ObjOid OPTIONAL,
name            [5] DistinguishedName,
rdn             [6] RDN,
attrSet         [7] SET OF Attr OPTIONAL,
children        [8] SET OF Child OPTIONAL }

Child ::= SEQUENCE {
child           [0] ObjOid,
rdn             [1] RDN,
flags           [2] BIT STRING {
       childIsAlias(0),
       childIsRemote(1),
       childHasDetectAccess(2) } OPTIONAL }
```

Figure 6: ASN.1 definition for an Entry

The inverted index scheme uses Lookup objects that are transparently maintained by the system. They consist of a variable length key and a set of `ObjOids` that are associated with the key. A number of attribute types are currently marked for indexing in the ADT; organization, common name, surname, title, and e-mail address, for example. The surname attribute is also marked for indexing based on its Soundex code [14] so that searches can be conducted without knowing exact spelling. Also, all distinguished names and knowledge references are indexed.

All objects are serialized by the object store using CASN1 encoding functions. Originally these serialization functions were specially coded. Experimentation showed that the functions generated by CASN1 used with the BER library, however, greatly reduced the size of the database on disk with modest additional computational overhead. Also, automating the generation of these functions made experimentation with the entry format much easier, so they replaced the hand-written code.

The local DIT is contained within a single `tdbm` database, which may either be persistent or volatile (completely contained in main memory). Entries returned by remote reads and searches can be stored in a separate `tdbm` database of cached objects. For cached entries, only the distinguished name is indexed.

After an initial object store has been created, either by an initialization program or by the DSA itself, the DSA can begin to serve clients. A special non-interactive user agent can be used to load entries into the local DIT. It connects to a specified DSA, reads entries in text form, and sends appropriate commands to the DSA to add or replace entries.

The OSI directory design has an inadequacy with respect to bulk loading of entries, such as when initially populating the local portion of the DIT. To guard against inconsistencies arising because of system crashes, the AddEntry operation must take place within an atomic transaction in the DSA. There is no means within the X.500 recommendations by which a user agent can group a series of operations, however, and so each AddEntry must end by a transaction commit. This has to be expensive, since an atomic transaction requires the use of stable storage. Loading could be made more efficient if there was a mechanism for a user agent to start, commit, and abort transactions. Of course, more efficient loading can be achieved by non-standardized means.

6 The Distributed Search Algorithm

The directory service provides a Search operation that asks the directory to return all entries (starting at some distinguished name) or only those that satisfy a boolean filter expression that makes assertions about the presence or values of attributes. Much of the design of the EAN DSA is geared toward making Search efficient.

The EAN DSA's Search operation can take advantage of inverted indexes and its multi-threaded environment. When searching through a local portion of the DIT using a common subset of possible filter expressions, the inverted indexes can be used to greatly decrease the number of objects that are read from the object store. If, in the process of a Search, remote references are encountered, new threads can be created to continue the Search in parallel. When all threads complete, the

results are combined and returned to the requestor.

Heuristics can sometimes be used when a Search finds that it must continue at another DSA. The Search operation can be very slow if it involves contacting many remote DSAs, so it is desirable to try to "prune" naming contexts from the search tree. We have looked at two heuristics, one based on disseminating attribute information and the other on caching information concerning searches that do not return results. Other approaches to speeding Search are given in [23].

In the former method, DSAs maintain an attribute bitmap for some standardized attribute types. When a new entry is added, each attribute value of every designated attribute type present is hashed to compute an index into the attribute bitmap for the attribute type and the corresponding bit is set[1]. These attribute bitmaps are disseminated to other DSAs and associated with the access point of the bitmap's creator. When a Search reaches an access point, the filter expression is evaluated against the attribute bitmaps to determine whether an entry exists that satisfies the filter. The filter can involve tests for equality (e.g., "`Surname=smith`") or approximate (Soundex) match (e.g., "`Surname approx smythe`"); the attribute bitmaps are not helpful for some filters, but simple equality tests appear to be used most often. A hashing collision can cause a naming context to be examined unnecessarily.

This scheme requires an extension to the protocol used between DSAs (the Directory System Protocol) so that the attribute bitmaps can be disseminated. Operations, including Search, may return a set of cross references which are cached by the DSA. Subsequent name resolution may be able to take advantage of the cross references by starting its resolution "closer" to the entry in question.[5] In the case of Search, each cross reference can optionally include its attribute bitmaps, thereby propagating the DSA's knowledge. Should a DSA obtain a filter bitmap for an access point for which it already has a filter bitmap, the two bitmaps can be logically ORed together if they are the same size. By associating a time stamp with each bitmap, it is possible to avoid retransmitting bitmaps that have not changed.

[5] It is possible for a new cross reference to effectively cause a Search to bypass a useful bitmap associated with a superior access point. A more sophisticated scheme might take this into account.

In the second heuristic, the first time a Search that is using a filter encounters a particular access point, it associates a bitmap with the remote DSA. Whenever a Search fails to return any entries that satisfy the filter, the filter is hashed to compute an index into a bitmap and the corresponding bit is set. When a later Search reaches the same access point, the filter is again hashed and the bitmap checked to determine whether the access point should be followed. To keep the cached information from becoming ineffective (i.e., false negative responses are being generated), a set of bitmaps can be kept for the access point with each bitmap representing a different time period. Old bitmaps can eventually be discarded, eliminating the least recently used information. This heuristic may occasionally fail because of hashing collisions; a user may wish to retry the search with caching disabled (perhaps causing the DSA to revise its bitmaps). The filter bitmap method can be used with any filter expression, does not require any extension to the X.500 protocol, and it is not necessary to distribute the filter bitmaps. A disadvantage is that there will be more unsuccessful searches than would be the case in the previous approach. This method is useful, for example, when an entry is moved or removed as searches following the first unsuccessful one need not contact remote DSAs.

Ideally, the size of a bitmap should be proportional to the total number of entries beneath an access point. A bitmap that is too small will result in too many collisions and unnecessary searches (by the first heuristic) or false negative responses (by the second heuristic). On the other hand, very large bitmaps will lead to increased communication costs. The matter is complicated by the fact that the number of entries changes dynamically as the DIT is updated but a bitmap size cannot.

The EAN DSA's Search algorithm is outlined below. Details, such as abandoning remote searches and error handling, have been omitted.

```
/* Search the DIT starting at the given entry and applying the given
 * search filter. */
Search(Entry, Filter)
{

    if Entry is a remote naming context then
        Unexplored = [Entry->DSA, Entry->name]
        return remoteSearch(Unexplored, Filter)
    else
        return localSearch(Entry, Filter)
```

```
    end
}

/* Conduct a remote search at each access point in the list of unexplored
 * DSAs. */
remoteSearch(Unexplored, Filter)
{

    for each [DSA,Name] in Unexplored
        /* Invoke heuristics for possible pruning of this naming context. */
        if pruneAccessPoint(DSA, Filter) = False then
            if maximum concurrency level has been reached for this search then
                wait for a remoteSearchThread to finish
                collect and merge its results and update bitmaps
            end
        end
        create remoteSearchThread(DSA, Name, Filter) process
    end

    while there is an active remoteSearchThread
        wait for a remoteSearchThread to finish
        collect and merge its results and update bitmaps
    end
}

/* The remote search process tries access points for the DSA until
 * a connection is established. */
Process
remoteSearchThread(DSA, Name, Filter)
{

    for each access point AP for DSA
        if bind to AP is successful then
            invoke Search(Name, Filter) at AP
            return result and exit
        end
    end
}

/* Search a portion of a locally held naming context.
 * A filter is simple if it consists of a single item or a sequence
 * of items arbitrarily combined by logical ORs and/or ANDs.  Also, only
 * equality matches or approximate matches are allowed and all attributes
 * must be appropriately indexed. */
localSearch(Entry, Filter)
{

    if Filter is simple then
        lookupList = localLookup(Filter)
        for each ObjOid in the lookupList
            E = ReadEntry(ObjOid)
            if E is below Entry in the DIT
                append E to the results
```

```
            end
        end
        let Unexplored be the set of all remote naming contexts that are
            below Entry in the DIT.
    else
        return traverseSubtree(Entry)
    end

    if |Unexplored| > 0 then
        remoteSearch(Unexplored, Filter)
    end
}

/* Recursively evaluate the simple Filter expression, looking up
 * each component of the expression using the inverted indexes and
 * returning the ObjOids of all entries satisfying the expression.
 * For example, if Filter="Surname=Neufeld or Title=professor",
 * one lookup will be done to locate all entries with the key
 * "Surname=Neufeld" and another for "Title=professor", and the
 * resulting lists of ObjOids will be the union of these two.  */
localLookup(Filter)
{
}

/* Conduct a recursive, depth-first search starting at Entry
 * reading each subordinate entry, applying Filter to each entry,
 * and returning a list of successful matches. */
traverseSubtree(Entry, Filter)
{
}
```

7 Distributed Name Resolution

Most of the X.500 directory operations take the name of an entry as an argument. The task of name resolution is to determine if a purported name refers to a locally held object, an object in another DSA's naming context, or no object at all. The EAN DSA implements both the standardized name resolution algorithm and a resolution algorithm based on descriptive names [17, 21]. A third method called "typeless names" has been investigated. Each method will be described in turn.

7.1 Distinguished names

A distributed name resolution algorithm for distinguished names is provided by the X.500 recommendations. The EAN DSA takes advantage of caching (when allowed by the requestor) and makes

extensive use of inverted indexes to implement the algorithm efficiently.

When resolving a purported name, a check is first made to see if the distinguished name has been cached as invalid. Users often misspell names (e.g., when trying to Read a specific entry) and contacting a remote DSA is relatively slow. If the DSA finds that a remote distinguished name does not exist, it caches the fact so that a subsequent request involving the same name can immediately be rejected. The cache is periodically cleaned up in case old negative results become valid. The directory service allows the user to indicate that caching not be used in this and other circumstances.

If a name is not known to be invalid, the next step is to find the "closest" naming context. This is done by a combination of lookups on the distinguished name and all known naming contexts and cross references. If the entry is local, it is read to see if it is a remote naming context. Only four object store operations are typically required. If it is determined that the object is local but could not be looked up, a tree walk is performed. Because of the way entries are currently indexed, this should only happen if the object does not exist and is not strictly necessary.

If the object is determined to be remote, both name resolution and the requested operation are continued at the remote DSA in a process called chaining.

As a special case, if the operation is a Read, an `ObjLookup()` can be done to see if the object has been cached. The distinguished name of the object is used as the key.

A difficulty with distinguished names is that the user must know the DIT hierarchy from the root down to the entry of interest. The Search operation can be used to determine the distinguished name of an entry, however, given a starting point in the DIT and an appropriate filter to reduce the number of entries returned. The EAN DSA's use of indexing makes this operation very fast when the entry is stored locally. A user may set a service control to limit the search to locally held naming contexts, so other DSAs can be pruned from the search. Returning to our example, a search under {`C=CA`} with a filter of `'CN=Peter Smith'` would return Peter Smith's entry, including its distinguished name.

The EAN DSA's algorithm is outlined below.

```
DistinguishedResolve(Name)
{

    repeat    /* to resolve an alias */
        let ObjOid be the entry having the closest remote naming context for
            Name or the local entry itself.

        if ObjOid is an entry for a remote naming context at DSA then
            result = invoke DistinguishedResolve(Name) at DSA
            return result
        end

        Entry = localNameResolve(Name, ObjOid)
        if Entry is not an alias return(Entry)
        Name = DistinguishedNameOf(Entry)
    end
}

localNameResolve(Name, ObjOid)
{

    repeat
        Entry = ReadEntry(ObjOid)
        if Entry is remote naming context return(Entry)
        if Entry is an alias then
            if alias points to another alias return(Error)
            derefEntry = DereferenceAlias(ObjOid)
            return(derefEntry)
        end
        if all RDNs in Name match those in Entry return(Entry)
        let Name be the distinguished name of an immediate subordinate
            entry that matches the next RDN in the purported name.
        Entry = LookupName(Name)
    end
}
```

7.2 Descriptive names

With distinguished names, the user must know the DIT hierarchy from the root down to the entry of interest. Any change to the hierarchy may result in this path name changing. Also, for reasons of autonomy and security, an organization might not want to have its corporate structure revealed. Descriptive names provide a solution to these problems since a user need only provide enough attributes of the desired entry to identify it unambiguously. Because descriptive names do not

reflect the name hierarchy, they are more convenient for users than distinguished names. It is not necessary to populate the DIT with a large number of alias entries to produce reasonable names for objects since there can be many descriptive names for an object. It is also not necessary to restrict the design of the DIT hierarchy for reasons of simplicity or security.

Descriptive names are an unordered set of AVAs. In general, only as many AVAs are required as are necessary to uniquely identify the object. One possible descriptive name for Peter Smith is {`C=CA, Org=UBC, CN=Peter Smith`}. In this case, it is unnecessary to include the organizational units "Science" and "CS". If there is more than one object with attribute `'CN=Peter Smith'` at UBC, then we would have to include more information. Any attribute from the desired entry or any superior can be added. The attributes are not restricted to those used in an RDN; any attribute type designated as a *naming attribute* can be used. For example, they may include a telephone number or e-mail address.

Like the Search operation previously described, descriptive names can be used to provide inverse name resolution. The descriptive name can be used to obtain the distinguished name for the entry.

In exchange for the flexibility and convenience of descriptive names, additional computational effort is required for name resolution. The algorithm for descriptive names is given in [17].

7.3 Typeless names

We have investigated another non-standardized means of making naming more convenient. A typeless name is a hierarchical name that allows the attribute type specification from each RDN to be omitted and restricts each of these name components to a single attribute value. In practice, most RDNs consist of a single AVA, so few names will not be expressible. The typeless name will have as many RDNs as the distinguished name for the same object. For example, {`CA, UBC, CS, 'Peter Smith'`} might be used to identify "Peter Smith".

Typeless names are implemented by assigning a special wild card attribute type to those RDNs for which the user has omitted the attribute type. As long as the given attribute value uniquely determines an entry at the corresponding level of the DIT, a distinguished name can be constructed

simply by replacing the wild card RDN with the true RDN for the entry. Once again inverted indexes allow this to be done efficiently since a single object store `ObjLookup()` can determine the mapping of an attribute value to an RDN.

8 DSA Security

There are many aspects to the problem of providing security for the DSA. For example, a DSA must be able to authenticate or verify the identity of both ordinary users and system administrators. The EAN DSA implements X.500's Simple Authentication scheme, requiring a user to present a distinguished name and its corresponding password. As part of DIT initialization, an entry is created for the administrator and a password AVA is associated with it. A user must present the administrator's distinguished name and password before maintenance commands can be issued. Although digital signatures have been integrated into the X.500 protocol to guard against tampering with a request or result, they are optional and have not been implemented. Passing cleartext passwords through the network obviously provides only a modicum of protection against unauthorized access since an eavesdropper can intercept them. This is, however, a common approach and is used by many systems.

Access control is left as a local matter by X.500. The EAN DSA allows each entry to have an optional access control list (ACL) attribute. Access control lists are the subject of the next section.

8.1 ACLs

The philosophy behind the design of the EAN DSA's ACLs is that users will want to be able to modify their own entries in the directory and should be responsible for doing so. Most attributes in these entries will be publicly readable. The DSA administrator is responsible for creating and updating entries that do not represent persons. Maintenance of an entry can be delegated by setting the ACLs to allow others to modify the entry. Figure 7 gives the ASN.1 description of the ACL.

If the distinguished name of the DUA (i.e., the authenticated name provided by the user) is equal to the distinguished name of the entry, then the user is considered to be the owner of the

```
ACL ATTRIBUTE
WITH ATTRIBUTE-SYNTAX ACLEntry
MATCHES FOR EQUALITY ::= {
  attributeType 5
    -- An EAN object identifier
}

ACL ::= SET OF ACLEntry

ACLEntry ::= SEQUENCE {
  name [0] DistinguishedName,
  attrType AttrType OPTIONAL,
  access [1] Access
}

Access ::= BIT STRING {
  detectAccess(0),
     -- entry and attributes can be listed
  compareAccess(1),
     -- presented value can be compared
     -- against entry
  readAccess(2),
     -- attributes can be read
  modifyAccess(3),
     -- attributes can be updated
  deleteAccess(4),
     -- entry or attributes can be removed
  addAccess(5)
     -- can add child entries
}
```

Figure 7: ASN.1 Definition for the ACL

entry. The owner of an entry always has all permissions and does not need to set ACLs to specify this.

An ACL has three components: one for the ACL itself, one for the ACL password, and one for the rest of the entry. By default, the ACL's "self" component gives no permissions, the password attribute grants `detectAccess` and `compareAccess` permissions, and entries are given public `detectAccess`, `readAccess`, and `compareAccess`. These can be changed later by those with permission to write the ACL attribute.

The ACLs of an entry's superiors are not significant when access rights are checked for an entry. All the permissions for an entry exist with the ACL attributes for that entry. This decision was made mainly for performance reasons, as searching up the tree for all applicable ACLs would be costly.

9 Maintenance

Like access control, directory system maintenance operations are outside the scope of the X.500 recommendations. New remote operations were introduced to the EAN DSA to inspect and alter the DSA's cache, knowledge references, and other state information. Only a properly authenticated user may execute the DSA maintenance operations. The command-line oriented DUA provides a way for an administrator to connect to any EAN DSA and issue maintenance-related commands.

Several operations are provided to administer the cache. The contents of the cache can be listed and the entire cache can be cleared. Another set of operations is available to maintain naming contexts. A context can be verified, removed, or added. The set of known contexts can be listed. Known cross references can be listed, removed, and verified. An operation is provided to list information about all active associations. The EAN DSA's maintenance operations are summarized in Figure 8.

A user agent is available that can connect to any EAN DSA and dump its local DIT in text format. This is the same format used by the loading program, so it is easy to do remote backups

Operation Name	Function
AddContext	Add a new naming context.
ListContext	List all naming contexts.
CheckContext	Validate one or all naming contexts by resolving them.
RemoveContext	Remove a naming context.
AddCrossRef	Add a new cross reference.
ListCrossRef	List all cross references.
RemoveCrossRef	Remove a cross reference.
CheckCrossRef	Validate one or all cross references by resolving them.
ListCache	List the distinguished names of all cached entries.
ClearCache	Remove all cached entries.
Status	List information about each current association.

Figure 8: Remote Maintenance Operations

and restorations.

10 Performance

In this section, the performance of the EAN DSA is examined and some representative execution times are presented for a set of common operations.[6] The first experiment looked at a set of operations conducted with the DIT on disk and again totally in memory. The second experiment investigated the scalability of the DSA by more than doubling the size of the DIT. Protocol efficiency was the subject of the last experiment. All experiments were done on a Sun Sparcstation 1 with 24Mb of memory.

In the first experiment, times are displayed for a disk-based configuration and a memory-based configuration, both having the same 5,900 entry DIT with almost all of the entries under the RDN "`Org=ubc`". Immediately beneath `Org=ubc` was the RDN "`OU=cs`", which had 170 entries that ranged in size from about 600 bytes to 13Kb. About 60 of these entries contain photos, varying in size from 2Kb to 12Kb. The entry for `OU=cs` itself was almost 12Kb.

The measurements are presented in four categories. The Invoke component covers the time from

[6] These are process virtual times as reported by `getitimer()`, accounting only for time when the user process is executing. An average of several runs is reported. The interval timer is not particularly accurate; the maximum variation for longer runs of the same configuration was sometimes as much as 10%.

just before the DSA decodes the remote operation request to invoke the operation (here, Read, List, or Search) up to and including the time for the DSA to transmit the reply.[7] Resolution covers the time to perform distinguished name resolution for the operation. The time to execute the requested operation proper and construct the result (which may be one or more entries or an error report) falls under Operation and the Result category covers the time to encode the result and submit it as the reply to the remote operation request. The Invoke time includes the Resolution, Operation, and Result times. The measurements appear in Figure 9. When considering these results, it should be noted that the DSA is "production quality" software, complete with extensive error checking and event logging; little time has been spent fine tuning the database software. The database layer (beneath the object store) takes roughly 0.01 to 0.02 seconds to retrieve an item from disk.

Perhaps surprisingly, the results indicate that keeping the database in memory gives little or no performance improvement. This is largely because the database layer and object store necessarily (because of their support for transactions) perform caching.[8] Also, the database layer is based on an external hashing method that can typically access an entry in one or two UNIX `read()` system calls.

The cost of reading and decoding a large non-leaf entry can be seen when the time for name resolution of `Org=ubc` is compared to that of `'CN=gerald neufeld'`. The `Org=ubc` entry is about 16.6Kb when encoded and about 50Kb when decoded into its C representation. The `'CN=gerald neufeld'` entry is 582 bytes when encoded and 3.4Kb when decoded. Non-leaf entries are often large because they can contain information about their child entries, offering a substantial speed improvement to the List operation. This tradeoff is expected to be worthwhile since we anticipate that users will often interact with the DSA by browsing through the DIT, making heavy use of the List operation. This feature is a configuration option of the DSA and can be adjusted or disabled.

For operations that return a large result (such as Search), encoding the result can be seen to make up a large proportion of the total cost. It is clear that substantial improvements in performance

[7] In all experiments, the time to decode the remote operation request was less than the resolution of the clock.

[8] There is currently no caching outside of the transaction mechanism; the second of two sequential operations will not benefit from any database-level caching.

Command	Invoke	Resolution	Operation	Result
List `OU=cs` (170 entries)	0.35 / 0.35	0.22 / 0.21	0.02 / 0.02	0.11 / 0.11
Search under `Org=cs` with no filter, returning all 170 entries (includes some photos).	11.71 / 10.92	0.22 / 0.25	8.47 / 7.54	3.01 / 3.13
Search under `Org=ubc` for `Surname=smith`, returning 43 entries.	2.52 / 2.35	0.22 / 0.23	1.92 / 1.75	0.38 / 0.37
Search under `Org=ubc` for `CN='gerald neufeld'`, returning 1 entry.	0.50 / 0.51	0.22 / 0.22	0.27 / 0.28	0.01 / 0.01
Search under `Org=ubc` for `CN=bogus`, returning no entries.	0.43 / 0.48	0.20 / 0.23	0.22 / 0.25	0.01 / 0.01
Read `OU=cs, CN='gerald neufeld'`, returning one entry.	0.11 / 0.10	0.04 / 0.05	0.03 / 0.03	0.02 / 0.02
Read `OU=cs, CN=bogus`, returning no entries.	1.16 / 0.97	1.10 / 0.92	– / –	0.01 / 0.01

Figure 9: Times for Some Common Operations (disk/memory, in secs)

could be made if the encoding and decoding of BER data could be speeded up. Our current work on this problem is discussed in Section 11.

After the experiments described previously were run, an additional 16,000 entries (roughly 6Mb in external text form) from the University of Michigan were added to the disk-based configuration to examine the scalability of the DSA. In addition to repeating the previous experiments, some new queries covering the entire local DIT were performed. These results appear in Figure 10.

Command	Invoke	Resolution	Operation	Result
List `OU=cs` (170 entries).	0.42	0.24	0.06	0.12
Search under `Org=cs` with no filter, returning all 170 entries (includes photos).	11.74	0.21	8.51	3.01
Search under `Org=ubc` for `Surname=smith`, returning 43 entries.	5.26	0.22	4.69	0.36
Search under `Org=ubc` for `CN='gerald neufeld'`, returning 1 entry.	0.50	0.22	0.26	0.01
Search under `Org=ubc` for `CN=bogus`, returning no entries.	0.47	0.22	0.23	0.01
Read under `OU=cs, CN='gerald neufeld'`, returning one entry.	0.10	0.04	0.03	0.02
Read under `OU=cs` for `CN=bogus`, returning no entries.	1.21	1.17	–	0.00
Search under `Org=ubc` and `Org=umich` for `CN='gerald neufeld'`, returning 1 entry.	0.68	0.04	0.62	0.02
Search under `Org=ubc` and `Org=umich` for `CN=bogus`, returning no entries.	0.59	0.04	0.56	0.01
Search under `Org=ubc` and `Org=umich` for `Surname=smith`, returning 133 entries.	6.57	0.03	5.35	1.19
Read `Org=umich, OU=infotech, CN='rose lee'`, returning 1 entry.	0.10	0.04	0.03	0.01

Figure 10: Results of the Scalability Tests (secs)

With one exception, there was no significant difference in execution times when the operations were repeated on the larger DIT. The search for `Surname=smith` was slower because each of the 133 matches had to be examined to select those under `Org=ubc`. A refinement to the indexing scheme has been designed that will largely eliminate this additional time cost.

As a means of testing the efficiency of the protocol stack, an experimental "backdoor" to the

DSA was added that bypasses all protocol layers up to and including the presentation layer, remote operations, and authentication. A thread within the DSA waits for a Read request to arrive in a UDP packet and, if the given name is resolved to a local entry, the entire contents of the entry are returned in ASCII format by a single UDP packet. More elaborate versions of this idea have been proposed as a means of providing directory access to systems that wish to use Internet protocols instead of the ISO protocol stack [12, 26].

The results of an experiment to investigate protocol efficiency are presented in Figure 11. The command-line oriented DUA is the first one listed. It establishes an association with the DSA, invokes a Read operation, terminates the association, and displays the result. The times for the second DUA (the interactive DUA) do not include association establishment or termination times. Both of these DUAs use the ISO protocol stack, with the ISO TP0 protocol on top of TCP/IP. The third DUA uses the simple UDP interface to the DSA. The elapsed virtual time spent in the DSA handling the request (exclusive of Bind and Unbind times) and the elapsed real time for the DUA to show the result are reported. The figures are for the best times obtained for a series of runs of each configuration. In each case, the same entry in the original DIT was read.

Command	Time in DSA	DUA Elapsed Time
1. Bind/Read/Unbind	0.10 / 0.08	0.9 / 0.9
2. Read (no bind)	0.09 / 0.07	0.28 / 0.25
3. Datagram/ASCII	0.07 / 0.05	0.13 / 0.11

Figure 11: Elapsed times for a Read operation (disk/memory, in secs)

Although all of the elapsed times are under one second, the results demonstrate the relatively high cost of connection-oriented communication. Quite early on it was suspected that typical "heavy weight" transport layer connection management would not be suitable for this common form of interaction with a DSA. The RRP protocol [18] was designed to lower the connection establishment component of these interactions. Evaluation of RRP and other means of lowering the response time of simple queries is ongoing.

11 Conclusions

The directory system has proved to be a valuable testbed for the design and evaluation of a wide range of software components, including the Threads kernel, object store, and communication protocols. Also, considerable experience has been gained in the design of multi-process structured applications. Much of this work will continue and be incorporated into new distributed applications.

Figure 12 shows the approximate breakdown of lines of C code (including header files and comments) in the directory system. The figure for the DSA programs was obtained from the top-level code for the DSA and associated maintenance programs. The executable DSA consists of the main program, the DSA libraries, common libraries, object store, tdbm database, Threads, communication service, ASN.1 library, and CASN1 output (i.e., the encoders, decoders, and data structures). Common libraries are used by both the DSA and DUAs; they contain code for handling attributes, loading configuration files, activity logging, etc. The communication service implements network interfaces through to the presentation layer (except for ASN.1 which is accounted for separately).

Module	Lines
DSA programs	3,000
DSA libraries	12,900
Common libraries	14,100
DUA	7,200
XDUA	17,000
Object store	3,500
Tdbm database	6,700
Threads	4,000
Communication service	15,000
ASN.1 library	4,100
CASN1 output	15,000

Figure 12: Code size breakdown

The EAN directory system has successfully interoperated with the QUIPU implementation [13], as well as DSAs implemented by Retix, Nixdorf, and ICL. The DSA has been tested by loading a DIT with over 31,000 entries, with most entries having at least 6 indexing keys.

As can be seen by the performance figures and other experimentation [8], one of the major costs is encoding and decoding the requests and responses. This is particularly significant as the size of the data increases. In response, we have initiated a project to determine more efficient algorithms for encoding and decoding as well as implementing new encoding rules such as Packed Encoding Rules, light-weight encoding rules, and encoding rules that assume a homogeneous language environment.

References

[1] J. Bentley. *Programming Perls*, Addison-Wesley, April 1986.

[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] CCITT. "Draft Recommendation X.219: Remote Operations: Model, Notation and Service Definition", Nov. 1987.

[5] CCITT. "Draft Recommendation X.229: Remote Operations: Protocol Specification", Nov. 1987.

[6] CCITT. "Recommendation X.500: The Directory – Overview of Concepts, Models and Services", Dec. 1988.

[7] D. Cheriton. "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, Vol. 1, No. 2, Apr. 1984, pp. 19-42.

[8] D. Clark and D. Tennenhouse. "Architectural Considerations for a New Generation of Protocols", *Proc. ACM SIGCOMM '90*, (*Computer Communication Review*, Vol. 20, No. 4), Sept. 1990, pp. 200-209.

[9] Computer Systems Research Group,
Computer Science Division, EECS. **ndbm(3)**, *4.3BSD Unix Programmer's Reference Manual (PRM)*, University of California, Berkeley, Apr. 1986.

[10] R. Gruber. "Optimistic Concurrency Control for Nested Distributed Transactions", MIT/LCS/TR-453, June 1989.

[11] T. Haerder and A. Reuter. "Principles of Transaction-Oriented Database Recovery", *Computing Surveys*, Vol. 15, No. 4, (Dec. 1983), pp. 287-317.

[12] T. Howes, M. Smith, and B. Beecher. "RFC 1249: DIXIE", University of Michigan, Aug. 1991.

[13] S. Kille. "The Design of QUIPU", Version 2, Research Note RN/89/19, Dept. of Computer Science, University College London, Mar. 1988.

[14] D. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison-Wesley, 1973, pp. 391-392.

[15] P. Mockapetris. "RFC 1035: Domain Names – Implementation and Specification", USC Information Sciences Institute, Nov. 1987.

[16] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.

[17] G. Neufeld. "Descriptive Names in X.500", *Proc. ACM SIGCOMM '89*, (*Computer Communication Review*, Vol. 19, No. 4), Sept. 1989, pp. 64-71.

[18] G. Neufeld and M. Goldberg. "A Request/Response Protocol for ISO Remote Operations", *Proc. TENCON '90*, Hong Kong, Sept. 1990, pp. 623-627.

[19] G. Neufeld and Y. Yang. "The Design and Implementation of an ASN.1 to C Compiler", *IEEE Trans. on Software Engineering*, Vol. 16, No. 10, Oct. 1990, pp. 1209-1220.

[20] 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.

[21] G. Neufeld. "Descriptive Name Resolution", *Computer Networks and ISDN Systems*, Vol. 23, No. 4, Jan. 1992, pp. 211-227.

[22] 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.

[23] J. Ordille and B. Miller. "Nomenclator Descriptive Query Optimization for Large X.500 Environments", *Proc. ACM SIGCOMM '91*, (*Computer Communication Review*, Vol. 21, No. 4), Sept. 1991, pp. 185-196.

[24] A. Pope. "Encoding CCITT X.409 Presentation Transfer Syntax", *ACM Computer Communication Review*, Vol. 14, No. 4, Oct. 1984, pp. 4-10.

[25] M. Rose and Dwight Cass. "RFC 1006: ISO Transport Service on top of the TCP", Version 3, Northrop Research and Technology Center, May 1987.

[26] M. Rose. "RFC 1202: Directory Assistance Service", Performance Systems International, Inc., Feb. 1991.

[27] M. Rose., J. Onions, and C. Robbins. *The ISO Development Environment: User's Manual*, Volume 1: Application Services, Version 7.0, Sept. 1991.

[28] 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.

[29] M. Thompson, D. Planka, and A. Murdock. National Library of Canada Directory Advisory Group Pilot Project Final Report, Sept. 27, 1991.

[30] C. Torek. "Re: dbm.a and ndbm.a archives", *USENET newsgroup comp.unix*, Apr. 17, 1989.