University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

5-2002

Design and Implementation of the L-Bone and Logistical Tools

Edward Scott Atchley
University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Part of the Computer Sciences Commons

Recommended Citation

Atchley, Edward Scott, "Design and Implementation of the L-Bone and Logistical Tools. " Master's Thesis, University of Tennessee, 2002.
https://trace.tennessee.edu/utk_gradthes/2024

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Edward Scott Atchley entitled "Design and Implementation of the L-Bone and Logistical Tools." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James Plank, Major Professor

We have read this thesis and recommend its acceptance:

Micah Beck, Brad Vander Zanden

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Edward Scott Atchley entitled “Design and Implementation of the L-Bone and Logistical Tools.” I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

James Plank
Major Professor

We have read this thesis and
recommend its acceptance:

Micah Beck

Brad Vander Zanden

Accepted for the Council:

Dr. Anne Mayhew
Vice Provost and
Dean of Graduate Studies

(Original signatures are on file in the Graduate Student Services Office.)

Design and Implementation of the L-Bone and Logistical Tools

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Edward Scott Atchley
May 2002

Copyright © 2002 by Edward Scott Atchley
All rights reserved.

DEDICATION

This thesis is dedicated to my loving wife, Kate. She encouraged me to pursue this goal and, for that, I am deeply grateful. I would also like to dedicate this to my parents, Ed Atchley and Sharon Atchley, for all their love and support over the years.

ACKNOWLEDGMENTS

I would like to thank a number of people whose help and guidance have allowed me to reach this milestone. First, I would like to thank my thesis advisor, Dr. Jim Plank. From his Data Structures class, to his Systems Programming and then Operating Systems classes, I have a deep appreciation for all types of programming. Completing his OS class, in particular, gave me the confidence that I could create anything. I would also like to thank Dr. Plank, along with Drs. Micah Beck and Terry Moore, for allowing me to work with the Logistical Computing and Internetworking Lab.

Much of the work in writing code to create these tools was a team effort. I would like to thank Ding Jin for his efforts in working on the L-Bone and the Logistical Tools. I would like to thank Stephen Soltesz for his long hours and endless enthusiasm while working on the Logistical Tools, the IBPster demo and much more. There would not be the L-Bone or Logistical Tools without IBP and the exNode Library. Thanks to Alex Bassi, Yong Zheng, Xiang Li, Ling Wo and Erika Fuentes for all their hard work and support.

I would like to thank Dr. Brad Vander Zanden for serving on my committee. Lastly, I would like to thank all the professors whose classes I took. I have learned quite a lot and I have thoroughly enjoyed myself at the same time.

ABSTRACT

The purpose of this paper is to outline the design criteria and implementation of the Logistical Backbone (L-Bone) and the Logistical Tools. These tools, along with IBP and the exNode Library, allow storage to be used as a network resource. These are components of the Network Storage Stack, a design by the Logistical Computing and Internetworking Lab at the University of Tennessee. Having storage as a network resource enables users to do many things that are either difficult or not possible today, such as moving and sharing very large files across administrative domains, improving performance through caching and improving fault-tolerance through replication and striping.

Next, this paper reviews the L-Bone, a directory service for Internet Backplane Protocol (IBP) storage servers (depots) which stores information about the depots and allows clients to query the service for depots matching specific requirements. The L-Bone has three major components: a client API, a stateless RPC server and a database backend. Because the L-Bone is intended to be a service available to anyone on the wide-area network, response time is critical. The current implementation provides a reliable service and a fast service. Average response times from remote clients are less than half a second.

Lastly, this paper examines the Logistical Tools. The Logistical Tools are a set of command line tools wrapped around a C API. They provide a higher level of functionality built on top of the exNode Library as well as the L-Bone library, IBP library and the Network Weather Service (NWS) library. This set of tools allows a user to upload a file into an exNode, download the data from that exNode, add more replicas or remove replicas from the exNode, check the status of the exNode and modify the expiration times of the IBP allocations. To highlight the capabilities of these tools and the overall benefits of using exNodes, I perform tests that look at the performance improvements through local replication (caching) as well as tests that look at the higher levels of fault-tolerance through replication. These tests show that using replication for caching can improve access time from 2 to 16 times and that using simple replication can provide nearly 100% availability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	application programming interface
FTP	file transfer protocol
Gb	gigabit
GB	gigabyte
HTTP	hypertext transport protocol
IATA	International Air Transport Association
IBP	Internet Backplane Protocol
IP	Internet protocol
ISO	International Standards Organization
LAN	local area network
LDAP	lightweight directory access protocol
Mb	megabit
MB	megabyte
NAS	network attached storage
NFS	network file system
NWS	Network Weather Service
RAID	redundant array of independent disks
RPC	remote procedure call
SAN	storage area network
TCP	transmission control protocol
URL	uniform resource locator
WAN	wide area network
XML	extensible markup language

1. Introduction

In this paper, I will review the current solutions to sharing and storing data across administrative domains in the wide-area network, from email and FTP to network attached storage (NAS) and storage area networks (SAN). My review will include the benefits and trade-offs associated with each. I will also look at two research projects, OceanStore and WebFS.

I will then outline a view of network storage developed by the Logistical Computer and Internetworking Lab (LoCI) called the *Network Storage Stack*. The Network Storage Stack is composed of layers of protocols similar to the network communication stack (TCP/IP stack). The goal of the Network Storage Stack is "to layer abstractions of network storage to allow storage resources to be part of the wide-area network in an efficient, flexible, sharable and scalable way" [ASP+02].

I will then review, in detail, the design and implementation of two of the components from the Network Storage Stack, the L-Bone and the Logistical Tools. I will show how the L-Bone provides a directory service for IBP depots and proximity measuring between IBP depots. I will also show how the Logistical Tools provide the ability to store and transfer large files as well as provide a measure of fault-tolerance through replication and striping.

1.1 Current and Proposed Solutions

Several solutions exist today to allow people to share files across administrative domains in the wide-area network. Each has its benefits as well as its trade-offs. I will not cover distributed file systems (e.g. NFS and Andrew) since they require that all clients be within the same administrative domain, which is a less interesting problem.

1.1.1 Email

Although email is not intended to be a network storage resource, many people use it to fill that role. In its simplest form, it allows the sharing of small amounts of data (messages). With the use of attachments, one user can send files to another user. Mail servers provide some storage within the network via their incoming and outgoing queues.

The benefits of email for data movement and data storage (i.e. while in transit or stored on the server) are its simplicity, pervasiveness, and sharing of community resources (mail server queues). Another benefit is that the sender does not have to have an account on the receiver's machine.

Most importantly, email is optimized for both the sender and the receiver. Typically, the sender's mail client will forward the outgoing mail to a local mail server, which permits a fast transfer. The local server then automatically routes the message to the recipient's mail server, which holds the message until the receiver is ready to download it.

For all its benefits, email has some glaring drawbacks. First and foremost, most servers limit the size of the file that the user can attach. This limit varies, but 10 MB is on the high side.

If the user needs to sends hundreds of megabytes or even gigabytes of data, email is not an option.

Other limitations to email include the needless duplication of data and the “all or nothing” approach to data access. If several people need to access to a file, the sender must send a duplicate copy to each user, which wastes resources. If the user is only interested in a portion of the file, he must retrieve the entire file first so that he may read the portion he wants. Finally, email does not allow third-party transfers from one remote location to another.

1.1.2 FTP

Another means of allowing remote file sharing is the File Transfer Protocol (FTP). FTP allows a user to store a file to and retrieve a file from a remote machine. Like email, it is widely available, and unlike email, it does not have the restriction on file size that email attachments have.

When a user stores a file remotely using FTP, the stored file has the same properties of any other file on the host machine. Since the operating system and file system will try to maintain that file indefinitely, a remote user can use up the host machine's storage resources. Because of this, most users of FTP require that remote users have accounts on the host machine before allowing writing of files.

The user account requirements are one of the drawbacks of using FTP. Another disadvantage of using FTP is that it is not optimized for the remote user. Since the files reside on the host machine and the transfers are typically single-threaded, downloads may take a long time if the file is large. FTP shares other limitations with email in that neither support partial downloads or third-party transfers. By its nature, FTP exposes the host machine’s directory structure, which may not always be desirable.

A variant of FTP is GridFTP under development by the Globus group. GridFTP adds “new extensions to the FTP protocol for parallel data transfer, partial file transfer, and third-party (server-to-server) data transfer” [Glo02]. Using GridFTP along with another Globus tool, Replica Catalog, will allow the user to optimize the transfer using the “closest” replica or allow the user to transfer from multiple sources in parallel. Although GridFTP addresses many of the concerns regarding FTP, it still requires a remote user to have an account and assumes that any stored data should be permanent. Additionally, the management of replicas requires additional tools.

1.1.3 HTTP

The HyperText Transport Protocol (HTTP) is the protocol that drives the World Wide Web. It allows a remote user's browser to retrieve HyperText Markup Language (HTML) pages and their embedded images with which the browser then builds the web page. HTTP can also be used to allow remote access to stored files via hyperlink. Thus, it provides a very simple means of file sharing, which any modern browser supports. The benefits of HTTP are its widespread usage and simplicity. Also, the remote user does not typically require an account to receive a file.

The disadvantages of HTTP as a file sharing and storing mechanism are many. To implement access controls, the machine's owner must use htaccess (or another password based method) and he must require any remote user to have an account. Like FTP, HTTP does not allow partial transfers and it is optimized for delivery. Most importantly, HTTP does not provide any means for allowing writes from remote users.

1.1.4 Network Attached Storage

Network Attached Storage (NAS) describes a machine that is attached to a TCP/IP network that provides storage. The data traffic flows over the same network as the storage traffic. It typically provides file I/O service rather than block I/O service [Sac01], although the iSCSI initiative would allow NAS devices to provide block I/O service.

The chief benefit is that NAS can provide familiar services, like file I/O service using NFS or block I/O service using iSCSI. On the other hand, NAS is primarily designed for use in local area networks and is not intended to be a sharable resource outside of the local administrative domain. It is not optimized for the remote user.

1.1.5 Storage Area Network

Not to be confused with NAS, Storage Area Networks (SAN) are separate from the regular communication networks with the sole purpose of providing storage. A SAN typically uses FibreChannel to connect clients and the storage device(s) and provides block I/O service [Sac01]. Because of the need to build a network to handle storage requests that is separate from the communication network, SAN is not suited for the wide area.

1.1.6 OceanStore

OceanStore is a project under development at the University of California, Berkeley. It is designed to be a global utility providing permanent data storage. It will not have any centralized state or control. Any server can create a local replica of data to improve access times and fault-tolerance. It allows writes by creating a new version of the file while maintaining all older versions as in a journaling file system. Although OceanStore will not have a centralized state or manager, the developers intend to build a distributed process (i.e. the introspection layer) that monitors the system and then reacts to data requests, failures or attacks. This layer will then migrate data objects, create additional replicas or isolate links under attack [KBC+00].

OceanStore promises high levels of fault-tolerance (e.g. 0.99999+% availability), permanence (i.e. 1,000 year duration), fast access and adaptation to network conditions. It remains to be seen whether than can realize their goals.

1.1.7 WebFS

Part of the WebOS project, WebFS is a global cache consistent file system. It allows reading from HTTP URLs as well as reading from and writing to WebFS sites. The write policies include "last writer wins" and append-only. It relies on public key encryption and access control lists to determine read, write and execute permissions. WebFS is similar to AFS in functionality but it has looser file semantics and it adds the ability to read from the HTTP namespace. Both the WebOS and the WebFS projects have been discontinued. [VEA96]

1.2 Network Storage Stack

The **Lo**gistical **C**omputing and **I**nternetworking (**LoCI**) Lab at the University of Tennessee has been developing an alternative framework for integrating storage in the network, which aims to improve its performance and reliability. Rather than treating storage in the traditional sense just as an attached resource, the LoCI Lab views storage as an integral part of the network. The LoCI Lab calls the combining data storage and data movement, *Logistical Networking.*

Logistical Networking takes the rather unconventional view that storage can be used to augment data transmission as part of a unified network resource framework. The adjective "logistical" is meant to evoke an analogy with military and industrial networks for the movement of material which requires the co-scheduling of long haul transportation, storage depots and local transportation as coordinated elements of a single infrastructure [BMP01].

The design for the use of network storage revolves around the concept of a *Network Storage Stack* (Figure 1). Its purpose is to layer abstractions of network storage to allow storage resources to be part of the wide-area network in an efficient, flexible, sharable and scalable way. It is modeled after the IP stack, which achieves all these goals for data transmission, and its guiding principle has been to follow the tenets laid out by End-to-End arguments [SRC84, RSC98]. Two fundamental principles of this layering are that each layer should (a) *abstract* the layers beneath it in a meaningful way, but (b) *expose* an appropriate amount of its own resources so that higher layers may abstract them meaningfully (see [BMP01] for more detail on this approach).

In this section, I review the middle three layers of the Network Storage Stack. In chapters 2 and 3, I give more detailed descriptions of the L-Bone and Logistical Tools. The bottom two layers are simply the hardware and operating system layers of storage. The top two layers, while interesting, are future functionalities to be built when we have more understanding about the middle layers.

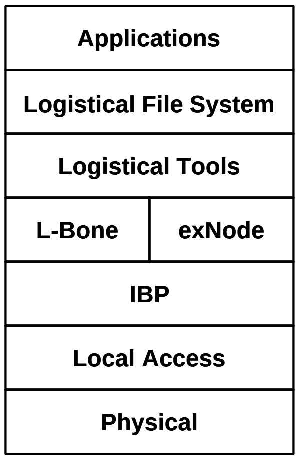


Figure 1: The Network Storage Stack

1.2.1 Internet Backplane Protocol

The lowest level of the network accessible storage stack is the *Internet Backplane Protocol* (IBP) [PBB+01]. IBP is a server daemon and a client library that allows storage owners to insert their storage into the network, and to allow generic clients to allocate and use this storage. The unit of storage is a time-limited, append-only byte-array. With IBP, byte-array allocation is like a network `malloc()` call: clients request an allocation from a specific IBP storage server (or *depot*), and if successful, a trio of cryptographically secure text strings is returned (called *capabilities*) for reading, writing and management. Capabilities may be used by any client in the network, and may be passed freely from client to client, much like a URL.

IBP does its job as a low-level layer in the storage stack. It abstracts away many details of the underlying physical storage layers: block sizes, storage media, control software, etc. However, it also exposes many details of the underlying storage, such as network location, network transience and the ability to fail, so that higher layers in the stack may abstract these more effectively.

1.2.2 L-Bone

While individual IBP allocations may be employed directly by applications for some benefit [PBB+01], they, like IP datagrams, benefit from some higher-layer abstractions. The next layer contains the *L-Bone*, for resource discovery and proximity resolution, and the *exNode*, a data structure for aggregation. Each is defined here.

The L-Bone (Logistical Backbone) is a distributed runtime layer that allows clients to perform IBP depot discovery. IBP depots register themselves with the L-Bone, and clients may then query the L-Bone for depots that have various characteristics, including minimum storage capacity and duration requirements, and basic proximity requirements. For example, clients may request an ordered list of depots that are close to a specified city, state, airport, US zipcode, or network host.

Thus, while IBP gives clients access to remote storage resources, it has no features to aid the client in figuring out which storage resources to employ. The L-Bone's job is to provide clients with those features. As of early 2002, the L-Bone is composed of 21 depots in the United States and Europe, serving roughly a terabyte of storage to Logistical Networking applications (Figure 2).

In Chapter 2, I review the L-Bone in detail. I will review the design goals and assumptions, the metadata stored in the L-Bone, the RPC call formats that clients use to contact the server, the client API and the L-Bone server’s structure. Lastly, I present data that shows the L-Bone’s response time to clients across the country.

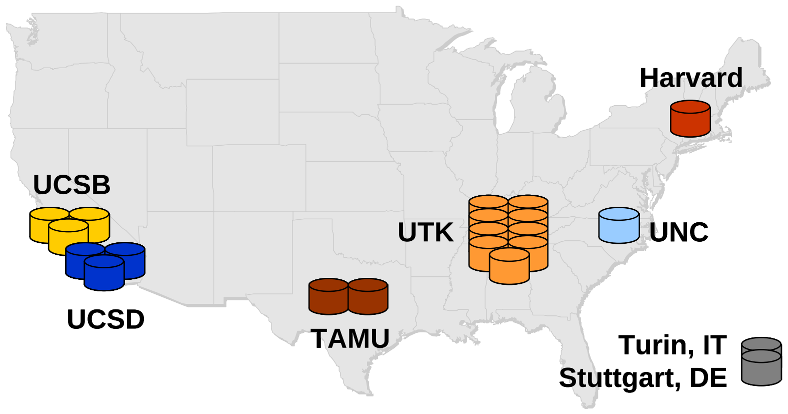


Figure 2: The L-Bone as of early April 2002

1.2.3 exNode Library

The exNode is a data structure for aggregation, analogous to the Unix inode (Figure 3). Whereas the inode aggregates disk blocks on a single disk volume to compose a file, the exNode aggregates IBP byte-arrays to compose a logical entity like a file. Two major differences between exNodes and inodes are that the IBP buffers may be of any size, and the extents may overlap and be replicated. For example, Figure 4 shows three exNodes storing a 600-byte file. The leftmost one stores all 600 bytes on IBP depot A. The center one has two replicas of the file, one each on depot B and depot C. The rightmost exNode also has two replicas, but the first replica is split into two segments, one on depot A and one on depot D, and the second replica is split into three segments, one each on depots B, C, and D.

In the present context, the key point about the design of the exNode is that it allows us to create storage abstractions with stronger properties, such as a network file, which can be layered over IBP-based storage in a way that is completely consistent with the exposed resource approach.

Since our intent is to use the exNode file abstraction in a number of different applications, we have chosen to express the exNode concretely as an encoding of storage resources (typically IBP capabilities) and associated metadata in XML. Like IBP capabilities, these serializations may be passed from client to client, allowing a great degree of flexibility and sharing of network storage. The use of the exNode by varying applications provides interoperability similar to being attached to the same network file system. The exNode metadata is capable of expressing the following relationships between the file it implements and the storage resources that constitute the data component of the file's state:

- The portion of the file extent implemented by a particular resource (the starting offset and ending offset in bytes).

- The service attributes of each constituent storage resource (e.g. reliability and performance metrics, duration).

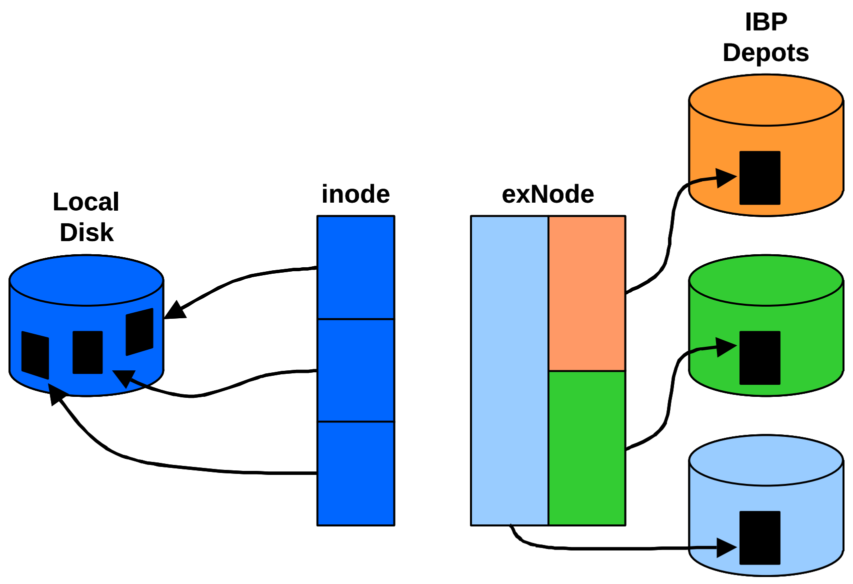


Figure 3: Comparing the UNIX inode and an exNode

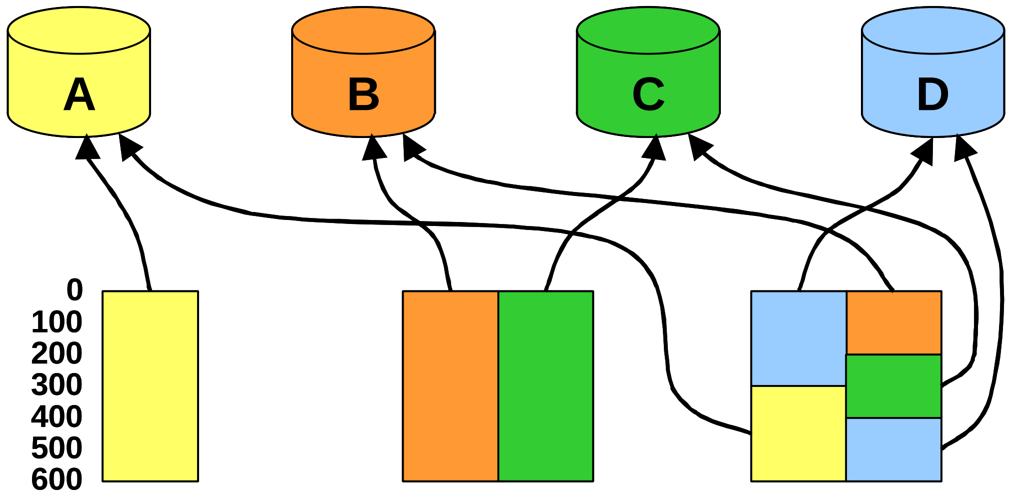


Figure 4: Sample exNodes

- The total set of storage resources, which implement the file and their aggregating function (e.g. simple union, parity storage scheme, more complex coding).

1.2.4 Logistical Tools

At the next level of the Network Storage Stack are tools that perform the actual aggregation of network storage resources, using the lower layers of the Network Stack. These tools take the form of client libraries that perform basic functionalities, and stand-alone programs built on top of the libraries. Basic functionalities of these tools are upload, download, stat/list, refresh, augment and trim. I will cover these in more detail in Chapters 3.2.1 and 3.2.2.

The Logistical Tools are much more powerful as tools than raw IBP capabilities, since they allow users to aggregate network storage for various reasons:

- **Capacity**: Extremely large files may be made from smaller IBP allocations. In fact, it is not hard to visualize files that are tens of gigabytes in size, split up and scattered around the network.

- **Striping**: By breaking files into small pieces, the pieces may be downloaded simultaneously from multiple IBP depots, which may perform much better than downloading from a single source.

- **Replication for Caching**: By storing files in multiple locations, the performance of downloading may be improved by downloading the closest copy.

- **Replication for Fault-Tolerance**: By storing files in multiple locations, the act of downloading may succeed even if many of the copies are unavailable. Further, by breaking the file up into blocks and storing error correcting blocks calculated from the original blocks (based on parity as in RAID systems [CLG+94] or on Reed-Solomon coding [Pla97]), downloads can be robust to even more complex failure scenarios.

- **Routing**: For the purposes of scheduling, or perhaps changing resource conditions, *augment* and *trim* may be combined to effect a routing of a file from one network location to another. First it is augmented so that it has replicas near the desired location, then it is trimmed so that the old replica is deleted.

Therefore, the Logistical Tools enable users to store data as replicated and striped files in the wide area. The actual best replication strategy – one that achieves the best combination of performance, fault-coverage and resource efficiency in the face of changing network conditions – is a matter of future research.

2. L-Bone

2.1 Design

2.1.1 Goals

The primary function of the L-Bone is resource discovery, to allow users to find IBP depots. Users should be able to specify their storage requirements and receive back a list of suitable depots. In addition to just finding depots, users should be able to determine proximity to depots and between depots. Users should also be able to query about the size of the L-Bone.

Additionally, the L-Bone implementation should provide the following:

- The L-Bone should be accessible over the Internet. It should not attempt to maintain state for the depots but instead cache data about them.
- The L-Bone should respond to user requests as fast as possible. If the service is busy or down, the client should have a timeout option so that the user does not block indefinitely.
- It should provide replication to avoid a single point of failure and to spread the load. Geographically dispersing the L-Bone will also improve response times.
- The L-Bone should scale up as more depots are added and more users interact with the service.
- The code should be as portable as possible to allow as many users as possible.

2.1.2 Assumptions

Since this is a research tool, we chose to use C as the development language and UNIX as the development OS. We felt that using C would allow us to maximize performance yet still maintain portability. To test portability of the UNIX code, we checked the code on Solaris (7 and 8), Linux (kernels 2.2 and 2.4), Apple's Mac OS X (Darwin) and AIX.

The L-Bone needs to maintain a certain amount of metadata about the listed depots. Rather than create a database tool from scratch, we selected an open-source application, openldap, to maintain the data. Openldap is optimized for lookups, and since the vast majority of L-Bone calls are reads, and not writes, openldap is a logical choice. For a distributed service, openldap is also easier to setup and maintain than a relational database such as mySQL.

To assist in determining proximity of clients to depots, we use the Network Weather Service (NWS) [WSH99] to determine available bandwidth between locations, the NetGeo service for hostname location [Net02], the US Census Bureau’s database of geographic coordinates for each US zip code and the IATA list of US and international airport locations.

All client/server communication is accomplished through RPC calls. Using RPC allows us to develop clients for other programming languages or operating systems without requiring the use of the C language or UNIX-specific libraries. Lastly, we use the pthread library whenever we need multi-threading to improve portability among UNIX variants.

2.1.3 Depot Metadata

The L-Bone can store a large amount of information about a depot, but the only necessary data for a depot to be accessible is its fully qualified domain name (hostname) and the port used by IBP. With just these two items, the L-Bone can poll the depot periodically and determine the status of the depot.

In order to allow searches for storage space, the L-Bone needs to keep data about the status of the depot. The L-Bone polls all registered depots at a specified interval and stores the results for later user queries. IBP provides a depot query function, `IBP_status()`, that returns the following data:

- The maximum amount of stable storage to be served
- The amount of stable storage currently served
- The maximum amount of volatile storage to be served
- The amount of volatile storage currently served
- The maximum duration it will allow for an allocation

The L-Bone keeps this data but replaces the currently served values with available storage values (total storage less currently served storage) for both stable and volatile.

In order to determine geographic proximity, the L-Bone tries to determine and store the latitude and longitude for each depot. It uses the following metadata to determine that location:

- Country (2 letter ISO code),
- State (US only),
- City,
- Zip (US only) and
- Airport code (3 letter IATA code).

Using the NetGeo service, the L-Bone can also try to determine latitude and longitude based just on the hostname. If the user provides overlapping or conflicting data, the L-Bone chooses the keyword that yields the highest precision. From highest to lowest precision, the categories are hostname, airport, zip, city, state and country.

As the L-Bone continues to grow, it is likely that depots will be run on a wide variety of machines in a wide variety of environments from university machine rooms to home users with persistent connections (cable or DSL). To allow users to select depots that have certain characteristics, the L-Bone optionally keeps the following environmental metadata:

- Type of network connection
- Frequency of machine monitoring
- Power backup availability
- Data backup policy
- Whether the depot is behind a firewall

The L-Bone also has some administrative duties that require additional metadata. To provide a minimal amount of security, the L-Bone requires that someone listing a depot provide his or her email address. The L-Bone uses the email address as the password when the owner wishes to change any of the user-configurable metadata.

Other administrative metadata include:

- Email notification policy if the depot is unreachable
- Last time the owner was notified
- The number of polling attempts
- The number of polling replies
- The status of the depot (used by the cgi scripts only)

Of these, only the email notification policy is user changeable. The rest are for bookkeeping purposes for the L-Bone.

2.1.4 Remote Procedure Calls

All requests from the client to the server are formatted with strings. This avoids the need to worry about incompatibilities between machines with little-endian versus big-endian architectures. So far, we have implemented three RPC calls.

2.1.4.1 Get Depots

For this RPC call, the client sends a 512-byte packet to the server. The server will then return a sorted list of depot and port pairs that meet the request requirements. The client message contains a version number, request type, the maximum number of depots to return, the minimum amount of stable storage in MBs each depot should have, the minimum amount of volatile storage in MBs each depot should have, the minimum number of days the depot should allow for allocations and a location string. All fields are 10 bytes long except location, which is 452 bytes long.

Client to Server

10	10	10	10	10	10	452
version	type	num depots	min stable size	min volatile size	duration	location

Server response to Client

10	10	256	10	
success	num depots	hostname	port	repeat for each depot

Currently, IBP allocations are limited to 32 bits, which can represent a 10-digit string of numbers. In the future, if IBP allows 64 bit (long long) sized allocations, the min stable size and min volatile size fields will need to be expanded to 20 bytes. At this time, the IBP team does not intend to increase to the long long format for two reasons: accessing the data in such a large file would be inefficient and aggregating allocations can attain the large file storage functionality. The exNode library, discussed in section 1.2.3, performs the aggregation.

For this call, the server always returns success; there are no error conditions. The server is allowed to return SUCCESS with a depot count of zero if none matched the request criteria.

2.1.4.2 Count Depots

The client will send a 20-byte message that contains the version number and the request type. It does not send any parameters. The server then returns the number of listed depots and the number of depots that responded to the last poll.

Client to Server

10	10
version	type

Server response to Client

10	10	10
success	total depots	live depots

Again, there is no unsuccessful return message.

2.1.4.3 Get Proximity

For the last RPC call, the client sends a version number, request type and a location string of 492 bytes. The server then returns a list of depots and distance metrics from the specified location.

Client to Server

10	10	492
version	type	location

Server response to Client

10	10	256	10	
success	count	hostname	value	repeat for each depot

2.2 Implementation

2.2.1 Client Library

The L-Bone client library provides five calls to the user:

- `Depot *lbone_getDepots()` – RPC call to server
- `Depot *lbone_checkDepots()` - client only
- `Depot *lbone_sortByBandwidth()` - client only
- `int lbone_getProximity()` - RPC call to server
- `int lbone_depotCount()` - RPC call to server

Each of these functions is a blocking call. To prevent indefinite blocking, each includes a timeout parameter.

2.2.1.1 lbone_getDepots()

The `lbone_getDepots()` call is the primary call in the library. It allows the user to specify storage and duration requirements as well as location hints. The output will be a null-terminated array of data structures containing hostnames and ports for IBP depots that meet the requirements set by the user.

The synopsis is:

```
Depot *lbone_getDepots( Depot lboneServer,
                        LBONE_request req,
                        int timeout );
```

The function requires two data structures, `Depot` and `LBONE_request`. The `Depot` struct is simply a #define of the `IBP_depot` data structure that contains a hostname and a port number. The `LBONE_request` is the following:

```
typedef struct lbone_request {
    int              numDepots;
    unsigned long    stableSize;
    unsigned long    volatileSize;
    double           duration;
    char             *location;
} LBONE_request;
```

The request holds all the information that the server will need to find depots for the client. The first item, **numDepots** sets a maximum number for the server to return although it could return less. Both the **stableSize** and **volatileSize** parameters specify what the minimum amount of each type of storage must be available. If both are non-zero, the depots must have both classes of storage available above the minimums. The **duration** amount is the minimum number of days (or partial days) that the storage allocation must exist.

The **location** field allows the user to associate keyword/value pairs. Some keywords affect which depots will be returned and other keywords affect the order in which depots will be returned. The user may include any combination of keyword and value pairs in the location up to the 452-byte limit.

As mentioned in section 2.1.3, the L-Bone maintains metadata about the environment of the depots such as data backup policy. Keywords that specify environmental metadata restrict which depots are returned. Just like **stableSize** or **duration**, a depot must meet the minimum level specified by the keyword/value pair. For example, for the data backup policy, the options are:

- Daily backups with multiple media
- Daily backups reusing the same media
- Occasional backups
- No backups

When a particular level is specified, that level and all levels above it would meet the requirement. So, if a user specifies "Daily backups with the same media", only depots that backup daily with either the same media or multiple media will qualify.

Geographic keywords do not restrict depots but they affect the order in which they are returned. For example, the client may specify several geographic keywords, such as `state=` and `city=`, with their associated values. The server will not just return depots exactly meeting that state and city (if any did), but instead, it will return the depots based on proximity to that city and state.

Lastly, the `lbone_getDepots()` function takes a timeout value. If the client does not return within this number of seconds, the function will discontinue waiting and return nothing (i.e. NULL pointer) to the user.

2.2.1.2 lbone_checkDepots()

Because the L-Bone server caches data that may be minutes or hours old, the `lbone_getDepots()` call may return depots that no longer meet the user's requirements for storage space and/or duration. To allow the user to check the list of depots, the client library provides the `lbone_checkDepots()` function.

The input is a null-terminated array of depots (as returned by `lbone_getDepots()`). This function returns a subset of that list or the entire list if all are available and still meet the requirements.

The synopsis is:

```
Depot *lbone_checkDepots( Depot *depots,
                          LBONE_request request,
                          int timeout );
```

This function does not contact the L-Bone server. Instead, it contacts each depot directly and uses an `IBP_status()` call to determine:

1. that the depot is reachable and functioning and
2. whether the depot's available storage space and duration still meet the request requirements.

This is a multi-threaded call that contacts each depot simultaneously. It requires the user to compile with the pthread library. Like the `lbone_getDepots()` call, `lbone_checkDepots()` provides a timeout parameter to prevent indefinite blocking unless the client so chooses.

2.2.1.3 lbone_sortByBandwidth()

By default, the L-Bone server returns depots to the `lbone_getDepots()` call sorted by geographic distance if the location field included the proper keywords. There are situations, however, when the user would prefer to sort the depots by highest to lowest bandwidth. The L-Bone client provides the `lbone_sortByBandwidth()` function for this purpose.

The synopsis is:

```
Depot *lbone_sortByBandwidth( Depot *depots,
                              int timeout);
```

This function attempts to determine the available bandwidth to each depot using `nws_ping()`. For each depot in the array, it creates a thread that calls `nws_ping()`. If the `nws_ping()` is successful, it records the bandwidth measured. If the depot does not respond before the timeout or it is not running NWS, the function records a bandwidth of 0. It then sorts the depots and returns a null-terminated array.

This function relies on a single `nws_ping()` call per depot, which can provide widely varying results. In the future, this call will be replaced by a call to the L-Bone server when the L-Bone server is able to provide more NWS forecasting and prediction information to the user.

2.2.1.4 int lbone_getProximity()

One of the goals of the L-Bone is to provide proximity information about depots to users. This first version of the L-Bone can determine geographic proximity to a location specified by the user and return a list of all depots sorted by the inverse of distance.

The synopsis is:

```
int lbone_getProximity( Depot lboneServer,
                        char *location,
                        char *filename,
                        int timeout);
```

This function does contact the L-Bone server, so the first argument is a struct containing the L-Bone server's hostname and port. The second parameter is the user's location string containing keyword and value pairs. This will determine by what location to sort the depots. The filename specifies an output file name. And lastly, the timeout prevents the call from blocking forever if the server is not available or hangs.

The output file format is a space delimited text file. Each line contains a hostname followed by a value. Higher values indicate closer proximity.

2.2.1.5 int lbone_depotCount()

If the user needs to determine the number of depots listed in the L-Bone, he can use `lbone_depotCount()`. It takes as input a struct containing the L-Bone server's hostname and port and, if successful, it fills in two unsigned long pointers. These pointers will have the number of total depots listed and the number of depots that responded to the last poll.

```
int lbone_depotCount( ulong_t *total,
                      ulong_t *live,
                      Depot lboneServer,
                      int timeout);
```

This call may assist the client with its depot request policy.

2.2.2 Server Process - lbone_server

The L-Bone server process, lbone_server, is the RPC server that receives client requests, queries the database and returns an RPC message to the client. It is a multi-threaded program that has two persistent threads and an unbounded number of temporary threads to handle client requests. The operating system, however, may constrain the number of simultaneous processes and/or file descriptors.

When the server starts up, it initializes a `ServerConfig` struct that stores the startup parameters, the listening socket number, a mutex lock and L-Bone state values. This struct will be passed to all threads to allow sharing of these resources. The `server_config` is:

```
typedef struct server_config {
      char              *password;
      char              *config_path;
      int               port;
      char              *ldaphost;
      int               socket;
      ulong_t           totalDepots;
      ulong_t           liveDepots;
      ulong_t           stableListedMb;
      ulong_t           stableAvailMb;
      ulong_t           volListedMb;
      ulong_t           volAvailMb;
      pthread_mutex_t   lock;
} *ServerConfig;
```

As the thread continues, it determines what the startup parameters are. To start, it needs to find the database, the port number that the L-Bone server should open and on which to listen, and a password. If any of these are not found, it exits.

If the server finds the necessary parameters, it tries to maximize its allowable resources. In case the host machine limits the number of user open files (i.e. file descriptors), the server attempts to get its resource limits and increase the allowable number of processes to the maximum (usually 1024).

To avoid exiting when a client connection is lost (SIGPIPE), the server starts a signal handler that intercepts SIGPIPE signals and ignores them.

The server then opens a socket on the port specified by the startup parameters. Once the socket is open, it forks a persistent thread using `pthread_create()` that will monitor the socket.

After opening the socket monitor, the main thread has completed its startup duties and it now it will enter an infinite loop. It will poll every depot listed in the database and update the cached data in the database. After the polling is complete, this thread goes to sleep for a set period of time, currently set at 45 minutes, and repeats the polling when it wakens.

The second persistent thread is the socket monitor. Like the startup/polling thread, the socket monitor enters an infinite loop. It calls `accept()` and blocks until a client initiates a connection. It then creates a `ClientInfo` struct that includes the `ServerConfig` struct as well as the file descriptor for this client. The `client_info` is:

```
typedef struct client_info {
      ServerConfig      server;
      int               fd;
} *ClientInfo;
```

The socket monitor thread then forks a temporary thread using `pthread_create()` to handle this client. It then loops and blocks on `accept()` until the next client request. To ensure that the temporary thread releases its resources as soon as it calls `pthread_exit()`, the socket monitor creates it as a detached thread. The socket monitor thread does not call `pthread_join()` which is equivalent to the UNIX `wait()` function.

The temporary client thread handles one client request and then exits. The client thread parses the first 20 bytes to determine the version number and the request type. It then calls a subroutine depending on the request type. When the subroutine is finished, the thread closes this socket connection and exits.

Currently, the server implements the following requests:

- `handle_client_request()`
- `handle_depot_count()`
- `handle_proximity_list()`

The `handle_client_request()` is the server end to `lbone_getDepots()`. This was the original L-Bone client call. It parses the message, then calls `get_result()` and then `send_list()` subroutines.

The `get_result()` subroutine gets a list of all depots from the database, then it determines if they meet the client's storage and duration requirements. If the depot does, it calculates the depot's distance from the client's location request, if there is one or it generates a random number if the client did not specify a location. Last, it inserts the depot into a red-black tree using the distance as the sort key. The `get_result()` function then returns the red-black tree list to the `client_request` handler. The list is handed to `send_list()` function which returns the sorted list of hostnames and ports to the client.

We added the `depot_count` handler to allow a client to ask how many depots are listed in the L-Bone and how many replied to the last poll. When the server's polling thread contacts each depot, it keeps count of how many depots are in the database and how many replied to the poll. The two numbers are kept in the `ServerConfig` struct that is passed to every thread. This function then simply returns those two numbers to the client.

The last handler is `proximity_list`. It simply parses the client message for the location string, passes that string to the `getProximity()` subroutine and then returns a list of all depots and their geographic distance to the location. This handler was added to provide a rough measure of proximity based on distance that may be used by the exNode clients if no other proximity metric is available.

2.2.3 Database Backend (openldap)

The L-Bone server is a stateless server. The only data kept internally in the server is the number of depots polled and how many responded. All other data about the depots is stored in a backend database. This releases the L-Bone server from all state management responsibility.

In addition to the shifting the responsibility for state management, another benefit from using a separate backend is speed of deployment. We were able to get a system up and running very quickly. The last benefit of this approach is that we can change database systems if another proves to be better (i.e. faster, more reliable, easier to replicate, etc.) by simply re-writing the lookup calls.

Since the L-Bone's primary purpose is to allow resource discovery, and resources would be relatively stable over time (i.e. not requiring many writes), we opted to use a directory application, openldap, rather than a relational database.

2.2.3.1 Directory Structure

Every LDAP (Lightweight Directory Access Protocol) application has its data organized in a hierarchical tree structure. Starting with the root of the tree at the top, it branches into different sub-trees and eventually end with data entries at the leaves.

The L-Bone structure begins with the root node, which is usually the organization in LDAP. Therefore the L-Bone root node is o=lbone (organization equals lbone). This root node currently has four branches or organizational units: depots, zipcodes, airports and locations. The depots, zipcodes and airport branches have no further branches and simply contain leaves. The locations branch has three more sub-branches: countries, states and cities. The L-Bone tree appears in Figure 5.

Rather than use a unique key to reference data entries as in a relational database, an LDAP server uses the path of the object from the leaf back to the root. This path is called the entry's distinguished name. For example, a depot with the name of “dsj.cs.utk.edu” has the distinguished name of:

depotname=dsj.cs.utk.edu,ou=depots,o=lbone

As of early 2002, the L-Bone contains approximately 30,000 zipcode entries and 8,000 airport entries, which include about 6,000 US and 2,000 international airports. The number of depots is in the range of 20 to 25. At this time, we do not use the locations sub-branch since we have not found free databases for countries other than the US that contain city or city and state data along with latitude and longitude values. In the interim, we can find much of the same info on international locations in the airport database, which includes city and country in the entry.

2.2.3.2 Data Entry Structure and Validation via Schema

Depending on the branch of the tree, the data entries may have differing attributes. For depots listed in the depots branch, section 2.1.3 has already listed the required and optional metadata stored in openldap. For each depot, it may have up to 26 attributes, although only four are required: depotname, hostname, port and email address of owner.

s

For items in the US zipcode branch, they must have a zipcode, latitude and longitude. They may additionally have country, state and city. In actuality, all zipcode entries include city, state and country.

The entries in the airport branch must have a 3-letter IATA airport code, longitude and latitude. They may also have country, state and/or city. All US entries have the city and state information, and international airports have country information. The international airports also have the airport name listed under the city attribute. Usually, this will include the city's name, which allows us to provide city/country lookup for non-US cities.

Whenever a new entry is added or an existing entry is modified, the openldap server uses this schema to check the entry and ensure that it includes the required information and that the

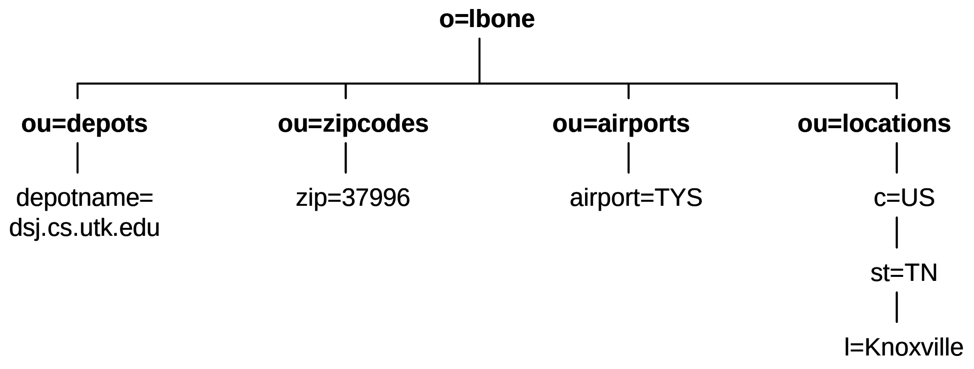


Figure 5: L-Bone Directory Tree Structure

entry does not include any arbitrary attributes not specified in the schema. See Appendices 1 and 2 for the complete L-Bone schema.

2.2.3.3 Finding Data Efficiently

Openldap provides a rich query language that allows the user to target a query for faster responses. The query is based on regular expressions. Any attribute may be included in a search. Attributes that are used frequently in searches can be indexed to improve performance. Currently, the L-Bone's openldap server indexes these attributes: objectClass, depotname, hostname, country, state, airport, zip and city. These items require an exact match, although the match is case insensitive. The city attribute may also be searched for substrings.

Another ldap feature that improves search performance is searching a particular branch. Rather than search the entire L-Bone tree looking for a specific depot or airport, the user may specify the branch where that type of data is located, depots or airports, for example.

The last feature of openldap that we have used to improve performance is caching. Openldap allows the user to specify how many records to keep in memory and how much memory to make available to indices. We have specified that openldap should keep 50,000 records in memory and use up to 1 MB for indices. Since our entire tree contains less than 50,000 records at present, the entire database is held in memory. This ability, along with the tree structure provided by openldap, allows us to provide very quick response to user queries.

2.2.3.4 Replication to Improve Fault-Tolerance

Since the L-Bone is a service available to any user on the Internet, it is important that we avoid single points of failure. If the L-Bone were limited to a single L-Bone RPC server and a single LDAP database, it would not be a very reliable service. Fortunately, openldap includes a simple solution to provide replication.

Openldap's slurpd daemon provides the replication service. The slurpd process looks for a log file that includes any changes made by the LDAP server, slapd. It then forks a child for each slave server and each child updates one slave database. After a child process is complete, it exits.

The final step to adding fault-tolerance is to run a local L-Bone RPC server with each slave LDAP database. The user can then query any of the L-Bone servers and get the information needed. We have successfully run a master and slave openldap servers, each with a L-Bone server. Although openldap provides the option to allow writes at the slave locations, its creators do not recommend it since it slows down read performance. We only allow writes on the master LDAP server. We have found that writes to the master are propagated to the slaves almost instantaneously. Currently, the primary L-Bone server and master openldap server are located on adder.cs.utk.edu on ports 6767 and 6776, respectively. A secondary L-Bone server and slave openldap server are running on galapagos.cs.utk.edu also on ports 6767 and 6776, respectively.

2.3 L-Bone Response Time

To determine an average L-Bone response time, we created a client that would generate a L-Bone request and then call `lbone_getDepots()`. To minimize the benefits of caching within the openldap server, the program would generate a random number that was used to choose the number of depots, the amount of storage and a location string from an array.

The program makes five calls to `random()`. These calls determine how many depots to request up to 25, how much storage to get up to 1 GB, which type of storage to get (i.e. stable, volatile, or both), how long the duration should be up to 10 days and, which location string to use, if any, from an array of 22 strings.

The locations strings used a variety of zip, state, city, country, airport and hostname keywords. Within a category, we used widely spaced values to ensure that it would have to look at all parts of the database. For example, five of the locations include the `zip=` keyword and the values were 01001, 21120, 43201, 65651 and 98983.

Tests were run from the UT campus, from the University of California, San Diego and from Harvard. The test script ran every 10 minutes over a 30-hour period. All calls were made to the primary L-Bone server located at adder.cs.utk.edu, port 6767.

On the UTK campus, as expected, the L-Bone was extremely fast. The mean response time was 0.15 seconds and the median time was 0.08 seconds. Out of 187 tries, only two responses took more than one second (Figure 6).

The wide-area clients were not as fast, but both still averaged under a half second. The UCSD results showed a mean of 0.42 seconds and a median of 0.35 seconds. Its longest time was 2.42 seconds and its shortest was 0.28 seconds. The Harvard test had slightly better times. Harvard's mean was 0.38 and its median was 0.29. The longest response measured at Harvard took 5.21 seconds and the shortest was 0.23 seconds.

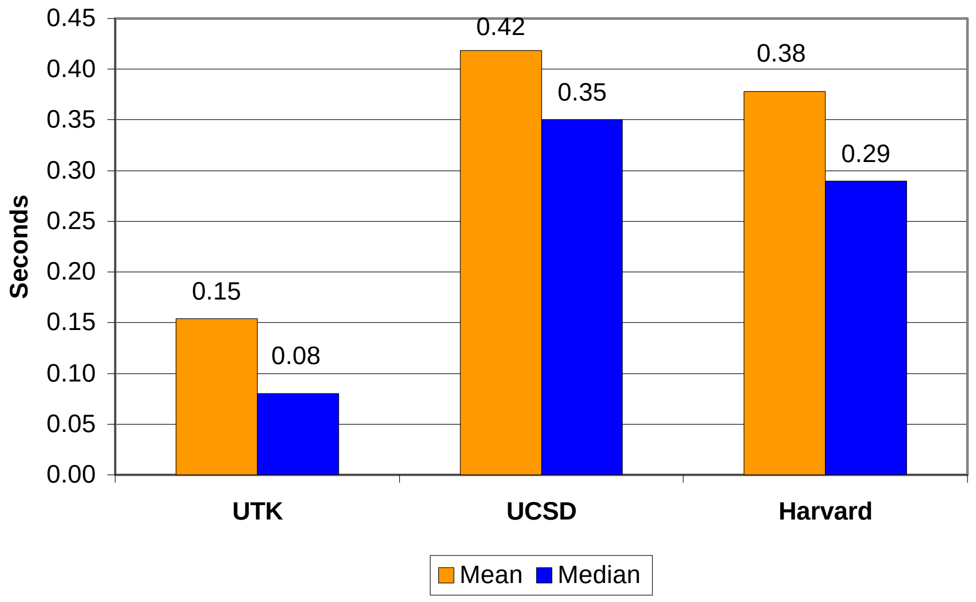


Figure 6: L-Bone response time

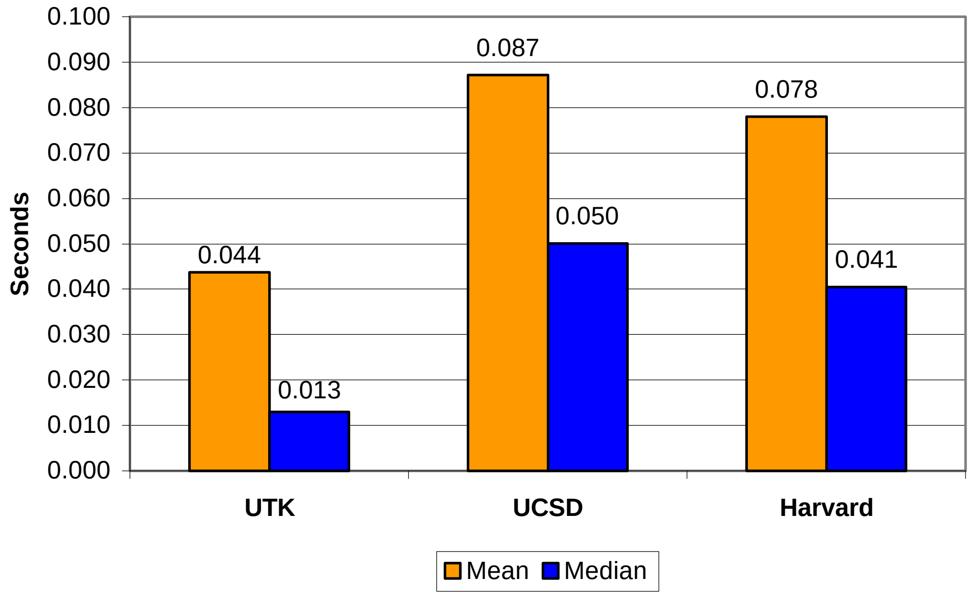


Figure 7: L-Bone response time per depot returned

Because the return RPC call uses 266 bytes per depot, it takes longer to return more depots. When we looked at time per depot returned in the wide-area, the numbers were nearly identical. This seems to indicate that the time to return a large number of depots should scale linearly with the number of depots returned. The median was 0.05 seconds per depot at UCSD and 0.04 seconds per depot at Harvard (Figure 7).

3. Logistical Tools

The Logistical Tools are a set of command line tools and the underlying C API that allow a user to easily find IBP depots using the L-Bone, store a file into the depots, and then store the file metadata, including the IBP capability keys, into a XML file using the exNodes Library. This paper describes the first version of these tools and the API.

These tools make use of the IBP client library, the NWS client library, the L-Bone client library, and the exNode library. Alex Bassi and Yong Zheng developed the exNode library. It provides the basic abilities to create an exNode, add certain metadata about the exNode, create a “segment”, which is a holder for an IBP capability and certain metadata, and add a segment to the exNode. The exNode library also provides the XML serialize and de-serialize routines.

Going back to the “Network Storage Stack”, the Logistical Tools are the layer on top of the L-Bone and exNode Library and provide a set of user tools that combine features of both.

3.1 Design

3.1.1 Goals

The primary motivation for this version is to provide a set of tools for the user that automates the storing and retrieving of files using the exNode data structure and its XML serialization. The tools should expose details of the underlying layers as necessary but automate routine matters as much as possible. Also, the command line tools should be a wrapper around a C API to allow others to use the tools from other applications.

The tools should provide the following functions:

- Store a file
- Retrieve a file
- Add replicas to a file
- Remove replicas from a file
- Extend the durations of the IBP allocations
- View the file's metadata including allocation metadata

The tools should try to perform “intelligently.” For example, if an exNode has several replicas, the download tool should choose to download from the replica that would provide the fastest download.

3.1.2 Assumptions

As in section 2.1.2, we chose to use C as the development language and UNIX as the development platform. Due to the underlying exNode library's use of the Oracle XML parser, we were further restricted to Solaris and Linux due to the fact that the parser is distributed in binaries and is only available for certain platforms.

For this version of the tools, we are only concerned with storing entire files. Thus, our tools can store a single file as well as an entire directory tree. For this version, we do not consider storing data from memory or storing sub-extents of a file. As for retrieving stored data, we allow downloading a part or the entire file. The retrieved data can be stored to a file or sent to stdout.

With these tools, we allow the user to store the file in a single, complete IBP allocation or in multi-part, fragmented IBP allocations. We place no restrictions on the number of fragments among different replicas and we do not require replicas to be aligned on offsets.

Also, we do not prohibit the user from deleting allocations from the exNode, as long as at least one allocation remained. This raises the issue of what a stored file's size means. We use the convention that the file's size is the value of the address of the last addressable byte in the file.

Using this convention, for example, if the original file is 600 bytes and the user deletes all allocations that contain bytes 401 to 600, the exNode then reports the file's “size” as 400. If the user then deletes all allocations that contain bytes 0 to 200, the exNode will still report the size (largest addressable byte) as 400 even though only 200 bytes were accessible (Figure 8).

This version of the underlying exNode library requires that all stored data is stored in IBP depots. The Logistical Tools were built with this restriction in mind. Future versions will allow for other storage types.

Since this was a proof of concept version, we did not focus on multi-threading the storing or retrieving of files. Also, there were many policy options available when we implemented these tools. Since we faced a deadline to have working tools ready for the SuperComputing 2001 conference, we did not try to find the “best” policy. Instead, we chose “reasonable” policies in order to get the tools working. We left the question of finding the “best” policy for future research.

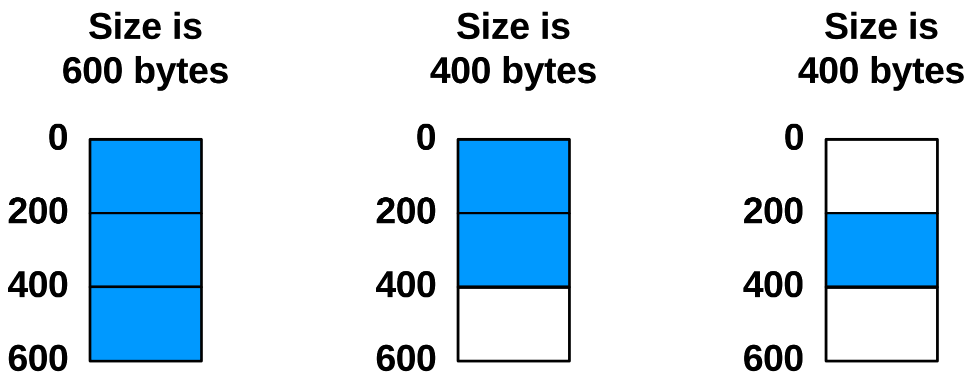


Figure 8: exNode size after trimming

3.1.3 Function Design

There are six functions that allow users to store data to exNodes, to retrieve data from exNodes and to manage exNodes.

3.1.3.1 Upload

The Upload function fulfills two duties: the creation of the exNode and storing data into the exNode. It allows the user to specify the file to be stored, to find depots using the L-Bone client, and to specify how many copies to create and whether the copies are single, complete copies or broken into multiple fragments.

3.1.3.2 Download

The Download function is the complement to Upload. It allows the user to retrieve data previously stored in an exNode. If multiple copies are available, it tries to choose the “best” depot to use to get the data. The tools allow the user to specify the proximity metric. It allows the user to download the entire file or any portion of it.

3.1.3.3 Augment

The Augment function allows the user to add more replicas to an existing exNode. It does not, however, add new data (extend the logical file's length, for example) to an exNode. The same options available for Upload are available for augment (multiple copies, fragments, select depots via the L-Bone).

3.1.3.4 Trim

The Trim function is the complement of the Augment function. It allows the user to remove IBP allocations from the exNode. The user is able to trim stale or expired fragments as well as to be able to specify fragments by number. Additionally, the user may delete the underlying IBP byte-array.

3.1.3.5 Refresh

Since the underlying storage relies on IBP allocations, which are time-limited, the Refresh function allows the user to extend the duration of the IBP allocations. The user may specify the number of days to add or subtract in addition to setting an absolute time.

3.1.3.6 Stat/ls

Using the C API, the Stat function provides a data structure with the exNode’s metadata and IBP capabilities. The Ls function displays the exNode’s metadata and the IBP capabilities on stdout like the UNIX ls command.

3.2 Implementation

3.2.1 C API

3.2.1.1 xnd_upload()

The `xnd_upload()` call creates an exNode, stores a file into IBP depots and returns a pointer to an exNode.

The synopsis is:

```
LPEXNODE xnd_upload ( char *file_target,
                      char *lbone_server,
                      int lbone_port,
                      int storageType,
                      double duration_days,
                      char *location,
                      int copies,
                      int fragments,
                      int buffsize,
                      int blocksize );
```

The first parameter, **file_target**, is the name of the output file where the user wants to store the data. The next two parameters, **lbone_server** and **lbone_port**, tell the tool where to find the L-Bone server in order to find depots.

The **storageType** and **duration_days** parameters expose the choices for underlying IBP storage. **StorageType** determines if the user wants *volatile* or *stable* IBP allocations and **duration_days** sets the minimum acceptable allocation period. The **location** parameter is passed directly to the L-Bone client call and indicates where the user wants to store the file and under what operating conditions the depots should exist. If the parameter is NULL, then return depots with no regard to location (random distribution).

The **copies** arguments tell the function how many replicas to include in the exNode. The **fragments** and **blocksize** arguments determine how each copy will be subdivided. They are mutually exclusive parameters. If the user selects fragments, the function will divide the file into *n* equal size fragments. On the other hand, if the user selects blocksize, the file will be stored into equal size blocks and the last block will almost always be less than the block size unless the file is evenly divisible by the blocksize. In neither case does the function allocate more space than it needs (i.e. no fragmentation). Lastly, the **buffsize** parameter allows the user to tune the amount of buffer space used by the function. This is usually set between 4 and 10 MB.

After the function checks the parameters, the function builds the `LBONE_request` struct based on the parameters. It determines how many depots will be needed, what size each allocation should be, whether the allocations need to be STABLE or VOLATILE, how long the allocations should exist. It passes the location string through to the `lbone_getDepots()` call. It makes the `lbone_getDepots()` call and gets back a list of depots. It allocates the storage for each fragment or block and then uploads the file. For each fragment or blocksize, once the first copy is uploaded, it then uses `IBP_copy()` to move data from the first depot to each additional

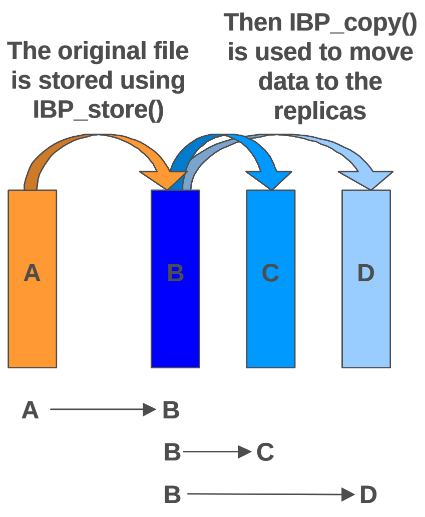

The original file is stored using IBP_store()
Then IBP_copy() is used to move data to the replicas
A
B
C
D
A → B
B → C
B → D

Figure 9: xnd_upload() policy

depot serially. For example, in Figure 9, the upload requires three copies. First, it uses `IBP_store()` to store the file A into IBP allocation B. Next, it uses `IBP_copy()` to copy the data from B to IBP allocation C. Lastly, it copies from B to IBP allocation D.

3.2.1.2 xnd_download()

The `xnd_download()` call takes a pointer to an exNode, downloads the data and sends it to an output file. It returns how many bytes were successfully read.

The synopsis is:

```
int xnd_download ( LPEXNODE pNode,
                   char *output_file,
                   XNDULONG offset,
                   XNDULONG length,
                   int bufsize,
                   int verbose );
```

The first parameter is the pointer to the exNode. The second parameter, **output_file**, is the name of the file where the function should store the data. If it is NULL, it will send the data to stdout. The **offset** and **length** parameters determine which part of the file should be downloaded. The XNDULONG data type is simply an unsigned long long, a 64 bit integer. The **bufsize** parameter sets the size of the internal buffer. The **verbose** parameter sets the level of status messages sent to stderr.

After checking parameters, the function calls `xndstat()` which checks the status of the allocations and returns a `XNDSTAT` struct. Next, it allocates the working buffer. It then opens a temporary file if **output_file** is not NULL or it directs the output to stdout. Lastly, it will start downloading the file.

It then enters a loop that will continue until all the requested bytes have been downloaded. At the current offset, it finds all allocations that contain the offset and sorts them by their proximity measure stored in the `XNDSTAT` struct. To determine how many bytes to download, it looks to see which is the smallest:

- The end of the current allocation,
- The offset of the start of any other allocation, or
- The end of the requested bytes.

Using this algorithm, whenever the set of allocations that contain the current offset changes, it will decide among the new set of allocations which has the best download performance.

In Figure 10, `xnd_download()` must make decisions at byte offsets 0, 900, 1200, 1800, 2400 and 3000. The *Download Choices* column shows one possible combination of fragment sources. Note that once it starts downloading from an IBP allocation, it does not need to continue using that allocation after the next decision point. Also, if during a download, `xnd_download()` has retrieved some of the bytes and the IBP allocation becomes unreachable, `xnd_download()` will use the current offset as a new decision point.

3.2.1.3 xnd_augment()

The `xnd_augment()` function adds more allocations to an existing exNode. The new allocations may be at different locations and may be different sizes than the current allocations. The function also allows the user to augment the entire file or a portion of the file.

The synopsis is:

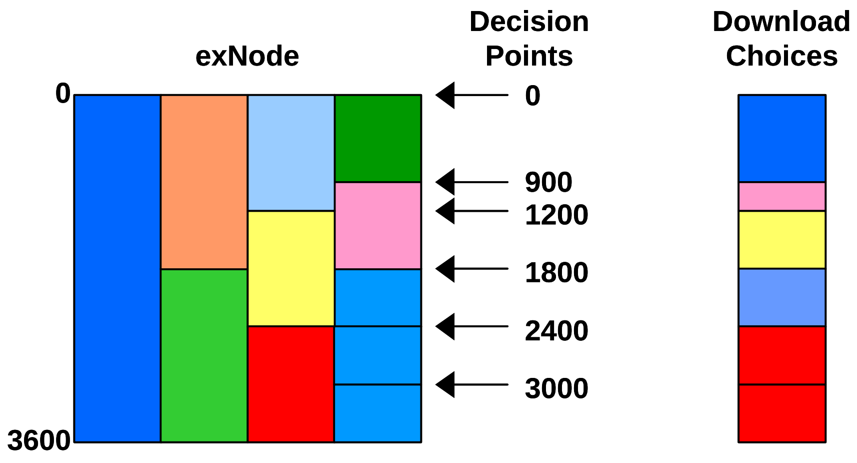


Figure 10: xnd_download() decision points

```
LPEXNODE xnd_augment ( LPEXNODE pNode,
                       char *lbone_server,
                       int lbone_port,
                       int storageType,
                       double duration_days,
                       char * location,
                       int copies,
                       int fragments,
                       int buffsize,
                       int blocksize,
                       XNDULONG offset,
                       XNDULONG length );
```

The parameters are nearly identical to `xnd_upload()`, except that the first parameter is an exNode and not a file name. The other major difference between upload and augment is that augment allows the replica of a portion of the file.

In this version of the tools, `xnd_augment()` tries to use the first available copy to move the file. This is not optimal. Ideally, it would use the copy that has the highest available bandwidth to the target. We intend to add this functionality to the L-Bone and make it available in the next version of the Logistical Tools.

3.2.1.4 xnd_trim()

The `xnd_trim()` function allows the user to specify which allocations to remove from the exNode. The user has the choice of deleting the underlying IBP allocation or simply removing it from this exNode. It returns the number of segments remaining after the trimming.

The synopsis is:

```
int xnd_trim ( LPEXNODE pXnd,
               int *segments,
               int options );
```

The first parameter is a pointer to the current exNode. The segments parameter is an integer array of segment numbers to remove. The current options include:

Non-Destructive	Remove the allocation from the exNode, but do not free it.
Decrement One	Remove the allocation from the exNode and decrement its IBP read reference count by 1.
Decrement All	Remove the allocation from the exNode and decrement its IBP read reference count until it frees the allocation.
Non-Existing	Remove the allocation only if it is unreachable.

This call will modify the existing exNode. If the user wishes to keep the original exNode, he will need to `memcpy()` the exNode before using this function.

By combining `xnd_augment()` and `xnd_trim()`, the user has the ability to route data within the network. The augment adds replicas in the new location and then the trim removes the original replicas, leaving only the data in the new location.

3.2.1.5 xnd_refresh()

The `xnd_refresh()` call tries to modify each IBP allocation's expiration time. The tool allows the user to request additional (or fewer) days and it allows the user to request an absolute expiration time. The modified exNode is returned.

The synopsis is:

```
LPEXNODE xnd_refresh ( LPEXNODE pNode,
                       time_t timeout,
                       unsigned char options )
```

The first parameter is the exNode. The **timeout** value is either the number of seconds to add or subtract from the current expiration times or it is the UNIX time at which the allocations should expire.

The options are:

Extend By	Add (or subtract) the number of seconds to each allocation.
Absolute	Set all allocations to this UNIX time.

This function attempts to modify each allocation separately. If some can be modified and others cannot, it does not rollback the modified allocation expirations to their previous values.

Also, if the user requests *n* days and if the allocation's host depot will only allow another *m* days where $m < n$, the function does not try to use the lesser available value. If *n* is not available, it does not modify that allocation.

3.2.1.6 xndstat()

The `xndstat()` call is used by most of the exNode functions. It returns metadata about the exNode in a standard data structure. This data structure is implemented in the tool layer, not in the underlying exNode library. The underlying tools have an abstracted interface that does not allow us to directly manipulate the data structures. In order to allow these tools to perform certain functions without having to make repeated calls to the same library functions, the `xndstat()` function makes these calls once and stores the data into this struct.

The synopsis is:

```
int xndstat ( LPEXNODE pXnd,
              XNDSTAT * buf,
              int options );
```

The XNDSTAT data structure is:

```
typedef struct xndstat {
      char        name[MAXNAMELENGTH];
      XNDULONG    size;
      XNDULONG    curSize;
      int         numSegments;
      XNDSEGMENT  **segments;
} XNDSTAT;
```

This structure simply is the stored file's name, its original size, its current size, how many allocations the exNode has and a pointer to an array of segments (allocations and their metadata). The `XNDSEGMENT` data structure is more complicated:

```
typedef struct xndsegment {
      int               id;
      char              hostname[MAXNAMELENGTH];
      int               port;
      XNDULONG          offset;
      XNDULONG          size;
      XNDULONG          startPosInStr;
      LPXNDSEGMENT      pSeg;
      IBP_set_of_caps   caps;
      Double            bandwidth;
      Double            latency;
      int               read;
      int               write;
      int               manage;
      int               dataType;
      int               capsType;
      int               reliability;
      time_t            startValidity;
      time_t            endValidity;
      int               readRefCount;
      int               writeRefCount;
} XNDSEGMENT;
```

The **id** is the segment's position in the listing. It is used by the `xnd_trim()` call to specify which allocations to remove. The **hostname** and **port** identify the depot's address where the allocation is kept. The **offset** and **size** describe this allocation's relation to the stored file.

The **startPosInStor** holds the value of the offset with the IBP allocation. Although the underlying exNode library allows us to store multiple pieces of data in the same IBP allocation, in this implementation we always use a new IBP allocation for each segment. Therefore, this value is always zero. The **LPXNDSEGMENT** is a pointer to segment within the real exNode data structure. We keep this for quick removal of the allocation when trimming. The **read**, **write** and **manage** integers will contain a positive value if the IBP allocation contains the corresponding capabilities or a `-1` if they do not.

The **dataType** describes what type of storage is used. These tools only use IBP. The reliability indicates whether the allocation is `STABLE` or `VOLATILE` while the **capsType** describes if the IBP allocation is a byte array, fifo, etc. In these tools, we only use byte arrays.

The **startValidity** currently is the time that the allocation is made. In the future, if IBP or other storage mechanisms allow for reservations, this would indicate when the storage would start. The **endValidity** is the Unix time when the allocation expires.

The **readRefCount** and the **writeRefCount** match the values returned by `IBP_manage()`. Any valid allocation must have a write count of 1 or more to allow writes. By the same measure, any valid IBP allocation must have a read count of 1 or more. As mentioned in the `xnd_trim()` function, if the read count is decremented to zero, the IBP depot frees that allocation.

3.2.2 Command Line Functions

For this version of the tools, we expected that most users would want to use command line tools rather than a C API. For each of the above tools, we created applications that did some error checking and passed parameters to the API for the user. To make the command line tools more user friendly, each looks to see if the user has set up a `.xndrc` file with user preferences. If it is found, the user does not need to enter all options on the command line unless the user wants to override one of the preferences. Some of the command line tools add a little more functionality to the API. Not surprisingly, they use the same names.

3.2.2.1 xnd_upload

The xnd_upload tool performs some basic error checking and then passes the options through to the API.

The usage is:

```
xnd_upload input_file [ options ]
```

The options are:

```
[ -f | -o output_file [ -dir directory ] ]
```

The **-f** option will make it use the input file name and append .xnd when it creates the exNode.

The **-o output_file** option will make it store the exNode using the name “output_file”.

The **-dir** flag will make it store the new exNode in the specified directory.

If neither -f nor -o is specified, it sends the exNode file to stdout.

```
-lbone-host host
-lbone-port port
-l location
```

The **lbone-host**, **lbone-port** and **-l location** options are passed through to the API for the L-Bone client. The location string must be in single or double quotes.

```
-t [ STABLE | VOLATILE ]
-d days
```

The **-t** and **-d** options are passed through to the API for the IBP client.

```
-C copies
[ -F fragments | -BS block_size ]
```

The **–C**, **-F**, and **-BS** options are passed through to the API.

```
-buffsize buffer_size
```

The **-buffsize** option is passed to the API.

3.2.2.2 xnd_download

The xnd_download tool performs some basic error checking, parses the `.xndrc` file and then passes the options to the API. It adds the ability to specify a negative offset that is understood to be an offset from the end of the file.

The usage is:

```
xnd_download xnd_file [ options ]
```

The options are:

```
[ -f | -o output_file [ -dir directory ] ]
```

The **-f** option will make it use the filename stored in the exNode when it downloads the file.

The **-o output_file** will make it use “output_file” for the new file.

The **-dir** will make it store the file in the specified directory.

Again, if neither -f nor -o is specified, it sends the file to stdout.

```
-offset offset
-length length
```

The **-offset** flag allows the user to specify from where in the file that the user wants to start downloading. The user may specify a negative offset to set the offset a specific number of bytes from the end.

The **-length** flag allows the user to specify how much of the file to download after the offset. The default is everything after the offset.

```
-buffsize [ 1 - 100 ]
```

The **-buffsize** flag is converted to MBs and is passed to the API.

3.2.2.3 xnd_augment

The xnd_augment tool adds the ability to augment using a negative offset and by segment number.

The usage is:

```
xnd_augment xnd_file [ options ]
```

The options are:

```
[ -f | -o output_file [ -dir directory ] ]
-lbone-host host
-lbone-port port
-l location
-t [ STABLE | VOLATILE ]
-d days
-C copies
[ -F fragments | -BS block_size ]
-buffsize buffer_size
```

These are the same as in the Upload tool.

```
[ [ -offset offset ] [ -length length ] ] [ -s ]
```

These flags let the user specify what he wants copied. The user may specify offset and/or length OR he may specify a segment. If he only specifies the offset, the tool assumes that the user wants to augment bytes to the end of the file. If the user only specifies the length, it assumes the offset is 0. If the user specifies a segment, it only augments that segment. Like download, the user may specify a negative offset to set the offset a specific number of bytes from the end. The default is to augment the entire file.

3.2.2.4 xnd_trim

The xnd_trim tool passes its options through to the API.

The usage is:

```
xnd_trim xnd_file [ options ]
```

The options are:

```
[ -f | -o output_file [ -dir directory ] ]
```

These are the same as the previous tools.

```
[ -n | -d | -D ]
```

The -n flag (non-destructive) leaves the underlying IBP allocations alone. The –d flag (decrement) will decrement the IBP allocation's read counter by 1 (if the counter was at 1 and is decremented to 0, it frees the allocation). The -D flag (decrement all or nuke) will decrement the allocation's read counter until it frees the allocation. The default is -n.

```
[ -a | -s segment [ segment ...] ] [ -nonexist ]
```

These flags determine if the user wants to trim all (-a) segments or selected segments (-s). If he uses -s to specify segments, he must use white space to separate the segment numbers. The user may modify either the -a or the -s with the -nonexist flag. It checks to see if the segments are available and, if they are not, it trims them.

3.2.2.5 xnd_refresh

The xnd_refresh tool converts a relative time request to UNIX time. It also adds the ability to get the current time in `ctime()` format and in UNIX time, which is seconds since epoch.

The usage is:

```
xnd_refresh xnd_file [ options ]
```

The options are:

```
[ -d days | -a absolute_time ] ]
```

The **-d** option will add (or subtract if it is negative) days to the expiration times of each segment.

The **-a** option will sync all segments to the specified time.

```
[ -what_time_is_it | -w ]
```

The **-w** flag (what time is it) leaves the exNode alone and prints out the time in seconds since epoch as well as formatted using `ctime()`.

3.2.2.6 xnd_ls

The xnd_ls tool will list the exNode's metadata and allocations.

The usage is:

```
xnd_ls [ -b ] xnd_file
```

The **-b** option will output the proximity metric from the user’s proximity file.

xnd_ls prints its output as follows:

```
$ test.xnd: test 100
$ Srwma  0 1 utk.edu:6714 100  0  90.90  Wed Oct 24 18:55:56 2001
```

It repeats for each allocation in the exNode. The output can be read from left to right as follows:

`test.xnd`	exNode file named test.xnd
`test`	name of the file stored in test.xnd
`100`	size in bytes of the file named test

`S`	underneath storage reliability of the segment
	`S - IBP_STABLE`
	`V - IBP_VOLATILE`

`r`	the exNode is readable
`w`	the exNode is writable
`m`	the exNode is managable

`a`	the underneath storage type of the segment
	`a - IBP_BYTEARRAY`
	`f - IBP_FIFO`
	`c - IBP_CIRQ`
	`b - IBP_BUFFER`

`0`	segment id
`1`	read reference count of the segment
`utk.edu`	name of the server hosting the underneath storage
`6714`	port of the server hosting the underneath storage
`100`	size of the segment in bytes
`0`	offset in bytes of the segment in the exNode
`90.90`	connection bandwidth in MBs of the server to the user's host machine (only available with the -b option)
`Wed ... 2001`	the expiration time of the segment

3.3 Tests

3.3.1 Replication for Caching

This test looks at the benefits of having a file replicated locally for improved access performance. It tries to simulate three geographically dispersed users that need to access the same data. Their options are to use an exNode with local replicas, or a centrally stored file available through FTP or a similar service. First, we created the *cached* exNode of a 6.5 MB file with replicas at UTK, UCSD and Harvard. This exNode represents the goal of having replicas near each user. For comparison, we then stored the same file in a separate, *non-cached* exNode at Texas A&M that would represent a non-replicated file that is stored at a centrally located server. Every ten minutes over a period of four days, each “user” downloaded the file from the cached exNode and then from non-cached exNode. Both operations were timed.

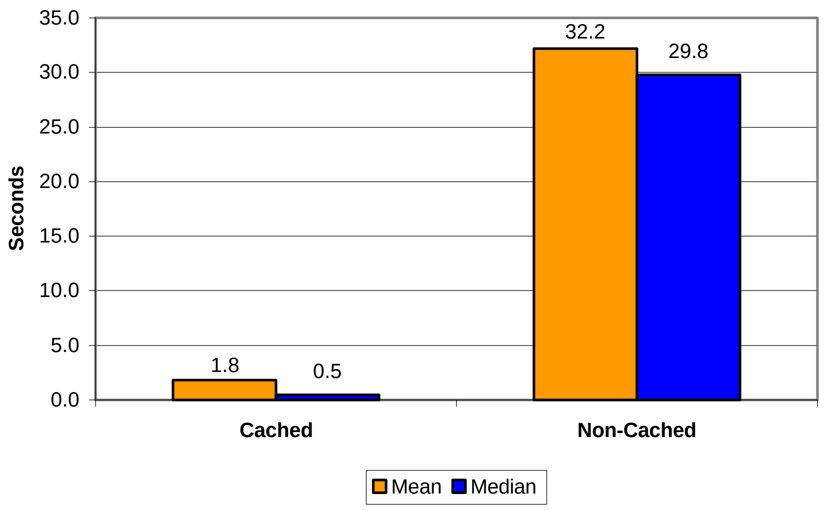


Figure 11: Cached vs. non-cached download time at UTK

The tests at UTK show dramatic differences between the cached exNode and the non-cached exNode. The test at UTK had a mean download time of 1.8 seconds and a median of 0.5 seconds. We could download the non-cached exNode on average in 32.2 seconds with a median download time of 29.8 seconds. Of the 517 cached downloads at UTK, over 250 of them took only 0.4 seconds. The fastest non-cached download took 16.2 seconds (Figure 11).

Similarly, the Harvard results show large benefits for caching data locally. The average time to download from the cached copy was 4.3 seconds with a median of 4.6 seconds. The non-cached downloads, on the other hand, took 44.0 seconds on average with a median of 42.8 seconds. Of the 424 cached downloads at Harvard, over 100 took only 4.6 seconds while the fastest non-cached download took 39.6 seconds (Figure 12).

The results at UCSD again show the benefits of caching but to a lesser extent than the other two test sites. The cached downloads, on average, took 7.3 seconds with a median of 7.1 seconds. The non-cached version only took twice as long rather than the 10 to 16 times longer at Harvard and UTK. The non-cached downloads averaged 13.3 seconds with a median of 12.5 seconds. While the frequency of times overlapped somewhat, the most frequent cached download time was 1.9 seconds (i.e. 48 of the 553 total downloads) while the fastest non-cached time was 10.3 seconds. Interestingly, the most common non-cached time was 10.4 seconds, which occurred 58 times. (Figure 13)

Clearly, using the exNode with local replicas provides a performance advantage versus accessing files from a remote FTP server or other centralized service.

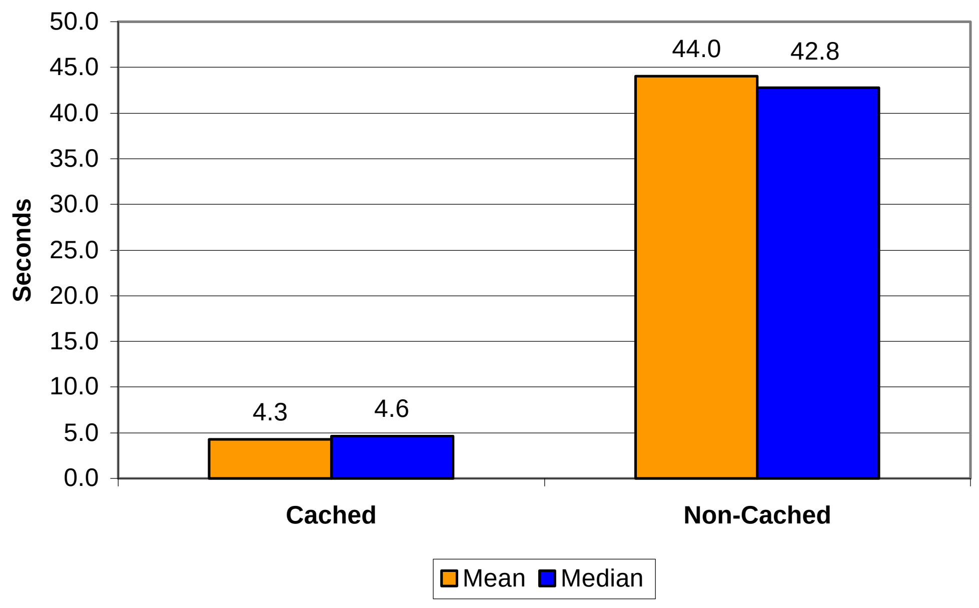


Figure 12: Cached vs non-cached download time at Harvard

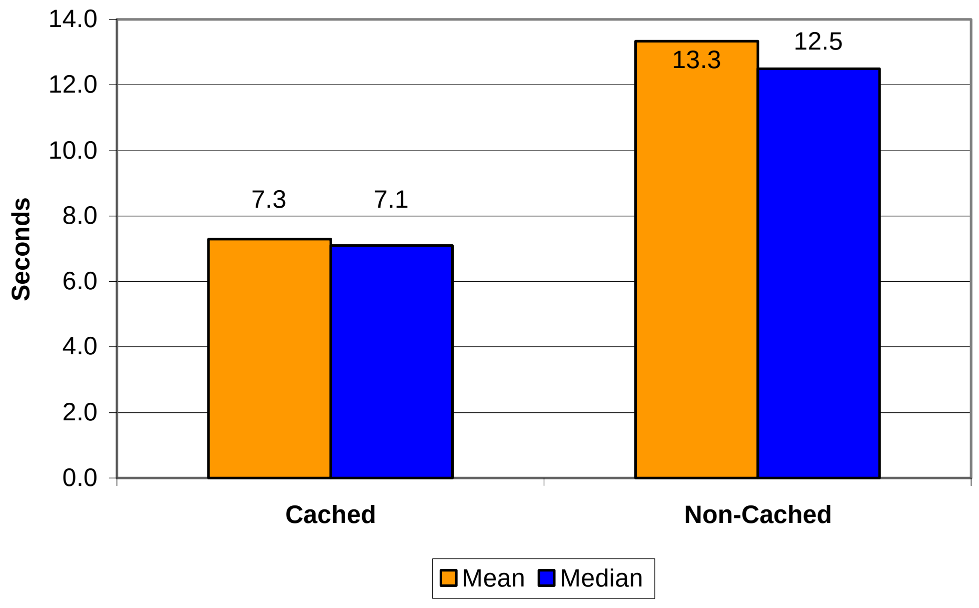


Figure 13: Cached vs. non-cached download time at UCSD

3.3.2 Replication for fault-tolerance

For this test, we stored a 3 MB file into an exNode. The exNode had five replicas spread out over thirteen depots in four states (Figure 14). To test the fault-tolerance of the exNode, every five minutes we checked fragment availability by using xnd_ls and then we downloaded the file using xnd_download. We ran this test on UTK 1 (Knoxville), UCSD 1 (San Diego) and Harvard (Cambridge) over a three-day period.

The fragment availability percentages from xnd_ls are shown in Figures 15, 16 and 17. Since the majority of segments are at Tennessee, we expect to see the highest availability numbers from UTK 1, and this is the case. Similarly, we expect the availability numbers from UCSD to favor the California depots. Interestingly, however, the San Diego test saw higher levels of availability from the Knoxville depots than from the same state Santa Barbara depots.

The most surprising result from Figure 17 is that the availability of the Harvard segment is so low as measured at Harvard. The reason is that the Harvard IBP depot went down for a period of time during the tests even though the machine remained functional. The depot has automatic restart as a **cron** job, but during that time, none of the tests could access the Harvard segment.

During the test, UTK 1 was able to access the 21 fragments on average 94.54% of the time. Out of 860 downloads, UTK 1 had 100% success retrieving the file. Since the exNode had two complete copies on the UTK network, most downloads could retrieve the entire file without leaving the UTK campus. The UCSD test experienced the lowest average availability rate at 90.93% among the 21 fragments. Even with the lower average availability rate, in 857 attempts, this site also experienced a 100% success rate in downloading the file. The Harvard test had a better availability rate than UCSD at 63.42%, and it too had 100% success in downloading the file. Looking at all three tests, we were able to successfully download the file over 2,400 times.

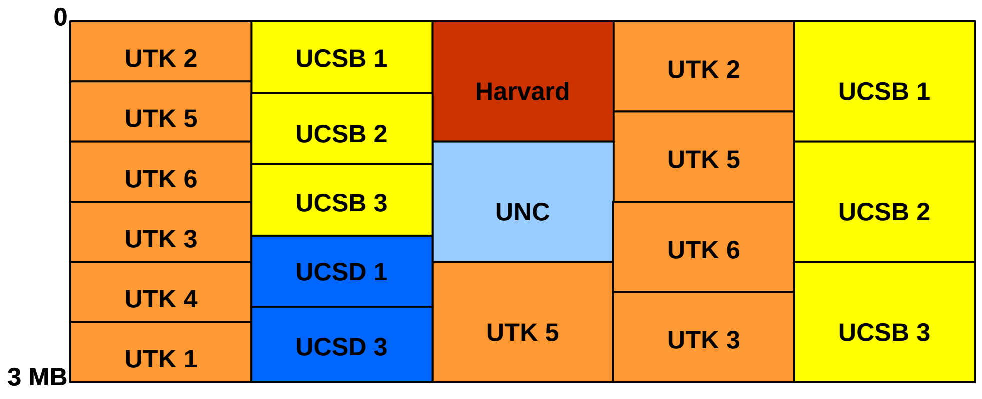


Figure 14: Fault-tolerant test exNode

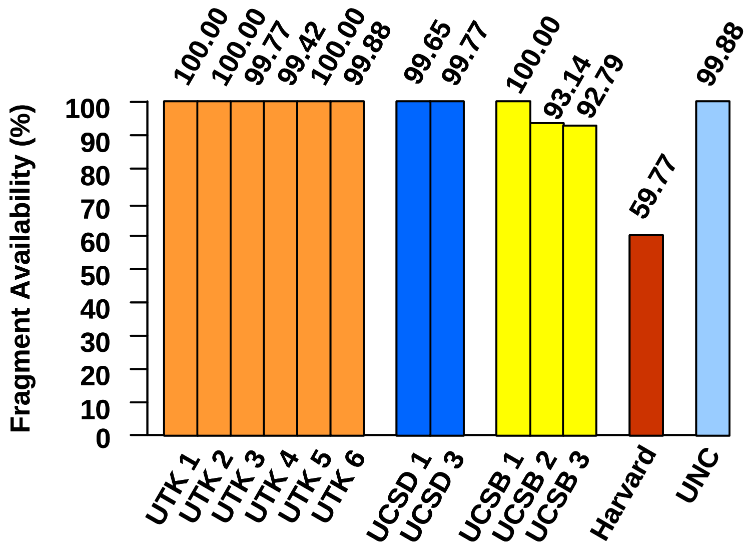


Figure 15: Depot availability measured at UTK

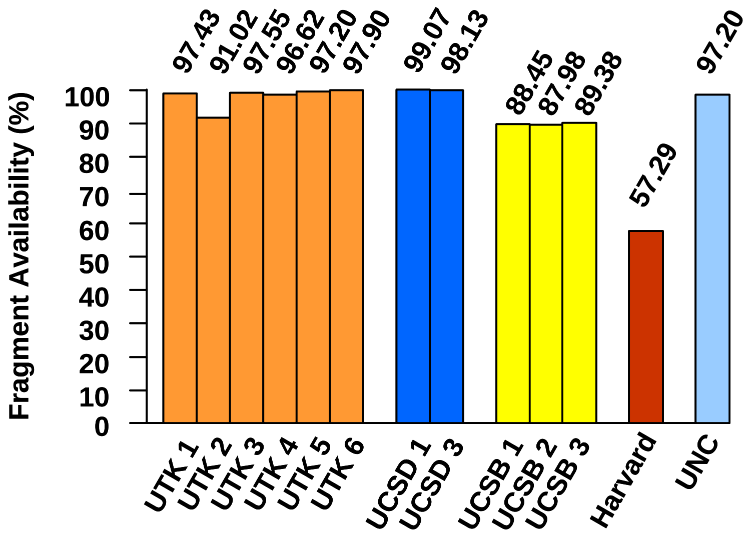


Figure 16: Depot availability measured at UCSD

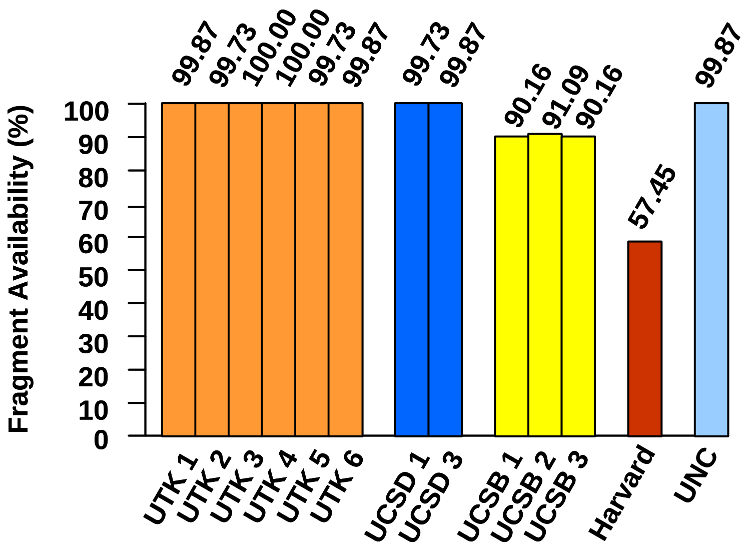


Figure 17: Depot availability measured at Harvard

Next, we wanted to test the fault-tolerance of the exNode when we simulated high levels of unavailability. We used the exNode from the previous test, but we deleted 12 of the 21 byte-arrays from their IBP depots, in order to simulate a high level of resource failure (i.e. machine or network). The resulting exNode (Figure 18) has 33% to 67% of each replica eliminated. Even with the eliminated segments, there are always at least two possible locations available for the download tool to choose, so in the event that one should fail, even if it were the closer of the two, the other is available for a complete recovery.

From UTK 1, we checked the availability and then downloaded the file every two and half minutes over three days. Similar to the last test, we saw individual fragment availability vary from 48.24% to 100%. On average, this test experienced 92.93% segment availability.

Using this restricted exNode, we were able to download the file 1,150 times before we experienced a download failure. Out of the 1,225 total tests, only 75 downloads failed. The first sixth of the file was available only from UCSB 3 and Harvard, which coincidentally had the worst availabilities (93.88% and 48.24%, respectively) of the nine segments. Accordingly, it is reasonable to expect failed downloads of this segment.

This shows that by using simple replication, exNodes can provide a high level of fault-tolerance. Even when the replication was reduced to about two replicas, it still managed to maintain nearly a 94% availability rate. In the future, we plan to add RAID-like encoding to provide high fault-tolerance with lower numbers of replicas.

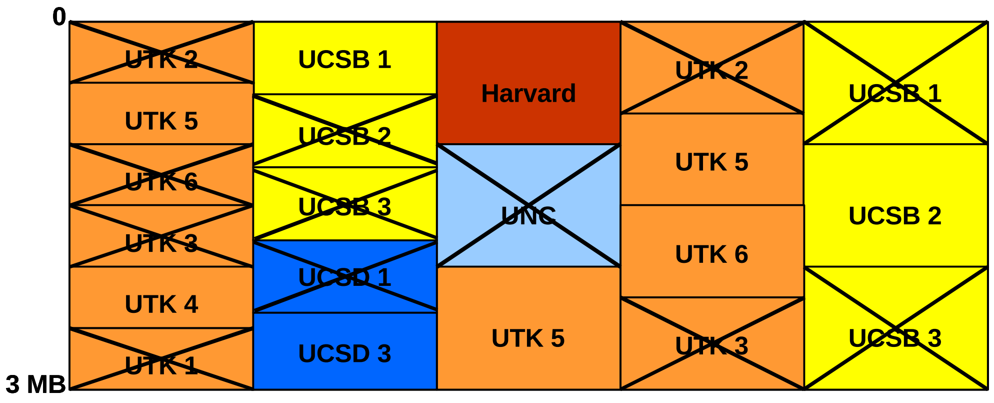


Figure 18: Trimmed exNode with 12 of 21 fragments removed

4. Conclusion

In this paper, I reviewed the current solutions to sharing and storing data across administrative domains in the wide-area network. Current methods including email, FTP, network attached storage and storage area networks all have benefits and shortcomings. The OceanStore project is looking into this problem, but they have not made any tools available for the public to try. The WebFS project allows users to read HTTP resources in addition to reading and writing to its own distributed servers. This project is no longer under development.

I outlined a view of network storage developed by the Logistical Computer and Internetworking Lab (LoCI) called the *Network Storage Stack*. The Network Storage Stack is composed of layers of protocols similar to the network communication stack (TCP/IP stack). The Network Storage Stack layers abstractions of network storage to allow storage resources to be part of the wide-area network in an efficient, flexible, sharable and scalable way. IBP lays the foundation of the stack, which provides access to the underlying storage resources. On top of IBP, the exNode Library provides the ability to aggregate IBP allocations into something like a network file. It also serializes the exNode into a XML text file that can be passed from user to user across administrative domains.

I then reviewed, in detail, the design and implementation of two of the components from the Network Storage Stack, the L-Bone and the Logistical Tools. I showed how the L-Bone provides a directory service of IBP depots. The L-Bone has three major parts: the client API, the RPC protocol and the server. The server uses openldap for data storage, which is tuned for fast performance. Tests show that the L-Bone responds in less than a half second on average in the WAN.

I also showed how the Logistical Tools provide the ability to store and transfer large files. Building on top of the IBP, exNode Library and L-Bone layers of the Network Storage Stack, the Logistical Tools provide an easy to use C API and set of command line tools that automate many tasks associated with creating exNodes, modifying exNodes and retrieving data stored in exNodes. At the same time, the Logistical Tools exposes the functionality of the lower layers (i.e. IBP, exNode and L-Bone) to let the user take advantage of their properties. To demonstrate two of the benefits of using the Logistical Tools, I reviewed tests that showed benefits of replication: caching and fault-tolerance. The results of the caching test showed performance gains of 2 to 16 times when a local replica is available versus using a centralized service like FTP. The fault-tolerance tests showed that using a high number of replicas could provide 100% availability and that even just two replicas could provide over 90% availability.

There is still much work to be done. We have many opportunities to add more functionality to the L-Bone and optimize the Logistical Tools. We plan to use the L-Bone to monitor bandwidth between depots using NWS. This will assist users looking to perform overlay routing using depots. The Logistical Tools will benefit greatly from multi-threading and using different policies for uploading, downloading and augmenting. This work is already under way.

LIST OF REFERENCES

LIST OF REFERENCES

[ASP+02] S. Atchley, S. Soltesz, J. Plank, M. Beck and T. Moore, Fault-Tolerance in the Network Stack. In *Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, April 19, 2002.

[BMP01] M. Beck, T. Moore, and J. S. Plank. Exposed vs. encapsulated approaches to grid service architecture. In *2nd International Workshop on Grid Computing*, Denver, 2001.

[CLG+94] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[Glo02] GridFTP. *The Globus Project.* Available: http://www.globus.org/toolkit/data-management.html, accessed March 29, 2002.

[KBC+00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. Appears in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, November 2000.

[Net02] NetGeo. *NetGeo – The Internet Geogrphic Database.* Available: http://www.caida.org/tools/utilities/netgeo/, accessed April 3, 2002.

[PBB+01] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[RSC98] D. P. Reed, J. H. Saltzer, and D. D. Clark. Comment on active networking and end-to-end arguments. *IEEE Network*, 12(3):69–71, 1998.

[Sac01] David Sacks. Demystifying Storage Networking: DAS, SAN, NAS, NAS Gateways, Fibre Channel, and iSCSI. *IBM Storage Networking*, 3-11, June 2001.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[VEA96] Amin M. Vahdat, Paul C. Eastham, and Thomas E. Anderson. WebFS: A Global Cache Coherent File System. Available: http://www.cs.duke.edu/~vahdat/webfs/webfs.html, accessed April 2, 2002.

[WSH99] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5-6):757-768, 1999.

APPENDIX

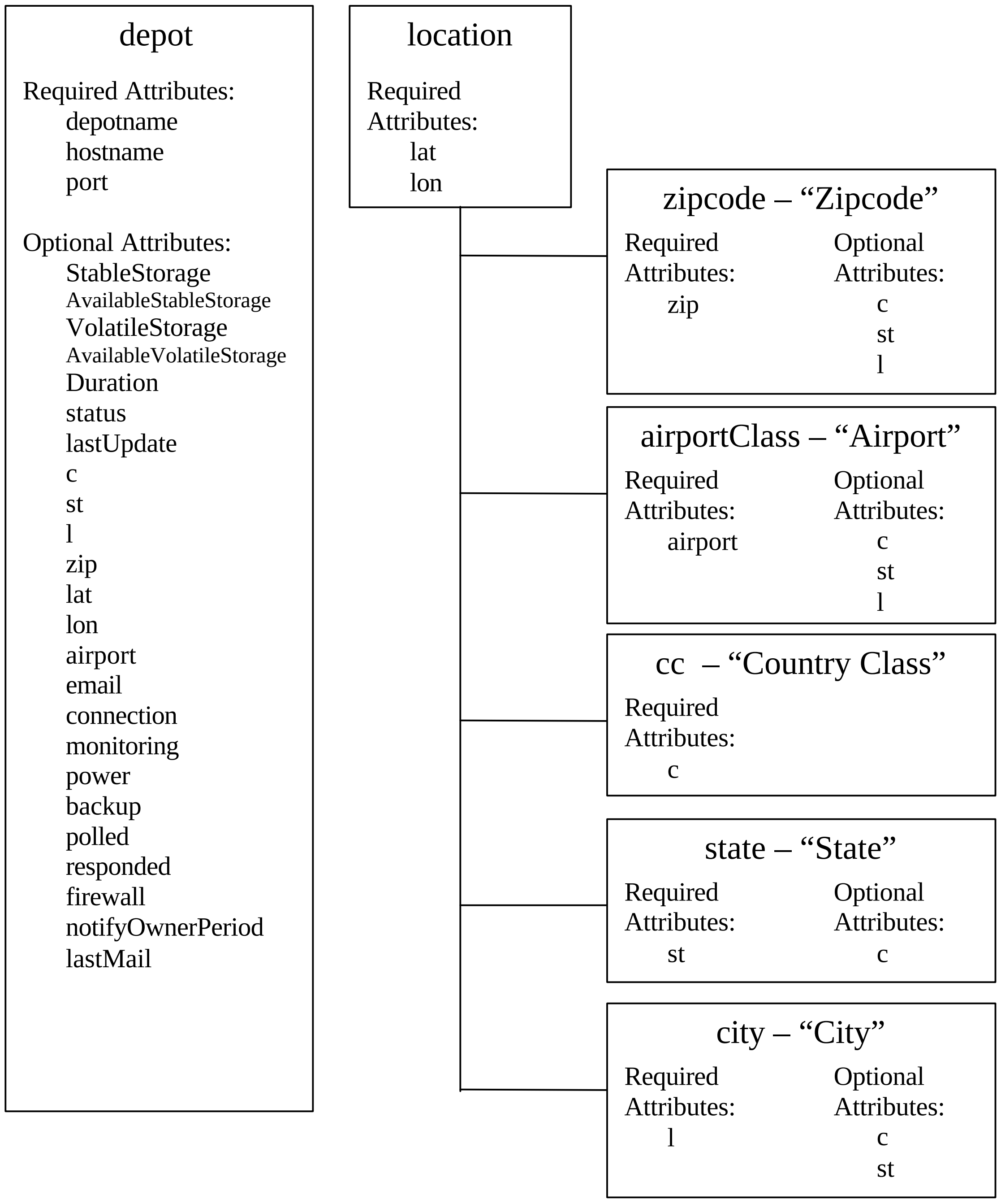
APPENDIX 1: L-BONE SCHEMA

Attribute Types

Attribute Name	Description
depotname	Depot name – may be same as hostname. Must be unique within the L-Bone.
hostname	IBP depot's hostname
port	IBP depot's port
StableStorage	The amount of stable storage the depot is capable of serving
AvailableStableStorage	The amount of stable storage available for new allocations
VolatileStorage	The amount of volatile storage the depot is capable of serving
AvailableVolatileStorage	The amount of volatile storage available for new allocations
duration	The maximum amount of time this depot will allow per allocation
lastUpdate	The last time the depot responded to an `IBP_status()` call from the L-Bone
status	Currently unused
lat	The latitude of the depot
lon	The longitude of the depot
airport	3-letter code of the nearest airport
zip	The five-digit US postal code for the depot (US only)
email	The email address of the depot owner
connection	Describes what type of internet access the depot has
monitoring	Describes how frequently the machine is monitored
power	Describes the backup power capability offered to the depot
backup	Describes the data backup policy for the depot
polled	How many times the depot has been polled by the L-Bone
responded	How many times the depot responded to L-Bone polls
firewall	Shows whether the depot is behind a firewall
notifyOwnerPeriod	Describes what frequency the L-Bone should alert the owner if the depot becomes un responsive
lastMail	Stored the time when the L-Bone last emailed the owner that the depot was unresponsive

APPENDIX 2: L-BONE SCHEMA

Object Classes



APPENDIX 3: SAMPLE XNDRC FILE

```
#  This file contains defaults for the ExNode library. These defaults can be overriden by specifying parameters on the
#  command line of each tool. Any line starting with a # is ignored. To uncomment a line, delete  the #.

#  Schema file and Oracle parser library.
#
#  Use SCHEMA_FILE to specify the absolute path for exNode.xsd. The default for SCHEMA_FILE is to look in the
#  current directory.
#
#  Use PARSER_DIR to set the env variable ORA_NLS33 to the absolute path to nlsdata_linux or nlsdata_sun. There is
#  no default for PARSER_DIR. If it is not specified here, the environmental variable ORA_NLS33 must already be set
#  before using any tools.
#
SCHEMA_FILE                 /neon/homes/atchley/projects/eXnode/exNode.xsd
PARSER_DIR                  /neon/homes/atchley/projects/eXnode/xnd/nlsdata_linux

#  LBone information.
#
#  The LBONE_SERVER and LBONE_PORT are required.
#
#  See loci.cs.utk.edu/lbone/lbone_api.html for details on how to specify location options. Location may be left unused and
#  will default to random depots.
#
#  Set DURATION_DAYS to be the number of days that you want the new storage to exist when you use xnd_upload,
#  xnd_augment and xnd_download. You may specify partial days. The default is 5 day.
#
#  The lbone uses the PROXIMITY_FILE to determine which depots to are best for downloading. If no file is specified, the
#  exnode tools will look for proximity.txt in the current directory. If it is not there, then no proximity resolution is done and
#  downloads will be from the first available segment.
#
LBONE_SERVER                adder.cs.utk.edu
LBONE_PORT                  6767
LOCATION                    zip= 37996
DURATION_DAYS               5.0
PROXIMITY_FILE              /neon/homes/atchley/projects/eXnode/src/proximity.txt

#  IBP information. STORAGE_TYPE may be either STABLE or VOLATILE and it defaults to STABLE. See the IBP
#  website for details.
#
STORAGE_TYPE                STABLE

#  Error message output. Set VERBOSE to 0 for error messages only. Set VERBOSE to 1 for basic status messages. Set
#  VERBOSE to 3 for detailed status and error messages. The default is 0.
#
VERBOSE                     0

#  Output directory. Specify a directory where you would like to store your exnode files. The default is the current
#  directory.
#
XND_DIR                     exnodes

#  Buffer size. Upload and download require a buffer as they move data to and from depots. You can specify values using
#  K for kilobytes or M for megabytes. Make sure there is a space between the number and the K or M. Currently, the
#  tools will accept a value between 1 K and 100 M. The default buffer size is 4 M.
#
  BUFFER_SIZE               4 M

#  Use FRAGMENTS_PER_FILE to break a file into a specific number of pieces. Use FRAGMENT_SIZE to store the file
#  in blocks of this size. You may specify either FRAGMENTS_PER_FILE or FRAGMENT_SIZE, but not both. If you set
#  both, it will use FRAGMENTS_PER_FILE. The default  is FRAGMENTS_PER_FILE = 1. You may use K for kilobytes or
#  M for megabytes. You must leave a space between the number and K or M.
#
#  Use COPIES to set how many copies of the file that you want to create. COPIES defaults to 1.
#
  FRAGMENTS_PER_FILE        1
  COPIES                    2
```

VITA

Scott Atchley first attended the University of Tennessee from 1983 to 1987. He received his Bachelor of Science from the College of Business Administration in March 1987 with a major in Marketing. He graduated with honors.

During his senior year, he began worked for the Cobia Boat Company, located in Sanford, Florida, as a salesman. His territory covered seven states in the southeast. After graduation, he continued in this capacity until 1998 when he became the Director of Marketing. He helped oversee the company's expansion into a new plant in Vonore, Tennessee. At that time, he became much more involved in the daily operation of company and was involved in production scheduling, product design and engineering in addition to his marketing duties.

In 1993, Scott became VP of Sales and Marketing at which time the field sales began reporting to him. In 1995, Yamaha Motor Corporation bought Cobia and planned to move it to Panama City, Florida. Scott left the company and joined Eastern Marketing Associates, a manufacturers representative firm that sells components to boat builders and distributors. At EMA, Scott was responsible for a five state region. In 2000, EMA merged with the R. J. de Recat Company and formed the Derema Group. During that year, Scott's territory increased over 70%. In the fall of 2000, Scott entered the University of Tennessee Graduate School in the Computer Science department. He stayed with Derema through January 2001 to bring his replacement up to speed.

During his tenure at UT, he has worked with the Logistical Computing and Inter-networking (LoCI) Lab working on the L-Bone and Logistical Tools. He plans to work for LoCI as a staff researcher after graduation.