

Distributed Modular Data Repository

Mike Morgan
mikmorg@mdmsolutions.org

February 22, 2007

Contents

1	Abstract	4
2	Introduction	4
2.1	DMDR	4
2.2	Applicability	4
2.2.1	Trusted Networks	5
2.2.2	Untrusted Networks	5
2.3	Prototype	5
3	Background	6
3.1	Motivation	6
3.2	File Systems	6
3.3	RAID	6
3.4	UNIX Extended ACL	6
4	DMDR Fundamentals	6
4.1	System Modularity	6
4.2	Centralized Indexer	6
4.3	Storage Servers (SAN)	7
4.4	Meta-Data Management	8
4.5	Logical Volume Management	8
4.5.1	Data Distribution	8
4.5.2	SAN Monitoring	8
4.5.3	Network Mapping / SAN Feedback	8
4.5.4	Client Parameter Handling	9
4.6	Logical File Mapping	9
4.7	Authentication And Security	9
4.8	Client API	10
4.9	Access Control Lists	10
4.10	Extended File Operations	10
4.11	Repository Maintenance	10
5	DMDR Architecture	10
5.1	Index Server	10
5.1.1	Functionality	10
5.1.2	Configuration	11
5.1.3	Crash Recovery	11

5.1.4	Global File Map (GMap)	11
5.1.5	Authentication	11
5.1.6	FMap, GMap and Meta-Data Loss	11
5.1.7	Database Connector	12
5.1.8	Soft File Locking (GLM)	12
5.1.9	Service Address Floating	12
5.1.10	Redundancy Upkeep/LVM SAN Monitoring	12
5.1.11	Moving Files	12
5.2	Storage Servers (SAN)	12
5.2.1	Functionality	12
5.2.2	Node Reentry / Data Checking	12
5.2.3	Reading Files	13
5.2.4	Writing Files	13
5.2.5	Modifying Files	13
5.2.6	Moving Files	13
5.2.7	Crash Recovery	13
5.2.8	Data Storage / File Management	14
5.2.9	Local File Map (LMap)	14
5.2.10	Sibling List	14
5.2.11	File Hard Locks	14
5.3	Client API	14
5.3.1	Functionality	14
5.3.2	Authentication	14
5.3.3	Crash Recovery	14
5.4	Communication Protocols	14
5.4.1	XML-RPC	15
5.4.2	SAN / Client Data Transfer	15
5.4.3	SAN / Indexer Control	15
5.5	LVM Architecture	15
5.5.1	Data Distribution	15
5.5.2	SAN Monitor	15
5.5.3	Network Mapping and SAN Feedback	15
5.5.4	Hybrids	16
6	DMDR Prototype	16
6.1	Configuration	16
6.2	Testing Environment	16
6.3	Modular Development	16
6.4	Indexer Functionality	16
6.4.1	Authentication	16
6.4.2	GMap	17
6.4.3	XML-RPC for Clients	17
6.4.4	UDP Control for SAN	17
6.4.5	Database Controller	17
6.4.6	Certificate Handling	17
6.5	Storage Server Functionality	17
6.5.1	File Table	17
6.5.2	Data Storage	18
6.5.3	Local File Map (LMap)	18
6.5.4	Reading File Content	18

6.5.5	Writing File Content	18
6.6	Access Control List	18
6.7	LVM Modules	18
6.7.1	LVM Skeleton	19
6.7.2	Basic Striping (RAID-0)	19
6.7.3	Basic Mirroring (RAID-1)	19
6.8	FMap Module	19
6.9	Client API	19
6.10	Client Frontend (TUI)	19
6.11	Client Scripting	20
6.12	Control Protocol	20
6.13	DTX Protocol	20
6.14	Security	20
6.14.1	Authentication	20
6.14.2	Encryption	20
6.14.3	Certification	20
6.15	Development Timeline	20
6.15.1	Checkpoint 1: September 27, 2006	20
6.15.2	Checkpoint 2: October 11, 2006	21
6.15.3	Checkpoint 3: November 1, 2006	21
6.15.4	Checkpoint 4: November 29, 2006	21
6.15.5	Checkpoint 5: December 9, 2006	21
6.15.6	Checkpoint 6: January 29, 2007	21

1 Abstract

This paper presents a new method for storing data across multiple untrusted machines, a Distributed Modular Data Repository (DMDR). This idea consists of creating a distributed volume manager, with a network of storage servers on possibly untrusted machines, considered the Storage Area Network (SAN). These SAN machines will together contribute a dynamic section of their physical harddrive space to a repository of data, simultaneously mitigating inter-SAN and client data transfers. The index server, or indexer, acts as a centralized point of authentication, and administration.

What makes this system different than other distributed systems such as Zebra, and Peer-To-Peer software, is the flexibility and possibilities for future development in incorporating various modules which can work together to create a custom, malleable storage medium.

2 Introduction

2.1 DMDR

The Distributed Modular Data Repository is designed as a framework for research and development in the field of network-based storage. This is not restricted to storage methods, however. There are many interesting areas such as security, file indexing, data mining, OS interoperability, wide area file sharing, concurrency, revisioning, and more.

The goal of this specific project, however, is to develop arbitrary storage methods, and be able to handle them efficiently, securely, and optimally. The project assumes a large number of storage devices on multiple hosts. Multiple storage servers may be active on a single host, one per storage device.

The original goal and purpose of DMDR is to utilize the every-day unused space that is so common in today's fervently growing networks. This includes large corporations, and multiple physical locations.

The long term possibilities of this project include secure transfers, storage, and authentication on an untrusted network of machines. The short term goal, however, is to allow multiple users access a common repository of data to use as they see fit.

One example usage of this would be to give a corporation's employees a place to store their personal and communal files, so that they are on the network avail-

able to any client machine. This can help keep production servers free of personal clutter, and gives the employees a chance to share their information easily.

2.2 Applicability

While targeted towards high-speed local-area networks, DMDR is also a plausible alternative for file sharing over wide-area networks and point-to-point connections. However, DMDR is not designed to reduce network traffic, but instead to be as concurrent and secure as possible. Traffic reduction is slightly emphasized in the actual transfer of file contents, however data being secure and authentic may prove this to be a much more improbable task.

DMDR is also designed to work with a dynamic amount of data on each SAN machine. This means that depending on how much space is demanded, the SAN machine will only have used what is required of it. The storage is targeted to files on the operating system's file structure, and not a raw partition. This does not, however, mute any chance of using a raw partition as the data storage device in the future. One would require an application-specific driver, and then modify the storage server's file access module in order to achieve this goal.

The ideal implementation for DMDR is split into three sections; SAN, indexer, and clients. This is a logical representation of the network, and parts may be aggregated into one, two, or more machines.

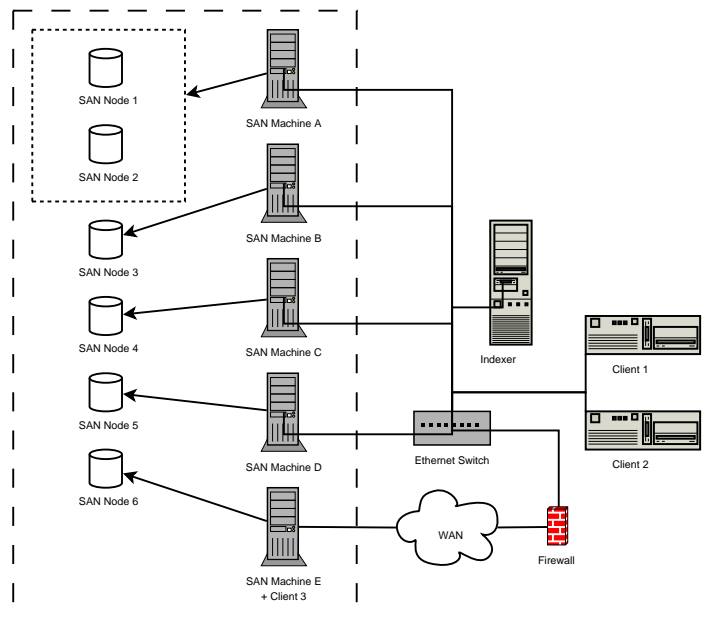


FIGURE 2.1: DMDR Network Overview

2.2.1 Trusted Networks

DMDR allows for multiple desultory implementations of data storage. Such significant uses may include the following:

Extraneous Storage For a large company, buying sufficient harddrive space may be expensive and unnecessary (especially with the number of untrusted harddrive space available). It may also be pertinent to share a repository of common data across a WAN, for collaborating with remote locations. DMDR can be useful in this case, with secure cryptography and SSL layered over the communication and storage nodes. This includes the use of public key encryption, with private keys located on the indexer, clients, and SAN.

Techniques to distribute and communicate over a naturally insecure network, while “injecting” any required trust with encryption, authentication, and certification can be found in the Security section. Further research involving techniques to advert possible damage from SAN nodes being physically stolen should be considered. For this reason, critical private data should not be stored on the SAN, until the fruits of research determine a method for securing this information sufficiently.

2.2.2 Untrusted Networks

It may be wise (or necessary) to use the DMDR without a layer of security. In this case, it may still be useful for many purposes outside of storing private and/or critical data.

Single Computer One may choose to install all three components on the same machine. This is by far the most secure method, and can be valuable in certain aspects. This may include the use of an atypical file mapping module (FMap), recovering a locally-exclusive distributed file, or use of any other further developed layer which may be realized in future work.

Home Network Given that all users on your network are trusted, it is possible to remove secure layers and be productive at a faster transfer rate. This scenario may include one indexer, with each machine on the network acting as both client and storage server. This can help your home evenly distribute, and redundantly store communal data. Personal data may be encrypted and stored on the repository to incite higher

levels of redundancy. Using this method, a group of people may share files for multiple, concurrent access.

Non-Critical Storage For a small company, buying large banks of physical space may become expensive, but necessary for holding certain non-critical information. For this scenario, DMDR can be useful for storing archives of information, given any preliminary encryption beforehand. Another use is for employees storing personal and communal files, and being able to access or share them with any connected client.

Peer-To-Peer The DMDR is designed to work over the WAN, handling changes in SAN dynamic IP over the UDP SAN-indexer protocol, as long as the indexer uses a static IP (or dynamic DNS). Clients should have no problems, as they are completely stateless when idle. Although this may involve legal premissis, the possibility of such an implementation is more than likely.

2.3 Prototype

The prototype developed for CS490 at Purdue University is a simplified proof-of-concept, which does not implement nearly all of the functionality discussed in this paper. The purpose of the prototype is to show the feasibility of creating a module-driven framework for controlling file I/O and upkeep.

The limitations on time have not allowed for the development of enough different modules and operations to successfully prove the concept of the DMDR, but instead have lead to a greater understanding of the fundamentals which would be used in a more stable and organized design.

The timeline for the prototype required a time extension, and was developed using C for the services, and a Java client; split into an API and terminal shell. The basis for simple I/O and authentication were built, and shown to handle the most complex operations required.

The design originally targeted a base of five SAN nodes with two clients, running simultaneously. This was extended to run a total of twenty-four nodes, with a separate indexer node. MySQL servers on the same LAN. See section 6 for further information.

3 Background

3.1 Motivation

The initial motivation for this project came from assisting in managing a corporate network of hundreds of client "dumb terminals" which are complete machines with up to forty, eighty, or more gigabytes of unused space. In an effort to utilize this space efficiently, DMDR was designed to saturate the extraneous space, and distribute files in a centralized fashion for archival purposes. It was devised so that it would be most efficient to store archives, as using the clients for transfers causes significant network overhead, to increase and sustain redundancy.

This eventually led to a general software framework designed for distributed, modular file storage solutions, with the capability of specifying distribution methods on a per-file basis. This became DMDR, as it is designed in this document.

3.2 File Systems

Many approaches to this solution were considered, including one based on a virtual file system layer for client machines. Although this was eventually phased out for design simplicity and cross-compatibility, it had a large impact on the overall design.

Research into file system construction realized the basic structure of the DMDR system, such as a central file index hard-locking for I/O, ACLs, meta-data, GMap, LMap, FMap, which all derive from counterparts in a VFS. However, these components are designed to be extendable and modular.

The standard directory-tree organization is mapped into DMDR as the FMap, which is not implemented in the prototype, leading to a required pre-existing knowledge of file-IDs on the client.

The LMap, which is more closely derived from a RAID management system, distributes chunks based on reconstitution algorithms defined by the DDA. This is related to the superblock in today's linear file systems.

3.3 RAID

Redundant arrays of inexpensive disks are a very important basis for DMDR. Designed to be implemented by LVM modules, these control data distribution, just as RAID management systems do on local and network drive systems.

This was an initial motivator, as it was found as a general requirement for efficiently and consistently placing file data on the network.

3.4 UNIX Extended ACL

Access control is a very important aspect of DMDR, and is implemented very simply in the indexer as a list of numeric user IDs. More complex schemes are possible to handle access with any style of grouping with an ACL module. Access is identified by a unique number, which may be mapped by a service-specific module as either a user or group, and may be then verified by an API call. ACL module associations can be implemented using read-only meta-data fields in each file.

4 DMDR Fundamentals

4.1 System Modularity

The DMDR consists of three applications, all of which are to be designed modularly so that further development and addendums can be included and interchanged, using unique version identifications. In order to support fail-proof reverse compatibility with deprecated modules, multiple versions of a single module may be installed simultaneously, defaulting to the newer version. Files which refer to an older version will still use the deprecated module, unless the module chooses to control deprecated module UUIDs as well. The system is thus designed to handle all tasks as modularly as possible.

The modules are extensively explained in the Architecture section. The following is a list of the modules which are to be dynamically linked and inserted by the user. Note that many modules may be required for each new functionality, such as the LVM.

4.2 Centralized Indexer

The biggest authority in the DMDR's implementation is the indexer. This service handles all of the authentication, security, distribution, management, administration, and centralized data. This is the largest bottleneck and possible security risk in the system, and therefore deserves the most attention.

The indexer, as the most complex section of the system, has the most modules associated with it. Every computer on the network must know its location, and be able to communicate with it at a sufficient rate.

Table 1: DMDR Modules

Name	Location	Description
DDA	SAN	Part of an LVM, controls data distribution.
IODF	SAN/DDA	Part of the DDA, creates an I/O Filter.
SAM	Indexer	Part of an LVM, administers and manages the repository.
NMSF	Indexer	Part of an LVM, assesses environment data and analyses the network.
FMap	Indexer	Creates a mapping to file objects on the indexer.
MDH	Indexer	Extends Meta-Data handling for more verbose information on files.
ACL	Indexer	Modifies file access control for file objects.
Auth	Indexer	Client authentication module (suggest PAM, Kerberos implementations).
Client	Client	Custom client frontends are easy to create with client API.
IDB	Indexer	Custom storage of inodes, meta-data, etc. easily modifiable.
RPC Hooks	Indexer/SAN	Create custom RPC receivers for extended functionality.
FOH	Indexer	Custom file operations for layering new functionality on DMDR.

The data stored on the indexer is also critical to reconstructing the files on the SAN. Therefore, the indexer should have precautions such as offsite redundancy.

Future work may include the ability for multiple indexers, which collaborate global data with each other for faster client access, and SAN upkeep. This is a simple figure of how the indexer works:

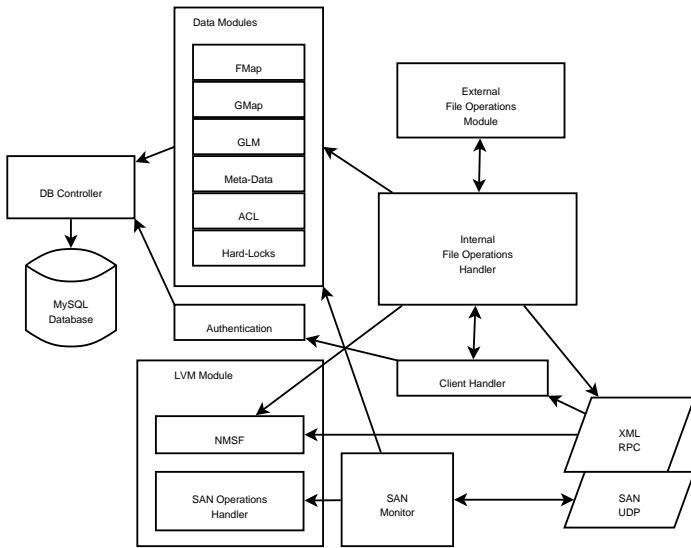


FIGURE 4.1: Indexer Design Overview

many storage devices. These storage devices, however, do not necessarily mean a new machine. Each device is a harddrive, USB pen, or whatever the SAN machine owner decides to use as the medium. The only restriction is that the medium is non-volatile, and should be expected to stay as a component of the SAN for an extended period of time; preferably its entire life span. The following is a figure of the internal operation for a storage server:

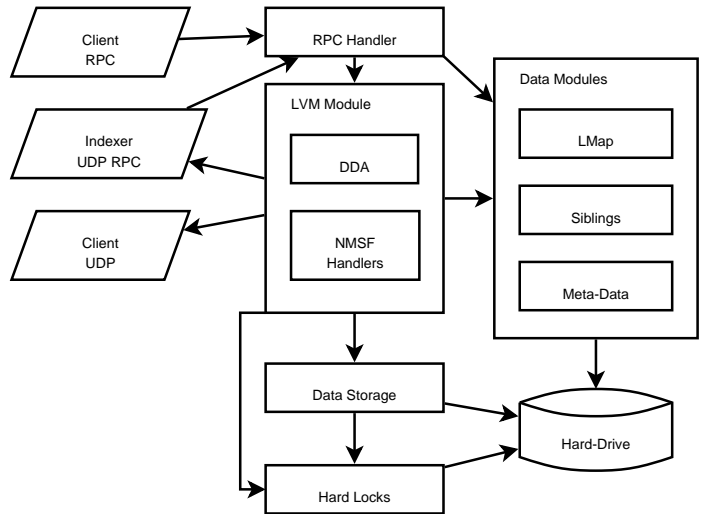


FIGURE 4.2: Storage Server Design Overview

4.3 Storage Servers (SAN)

The purpose of the DMDR is to store data across the SAN. This section explains the use of storage servers, which run on these machines.

The SAN is made up of nodes, which can be one or

Storage services should be instantiated for each device, meaning that each harddrive will run a server. This helps to simplify the logical-to-physical model which may be used to analyse structure and provide parameters for deterministic LVMs. It also cuts down

on development, and helps multithreading the device accesses.

4.4 Meta-Data Management

The indexer handles Meta-Data using a module, which handles the storage of information which is specifically associated with a file, or an FMap meta-file. Meta-data can be stored on the indexer's database resource, and is planned as such in the prototype. Meta-data must be able to store certain internally used fields, such as ACL data, LVM parameters, and integrity checking values. Note that integrity values are not internally used, other than being sent to the client upon request. However, ACL and LVM data are used by their respective modules.

Meta-Data is also extremely important for any modular layered functionality such as revisioning, extended FMaps, and other custom file attributes. Such attributes may include song and movie authors, titles, etc. Meta-Data can be stored in other methods than the indexer's database resource including XML files, or any other storage that is non-volatile. It should be noted that Meta-Data lookups should be timely. This may be an important factor later on, as the indexer could be bottlenecked by this action. Therefore, it may be wise for the indexer's database resource to access an off-site clustered SQL server, which is dedicated for this purpose. This should, however, not be necessary for any practical application. Local database management should be sufficient.

Warning The meta-data for files is a critical resource, and must be preserved at any cost. Therefore, it is suggested that functionality on the indexer backs up the Meta-Data storage location to an offsite location, which may include the SAN itself. Meta-Data is not the only information on the indexer that is so critical. The GMap and Authentication tables should also be backed up routinely. This is the cause for the suggested use of a centralized data resource, such as the indexer's database.

4.5 Logical Volume Management

The Logical Volume Management, or LVM, describes a group of modules and internal functionality that together operate in order to distribute data, and control data distribution. This section describes how this is done, using the indexer and the SAN nodes.

4.5.1 Data Distribution

This includes data distribution algorithms (the DDA modules), which has associated modules in the LVM. Each LVM is composed of exactly one DDA module, and any other modules which may be used in order to maintain the distributed data, or the process of distribution. The LVM may require modules to be installed on all three applications to run properly.

4.5.2 SAN Monitoring

SAN monitoring is also considered a section of LVM functionality. This module should be written specifically for the DDA being used, as its functionality is specific to the distribution method. The monitor should be a module designed for the indexer. While this may be a bit of a misnomer, its true functionality is to administer any SAN-wide methods for the distribution algorithm. This module handles SAN node failures, SAN node downtime, the permanent removal of a SAN node, and any unexpected SAN node loss (such as a crashed machine).

Functionality for such events exceed the LVM module, and is handled mainly internally by the indexer. The indexer then calls the LVM in order to handle node reconstruction algorithms, and redundancy upkeep. Node reconstruction should update the GMap, Meta-Data, LMap and sibling tables for any nodes which hold any section of the file. SAN machines should not be called in order to have their tables updated, as they may not be available. Therefore, the reentry of a SAN node should handle any local updates.

Distribution algorithms may be as complicated as to require SAN-side file reconstruction (such as server-side compression techniques). In this case, the transaction authority must act as the client for its siblings, make any necessary changes to the data, and pass it back to the client. This is a very network-intensive method, and should be discouraged. However, this possibility still exists, and would be considered an I/O Data Filter, or IODF.

4.5.3 Network Mapping / SAN Feedback

The LVM may also choose to include a network mapping and SAN feedback, or NMSF, module. This is an optional module, that can be used by multiple LVMs. The NMSF calculates the topology of the network, including euclidian distances, arbitrary node weights

such as space available, and more. This should be centralized on the indexer, although secondary RPC hooks in the SAN nodes may also be included as to supply information to the NMSF for data assessment. The purpose of this is to feed it back to an authoritative SAN node, so that it can be used in calculations for determining distribution.

4.5.4 Client Parameter Handling

Unfortunately, the client is required to know something about the LVM in order to provide options to the user for how their data will be managed. This allows a user to specify which LVM to use, and how to use it. When a client selects an LVM for use, it has the option of telling the LVM things such as requested level of minimum redundancy, maximum redundancy, minimum spanning, maximum spanning, approximate or specific locale, and many more. The options available are completely up to the LVM developer, and the client developer.

4.6 Logical File Mapping

The logical file mapper, or FMap, is a modular selective high-level implementation for indexing files and meta-files. The FMap is designed to map an arbitrary identifier with a static GMap file ID. This allows for extensive, possibly subjective identification methods for files. The purpose of this is to allow the creation and use of directory structures, search algorithms, and other unorthodox methods of indexing files. This includes surrogate files such as symbolic and hard links, directories, and file names. Note that the data structures in the indexer identify files not as file names, but numerical IDs.

Once an FMap module is added, it should not be removed. This may cause a disassociation, and therefore ambiguity of inodes. Note that inodes may become ambiguous when using a new FMap to index files, where that file has not been added to the FMap. For this reason, clients should be able to use multiple FMaps.

Meta-file is a term for any files which do not exist in the GMap; in other words, they exist only on the indexer. The purpose of a meta-file is to include surrogate files used by the FMap in order to more easily navigate an implementation of file indexing. These are not required entities; there may exist any number of meta-files, including zero. This is an important concept only for those who wish to implement symbolic

links and directories as existing file objects. This is a related subject in file system inode tables.

Important Note Moving a file by changing its association in a certain FMap will not move the file relative to its position in other FMaps. This could be an asset or not, depending on how the users expect the mappings to act. The clients using a given FMap will also require the ability to use the FMap. This may require a client to support the FMap, especially if it is a GUI.

Examples of FMap modules include the commonly used directory structure in today's operating systems, a searching system such as Google, MP3 ID3 tags, and more. The data used to index the FMap should be stored on the indexer, in the Meta-Data table. In the case that the entire body of a file is required, such as data mining algorithms require, this will only be possible on non-encrypted text files in the repository, and lookups will require RPC calls to the SAN machines. The use of such an algorithm would require a very intense distributed search algorithm, which handles the LVM schemes used (therefore not suggested).

4.7 Authentication And Security

The proposed idea of storing arbitrary data on untrusted machines such as our SAN is a very involved statement. See the section on Security for more information.

The DMDR uses an authentication module to decide how users, groups, and passwords are associated. The client will supply a user name and password when it attempts to authenticate. This information is then passed to the module for processing. The ideal module will use authentication lookups in the indexer database. One possible implementation is a basic UNIX password and group association.

Authentication information must be created by the indexer administrator, by adding information to the database. This can be done with an external program, or with internal RPC handlers. Either way, this should be secure, and is suggested that operations such as this should be done on the indexer itself.

The authentication system is meant to be modular, in order to allow plugging in to other authentication procedures. Such procedures may include PAM, Kerberos, or even IMAP. If this route is taken, however, there should be a local table for associating user and/or group IDs with value types used by the ACL

module. The values should be kept as simple numerics, so that ACL modules can be interchanged.

4.8 Client API

In order to allow clients to be developed quickly, and without having to know too much about the DMDR internals, we will supply an API. The API accepts calls for authentication, reading and writing files, finding files with the FMap, and more extended RPC calls which can be implemented by modules.

The client API handles the writing to, and reading from files on the operating system. What is left for the client is simply frontend work, and interoperability with the FMap and extensions. The following is a simple figure of how a client should be written:

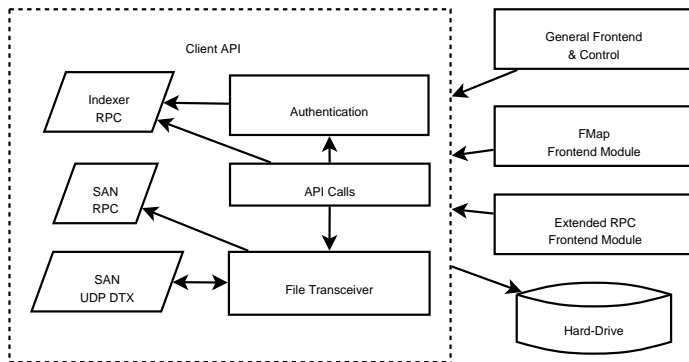


FIGURE 4.3: Client Design Overview

4.9 Access Control Lists

Access Control Lists, or ACLs, are used to secure the contents and meta-data of a file, or meta-file. These are required in order to determine file ownership, such as user and group. Advanced implementations of ACLs may include something related to the true UNIX ACL Kernel module. The ACL module should use Meta-Data fields to store information for each file it protects. Any file with no ACL Meta-Data should be known as not owned, and completely publically available to any authenticated client.

4.10 Extended File Operations

The idea of allowing file operation calls to be modified and added is a very powerful prospect. The DMDR already allows enough to be modified, that calls can now pass through and create “hybrid” functionality. The file operations include reading, writing, deleting, and moving files. These are the internal calls, which can

be overridden. Completely new operations can also be added.

This allows for new operations on files, which may exceed internal functionality. This can be used to layer revisioning control, concurrency handling, and much more. One also has the option of adding fields to file meta-data, which can be used to support these types of changes.

4.11 Repository Maintenance

Once we have data on the repository, there are many aspects of controlling the SAN, and monitoring it, so the data is at the least risk and highest efficiency. The core of this functionality is in the LVM’s SAN monitor. The indexer has a few methods of watching and controlling the SAN. Events are mainly passed from these monitors to the LVM, once an action is required on a file.

The indexer uses a Heartbeat monitor to make sure the SAN nodes are still alive. The purpose of this is to find SAN nodes which have “hard exited,” or crashed. In any other case, a SAN node should be able to instantiate actions. These include SAN node addition, permanent removal, “soft exits,” and general upkeep. More detailed information on maintenance can be found in the Architecture section (5).

5 DMDR Architecture

5.1 Index Server

5.1.1 Functionality

The index data as a centralized point of authority for managing file operations, global meta-data, and access/lock control including security features which must be mutually accessed.

This also holds a map (GMap) of files to the respective containing nodes in the SAN. This list is then used to determine a transfer authority on the SAN, without P2P access for each SAN node by each client. This also ensures consistent data manipulation and updating in the SAN.

The index daemon is the most critical potential bottleneck in the system, and is therefore designed to be efficient in the management of many SAN nodes simultaneously. The system is not fully symmetric as updates must be universalized with a central point for each node to contact after a failure. This is needed also for certain security measures and design simplicity.

5.1.2 Configuration

The indexer is meant to be configured by an external application or RPC calls. In the optimal case, we would develop an external application which modifies the authentication table, and any other preliminary data structures in the database. This should have no concurrency issues, as SQL servers usually handle this by mutual exclusion.

In the case that one uses RPC for configuration, it would require strong security, and a new client application would have to be written. The client API could be extended to handle these RPC cases, however.

5.1.3 Crash Recovery

The indexer is developed so that it remains stateless for the purpose of making such an event very easily handled. The only event which would require extra effort is if the indexer crashed during SAN node reconstruction.

Crashes in the system can be categorized as tolerant when they occur at a non-critical time for that machine specifically. Nodes hold persistent queues for waiting on connections from clients (DTX). At this point, if the client does not connect, residual queue data will result in a memory leak, and degrade overall system performance. This can result in a crash. To solve this, the design should time out after waiting.

Any other potential faults in the system design are encompassed in the failure (unprovoked) in either the indexer, SAN nodes, or client. Of these, all are tolerant except for those which occur while a hardlock is being held.

Hard locks are designed to lock/unlock synchronously, directly before and after each local operation, encompassing the queue for DTX waiting. These locks are used to identify partial repository data which is deprecated, and therefore will be removed from the system, while simultaneously removing the associated hard-lock.

The hard-lock may become zombied when a client or node is being waited on. These zombies, as in the previous case, may be eliminated by using a timeout monitoring thread.

5.1.4 Global File Map (GMap)

The GMap consists of two tables; the GMap file cross-reference, and the SAN node list. The file xref table holds a list of all files, each with a list of SAN nodes

which hold a part of the file. The SAN node list holds all of the nodes as records with unique IDs. This table is then referenced by the file cross-reference table to locate a SAN by its MAC address, IP address, and port number. There should be functions to handle inserting/deleting nodes, and references. Note that inserting a “file” is the same as inserting a reference, as a “file” cannot exist without a relation to a node.

5.1.5 Authentication

Client authentication is the first important step to making this system feasible. The client will first use an authentication request RPC call to the indexer. The indexer then handles this by checking passed credentials with the authentication table; functionality of this is provided by the authentication module. The check returns true or false, if the client passed the correct information.

If authentication succeeds, the indexer then passes the client a key which will be used in transfers. This key will later on be used for encryption purposes, but for now will be simply used to identify the client versus others. All RPC calls other than the authentication request require this key.

Note The encryption key sent from the indexer to the client must also be encrypted, by a public key supplied by the client in the authentication request. This is the only secure method for authentication, and will be implemented only in the secure version of the DMDR.

The certificates for both machines have respective tables with records associating a certificate with a machine name, MAC address, and IP. When a transaction is received, the certificate is looked up in the respective table, and the MAC address, IP, and possibly port values are checked against. This is for security reasons, in order to authorize the machine.

5.1.6 FMap, GMap and Meta-Data Loss

There exist many failure points in the system which must be backed up the design. In the case of indexer data loss, complete recovery may not be possible. Therefore, a client may be embedded in the indexer to backup internal tables. This point of failure could be eliminated with a symmetric design, however, module design will be more complex. To keep this potential failure point to a minimum, it is possible to make a redundant indexer fail-over. This latter solution would

require dynamic reconfiguration for the storage servers to acknowledge the address switch, and possible state deprecation.

If partial data loss/corruption occurs; related file may be restored to the indexer admin by manually entering GMap locations for the file. Metadata is not required for reconstructing files unless a particular LVM makes use of metadata for reconstruction (a very bad idea).

5.1.7 Database Connector

The database connector is a layer of abstraction between API calls in the indexer, and a database manager such as MySQL, ODBC, etc. This is simply a library of API calls included in the indexer source. Note that the database interface allows the administrator to select a redundant design, to ensure that the database content is kept through a localized system crash. The database may also be located on a remote host.

Redundancy It is more than feasible to utilize a database's inherent redundancy implementation to stabilize the DMDR network. This can greatly decrease the chance of data corruption on the SAN.

5.1.8 Soft File Locking (GLM)

The GLM manages soft locking, a virtual locking scheme for clients only. This is not replicated over the SAN, and should only be used as a guide (as file locks are in UNIX). These locks are useful for notifying others that a file is being modified in a client cache.

5.1.9 Service Address Floating

The event of either indexer or SAN node address change (as each machine is identified using an IP assumed globally commensurate), may cause a node from being located and reassimilated.

Each SAN node is preconfigured with an indexer IP, to refrain from requiring network broadcasting and bridging. This problem may be resolved using a non-cached smart DNS client, or a SAN broadcast configuration notice, sent by the indexer at a certain interval. Each SAN node resolves migration by sending its address to the indexer on initialization. This is then correlated with the node UUID in the SAN table until the node detects itself being disconnected.

5.1.10 Redundancy Upkeep/LVM SAN Monitoring

In order to provide a level of redundancy sufficient for a specific file, an LVM module may choose to be alerted on the failure/loss of a node which is currently holding files with that LVM. When the indexer notices a loss of connection to a node, each LVM will be alerted (if registered). The LVM then may compile a list of files which were mapped to that node, and queue any siblings for current redundancy information. If it is possible to reconstruct the file (and the node's timeout is sufficient), the LVM may choose to incorporate one or more new siblings by requesting a sibling to handle the operation for reconstruction.

This authority then contacts the new nodes, and any required nodes for reconstruction. This instantiates SAN DTX hardlocks on each side, and reports a GMap addition of the new nodes.

Note: Prototype will not implement this part, as it would require immense work, insufficient time granted.

Node timeout is particular to each LVM, and extensions may be granted by nodes soft-exiting with a temporary downtime assessment report. If this timeout is insufficient, the LVM should request that the message is re-sent.

5.1.11 Moving Files

Files are only "moved" relative to a FMap association, therefore this is handled by the FMap module developer.

5.2 Storage Servers (SAN)

5.2.1 Functionality

Each storage server is designed to handle all direct data I/O, manipulation, etc. using a LVM module (identified as a header to each operation by the indexer), as an API proxy. This allows the system module to distribute/off-load overhead from the indexer (which only handles authentication, global mapping, and metadata, and also runs SAN consisting checks/reports), to the SAN nodes and clients.

5.2.2 Node Reentry / Data Checking

On the event of a node coming back online, it must first follow initial upkeep and consistency checks.

- Remove residual hardlock entries and associated data.
- Notify indexer of current address.
- Request list of modified files since last change (excluding hardlocked files).
- Determine deprecated files using modified timestamps.
- Request inter-SAN DTX connection from currently GMapped siblings, find authority, request updated data for each deprecated file. (Must be done in chronological order, to keep recursiveness).
- Notify indexer of reinstatement for assimilation.

5.2.3 Reading Files

The indexer selects a node from the GMap associated with the file to read, and sends the certificate and file ID to this node, deeming it as the session authority, after setting a hardlock on the indexer for that file.

The SA then receives the certificate, file ID, and modified timestamp. Each sibling is requested for its GMap and modified timestamp. If the mtime on any of these nodes is out-of-date, an update is started and the read attempts to continue without that node. This “active siblings” list is then sent to the LVM, to determine a list of used siblings for this session. These nodes are then sent back, and contacted to join the DTX session. The DTX address array is then returned through the indexer to the client, as the result of the `fop.read` XML-RPC request.

The client establishes a DTX pipe with each node, and sends a header to identify the request. Each node’s LVM then sends the content using a DTX data transmission API. Each session is closed, and the client then puts the data chunks in order to reconstruct the file. At this point, optional checks may be done on the client, such as a MDS check, based on metadata on the indexer – the trustbox.

5.2.4 Writing Files

Following the indexer receiving the XML request from the client, a new file UID is generated and locked. The indexer then determines a suitable node to be authoritative for the session. The authority queues the local LVM for a list of nodes to use as siblings. Each sibling

is requested to join, and a DTX write authority event is queued, until the client connects to the authority. The authority then splits the data record into dynamically-sized chunks, and sent through the LVM’s DDA to determine its destination. Each chunk is then sent to the respective repository and the write operation is completed by notifying the indexer.

5.2.5 Modifying Files

First, the file is removed from all available nodes. Any nodes which are unavailable will find the mtime out-of-date, and upkeep will handle that scenario. The file is then written, just as in normal writing.

5.2.6 Moving Files

The location of files are respective to the FMap mapping. Each of these files are then identified by a unique numeric identifier, which is determined by the system itself. Mapping these numbers to a new location is therefore a function of the FMap module.

5.2.7 Crash Recovery

In the event of a hard-lock being held across termination of the storage service, the node will remove all data associated with that hard-lock upon restarting. This method of recovery allows the developers of DMDR and any modules within, to ignore these events for the most part. Crash recovery is therefore completely handled, unless redundancy levels are too low.

In the case that the redundancy level for a certain file dips below the file’s LVM’s limit, the recovery algorithm for the LVM will be executed on the Indexer. This case requires a reconstruction phase with sufficient siblings for the file to be active. If this case is not true, the LVM will report that the file is possibly lost. In the case that a client attempts to read the file, the LVM will first attempt to reconstruct the file with the current GMap information. If this cannot be done, the client will be sent an error message, notifying that the file could not be restored.

Example Redundancy Upkeep Algorithm The client gives an authority node a file, with minimum redundancy being 2. The true redundancy, enforced by the DDA, is 2 (across nodes A and B, out of A, B, and C). Node B fails. The indexer notices this by one of its monitors, and requests owners of the file (now only A) to copy the contents to another node. The indexer

sends node A a list of possible siblings, B and C. Node A attempts connection with B and C, of which C responds. Node A therefore copies the file to C. Node A then sends the indexer a summary of the transaction, and the minimum redundancy level is now restored.

5.2.8 Data Storage / File Management

Storage servers (SAN nodes) manage stored data as a repository, self-contained in a system folder. This system folder contains files for node configuration and metadata, hard-locking, file table, and data files. Data files include sibling lists and contents for each file stored on the node. The file table contains a list of file UIDs (as listed on indexer), associated with a file UUID (node-specific), LVM UUID, and modified timestamp. This information provides enough to check the version of the file, the location of the sibling table, and content file (which also stores the LMap as the header), and identify the controlling LVM module.

5.2.9 Local File Map (LMap)

The storage server uses the LMap to find chunks owned by a file. This is implemented by the LVM, and allows the storage server to identify the data contents for re-assembling the file. When writing a file, the LMap is written at the beginning of the data file, which stores the contents. The LMap can be thought as the disk ID information for a disk on a RAID; such as disk 0 in a 3-disk stripe array, should have the information that it is disk 0, and that there are 3 disks in the array. If a chunk exists on the local machine, it will be found in that file's LMap. If it is not, or if it has a redirection entry in the table, then it must exist in a sibling's LMap.

5.2.10 Sibling List

Each file stored partially (or wholly), is given an associated sibling list, on each node it is on. This allows any node to become a DTX authority, and properly distributes actions across the SAN. Siblings are simply listed, without any associated information. This is due to the fact that each sibling may become out-of-date or out-of-service, and the associated information which is replicated may be incorrect. Therefore, control RPC is used to request inter-SAN information.

5.2.11 File Hard Locks

Hard locks are used to ensure an operation is completed before termination so that partial/corrupt data losses do not register in the system as complete. This is done by writing a hard-lock table to the repository, so that updates are synchronous as mutex locking. Hard locks exist on each node, and the indexer. If any machine finds valid residual locks active on any files when booting, the service removes any data owned by that file, including the lock, so as to disacknowledge the existence of the artifact.

5.3 Client API

5.3.1 Functionality

The client API is designed to be portable and reverse compatible. These calls encompass indexer XML-RPC, and SAN DTX in a dynamic library of file operation and system maintenance calls, including connect/client authentication/security handling, file reading and writing, etc.

5.3.2 Authentication

Clients are identified by a certificate generated on each connection. This certificate is sent with each operation. Authentication is done at the connect RPC call, and is the only call that does not require a certificate. The initial design calls for no required authentication, however, additional connect parameters can be added to read a password or other security token.

5.3.3 Crash Recovery

The client is designed so that crashes do not harm the system, by implementing failure-event cleaning on storage servers for DTX, and the indexer for eliminating session-specific data after a specified timeout??. Any other residual client data should be handled above the API.

5.4 Communication Protocols

The three components; indexer, SAN nodes, and the clients; will all use the following three lines of communication. Outlined in the following subsections, RPC is the most widely used for control, along with the UDP indexer control protocol, and the data transfer protocol to transfer file contents.

5.4.1 XML-RPC

Most communication, which does not require large transfers or complete statelessness (such as with SAN nodes connected to the indexer, for control and monitoring only), use the XML-RPC standard. This protocol is used by all three programs. This includes SAN to SAN, Client to SAN, Client to indexer, and SAN to indexer.

Clients can import the XML-RCP standard into almost any language. These packages are widely available by many different open-source groups. Implementations have been found in C/C++ [8], Java [9], TCL [10], and more.

5.4.2 SAN / Client Data Transfer

The storage servers and the clients must transfer data quickly, securely, and with high data assurance. Using UDP, this protocol will be the path that the binary content is transferred over. A response UDP packet should be sent back to the transmitting SAN node, in order to verify (using MD5 or CRC) the integrity of each datagram. The verification may be sent twice on a lossy network, in order to assure the MD5 is sent, and to reduce the number of false negatives. If the datagram response is not sent back for a certain chunk, the SAN node sends the data again, this time with the same chunk ID associated.

The above is continued until each chunk is verified with the response, by the SAN. Transmission should be requested by the SAN, after all chunks that it is responsible for are verified. If a chunk is sent more than a constant number of attempts, the transmission will be terminated by force, and the indexer will be acknowledged.

The client may be a SAN machine, in case of SAN to SAN data transfers. This can be used to replicate data, ensure redundancy, etc. These transfers will be instantiated and maintained by the DDA on the authoritative SAN machine.

5.4.3 SAN / Indexer Control

This line of communication is a bidirectional control path. Using UDP, it ensures statelessness between the SAN nodes and the indexer, allowing nodes to “float” until requested. Each SAN node and the indexer will have a service for this path, so transmissions may be initiated by either party. Integrity will be handled using MD5. Each datagram should arrive in two parts:

the message, and the MD5 sum of the message. The statelessness will help keep traffic to a minimum, and use as few resources as possible, considering that there may be a large number of nodes.

5.5 LVM Architecture

5.5.1 Data Distribution

Each LVM module is composed of two parts; one for the indexer, and one for the storage servers. Data distribution is part of the storage server, and control is sent to the LVM with event hooks, such as beginning write, write chunk, and end of write. These commands are executed in the authority, and inter-LVM RPC over the SAN/Indexer control protocol is accepted. Most LVMs do not require this, as “writers,” which mainly translate to TCP pipes, are initialized by the storage server, to be used to write data directly to the data file on the respective node.

Reading and removing files are also a function of data distribution. This is because they are mirror functions of each other, and should be grouped together.

5.5.2 SAN Monitor

SAN monitoring is part of the Indexer. The Indexer makes a round-robin check of each node currently connected, and verifies that the node is still connected. In the event that a node becomes unreachable, the GMap for that node is parsed. For each file on that node, a sibling node (which is still active) is contacted. This node is given the instruction to check redundancy levels for each file. If the sibling determines that the redundancy is insufficient, it requests a new sibling from the indexer to be mapped to that file. The modified time for that file is updated directly after the LVM is contacted to fill the node. Note that this file is hard-locked over this transaction. If the node fails to write the file (such as the Indexer finds that the node is lost in the next round-robin check, with the hard-lock remaining), the same event is activated again, with each failed node’s list of files.

5.5.3 Network Mapping and SAN Feedback

This is considered an extension of the LVM module on the Indexer, used to attempt a more efficient transaction scheme for each LVM specifically. The network map allows the LVM to determine a logical map of

the SAN, and connected clients, to determine the next write's sibling list.

SAN feedback allows the SAN to request network mapping information from the indexer, for reading files. Note that writing files does not require feedback, as data should be appended to the LVM parameters structure.

5.5.4 Hybrids

LVM hybrids are a theoretical method of creating multiple embedded LVMs, so that algorithms which use multiple layers of distribution are feasible. This is not implemented directly into the DMDR system. This idea is useful for logical analogies such as RAID-0+1, RAID-10, etc.

6 DMDR Prototype

6.1 Configuration

Normally, the indexer would require the authentication table to be configured for clients connecting. In the prototype, we use three sample user accounts, with user names. These will be used for concurrency testing, and ownership association. The users do not require authentication at this point.

The SAN and Client machines must also know where the indexer is located, by host name. This value will be sent as a parameter to the programs, respectively. The clients determine SAN node addresses through the Indexer dynamically.

Each SAN node is also configured with local address information, which is sent to the client. This is optional, as the node may not be able to guess its address due to routers/firewalls. If no address is supplied, the node uses the first non-localhost address found.

6.2 Testing Environment

The prototype will assume a specific environment at first, in order to simplify the first design. The original design called for 5 SAN nodes. However, this environment will include 22 SAN machines, one separate indexer, and two separate client machines. Each SAN machine will run two separate repositories, bringing the SAN to a total of 44 nodes.

The indexer is implemented on the same subnet as the SAN nodes. In theory, SAN nodes may be placed off-site; however, in order to keep the design simple to

test, there are no off-site nodes used. Upon request, an external node may be incorporated easily.

Clients will be run both on and off the subnet. The client is compatible with both Linux and Windows.

6.3 Modular Development

For the purpose of simplicity, only LVM modules will be dynamically linked. Other modules, such as GMap, etc, will be instead object files, statically linked into the binaries. Note that the development will remain modular where it would normally be, unless specified otherwise. However, some modules may not be developed at all. Such parts may include the FMap, more involved LVM modules and IODF counterparts, NMSF analysis modules, extended meta-data handling modules, extraneous authentication and ACL modules, involved client frontends, and extended RPC hooks and file operations handlers.

6.4 Indexer Functionality

The indexer has many sections in it which are a bit vague, as they can be modified in many ways with multiple modular arrangements. For the prototype, the design excludes many modules. The indexer will function simply as a proxy, and GMap distributor, and client identifier.

The prototype's indexer will handle client transactions up to DTX, at which point the nodes will directly connect to the client to do data transmissions.

The indexer and storage servers are both implemented using C. This incorporates low-level speed and efficiency. The downside is that development is much slower in C, and given a more efficient development language, the prototype would have been completed more successfully.

6.4.1 Authentication

Authentication for the clients will be done by an internal table, listing user names and numeric IDs. The main use of this is to simply identify two or three separate accounts which will be used for testing concurrency issues.

The authentication system was decided to be stripped, as passwords are not a fundamental of DMDR. More advanced authentication processes are possible, as they are contained in a static module.

6.4.2 GMap

The GMap implementation follows that of section 5.1.4.

The indexer will implement the GMap as two separate tables. The main table is the file crossreference. This xref table includes one row per reference between a file, and a SAN node. The file IDs should be valid results of any FMap. The machine IDs, which identify the SAN nodes referenced, are keys of the second table. This machine table is a list of machine IDs, along with the MAC address, IP address, and Port number.

The GMap mainly consists of the crossreference table, listing file-to-node associations. The module itself interacts with the database controller (6.4.5).

6.4.3 XML-RPC for Clients

This layer of communication uses the C/C++ XML-RPC library [8]. The RPC server uses the Abyss program, included in the xmlrpc-c library code.

Abyss is a very small HTTP server, with xmlrpc-c hooks already embedded. These hooks look for the /RPC2 URI request. Abyss is embedded in the indexer.

Development of the XML-RPC handler is done as a single module, which has a function to add a function as a handler. This was originally to be used in storage servers also. It was decided that the clients do not require enough communication with the storage servers to validate this.

6.4.4 UDP Control for SAN

The method of transferring control statements between the indexer and storage servers requires a bidirectional, stateless line of communication. Therefore, UDP is used, and a custom communication library was written. Both elements (the indexer and storage servers) use the same library, as to keep the programming simple. There is no reason to develop this twice, as they both have the same functionality.

This library consists of an RPC overlay system. Most of the functionality is the same as the XML-RPC in section 6.4.3.

The UDP control library also includes a latency calculator, used to determine best-suited authority machines, and as a fallback option for calculating best nodes to use in reading/writing files.

This section of the prototype resulted in a large time-sink. It was discovered that TCP should have

been the choice of protocols, simplifying the development. UDP was used due to the fact that connections are made sporadically. This seemed to be an unnecessary step for optimization in the prototype.

6.4.5 Database Controller

The database controller is essential in creating a simple design for the indexer. One central data resource makes the creation of most of the modules very simple, as many SQL systems already handle the storage and concurrency algorithms.

The prototype implements a simple database controller which passes all queries through to a MySQL backbone, using the MyISAM engine. These tables are known to handle concurrency, immediate write-through (so that data is not held in volatile RAM), and a fast reading time. These are ideal for our data structures which we will be handling. For the purpose of the prototype, all non-volatile data will be stored using this controller.

6.4.6 Certificate Handling

Certificates for both clients and storage services are managed by the indexer, internally. Client certificates, which are created by the authentication module (see section 6.14.1), are stored in a table using the database controller (6.4.5), in nonvolatile tables. The certificates for SAN nodes are also stored in this manner, although created by the SAN nodes themselves.

SAN node certificates are created by using a type-1 UUID. The indexer tracks last IP and MAC address information, for reconnection purposes. Note that certificates are random (type-1 UUID), as to secure future transactions.

6.5 Storage Server Functionality

6.5.1 File Table

Each file is requested by a numeric UID, which is the same as the one used by the indexer's GMap (section 6.4.2). For each file, there are the following fields:

Last Modified The table requires a timestamp, which will be used to assure the contents are not expired.

On Disk Reference This column locates the file contents. The prototype will use filenames on the local harddrive; one local file per record.

Chunk Size This is used to calculate offsets in data storage. The chunk size is also be used to send the client data in legitimate chunks.

6.5.2 Data Storage

When data is to be written or read from disk, the storage server will use a file-based structure. Each file which is stored on the SAN node, in full or part, will have its own file. However, this file does not consist entirely of the contents of that file. The LMap implementation (6.5.3) will also be stored in this file.

The file is layed out such that the LMap data is stored in the very front, and directly after the LMap begins the content. The content is stored in chunks, of which size is indicated by the previously-used file table.

The LMap will consist of two 32-bit values, one for a chunk identifier, and the other for the local offset. This means that every chunk stored on the machine will require an extra 8 bytes for location. One should choose a relatively high chunk-size so the LMap is kept small.

The first value in the file is a data offset value. This is the offset to the first byte of the first chunk in the file. From there, chunks can be found by multiplying the chunk's offset value by the chunk size, and adding it to the data offset.

Large Files The maximum file size is 2^{64} , but this is much larger than most operating systems can handle. The prototype will limit file sizes to be, including the LMap size, 2^{32} bytes. The purpose of this is to cut down on development time, as large file support on 32-bit systems can take a significant amount of time to implement, and will not work well with other operating systems, unless a library is developed to cross-compile.

6.5.3 Local File Map (LMap)

The LMap, or Local File Map, is a method of locating chunks which comprise a file's contents. The LMap will be stored with each file, at the beginning of the data on disk. This is described in section 6.5.2. The LMap will have an entry for each chunk present on the local copy. If, however, the chunk is to be located on another node, it will not be in this LMap.

6.5.4 Reading File Content

When a SAN authority is requested by a client to send the contents of a file, it first looks the file up in the File Table (6.5.1). The Last Modified field for this file is then checked against the indexer's value, to make sure the contents are up-to-date. If not, the node requests a data update from the indexer.

If the file has a valid timestamp, the On Disk Reference is then looked up, and the data offset value is read from the top of the file. The data offset is used to calculate a page size, which will then by memory mapped into RAM, to load the LMap for referencing. The first two chunks of file content data are also memory mapped. These two data maps are then swapped around, while being processed, as to speed up file reading.

Before any chunks are read, and directed to the client (or possibly simultaneously), the LMap is traversed, and any missing chunks are requested from clients on the sibling list. This is done by first requesting a list of chunks on each sibling. The DDA then delegates which machines should send which chunks back to the client, and requests the action from the siblings used. Note that the DDA may specify that not all siblings are even used at all.

6.5.5 Writing File Content

The SAN authority uses the DDA for all writing operations. The DDA is passed a list of possible siblings, and a list of used siblings is returned. In the call, the DDA writes part of the file locally, and stores a distriptive LMap at the head, and does the same with siblings. This is much easier and less intensive than reading.

6.6 Access Control List

The ACL module developed for the prototype is a statically linked object. This library simply identifies the owner (or writer) of each file, by associating a username. There are no passwords required to enter the system, however, a valid user name must be given.

6.7 LVM Modules

The prototype will require basic LVM modules in order to test the DMDR, as the LVM code is the true heart of the system. Without this functionality, there can be no test cases, other than the very simple framework

working together as a shell. Therefore, I propose the following modules:

6.7.1 LVM Skeleton

The LVM skeleton is the preliminary LVM module, developed while the DMDR system is developed. This implements all of the features which LVMs can handle, and new modules are created by copying this skeleton module, and making modifications. This simplifies the development of new modules, by providing a hands-on commented example to adjust.

The skeleton provides simple, non-redundant striping functionality. This is closely related to the Basic Striping module (6.7.2), but differs in that it does not allow the user to specify a level of redundancy.

6.7.2 Basic Striping (RAID-0)

The first and most basic LVM to be developed will be the striped distribution system. This simply means that data will be separated across multiple machines, with zero redundancy. If one node fails, the file will be lost for the duration of that node being down. This would never be a good method of storage unless redundancy is added. However, this is a good simple proof-of-concept to make SAN machines reconstruct a file.

The striping algorithm is very simple. The distribution module will be placed on the nodes, and no NMSF handlers will be required. There will be a SAN operations handler on the indexer, in order to make sure the stripe still exists for each file. If one node fails in a permanent fashion, the file will be erased from the sibling nodes, and tables will be cleared such that the file does not exist anymore.

The basic striping module should also allow for a level of redundancy. This means that each 'stripe' is mirrored across multiple nodes. This is in effect closer to RAID-0+1 in functionality.

6.7.3 Basic Mirroring (RAID-1)

The second basic LVM to be created is file mirroring. This is the most redundant method of storage. The client will be able to choose how many machines are to store the file, at a minimum. The distribution module on the SAN will copy the contents it receives to that number of other nodes, minus itself.

The SAN operations required by this method is completely different than striping in section 6.7.2, as

there will be redundancy upkeep, and sibling data will never require deletion. Redundancy upkeep will be required in scenarios such as the following example:

6.8 FMap Module

For the first prototype, the FMap module is not to be implemented. If time allows, however, this module is the first to be developed.

The FMap is bypassed, as it is not a critical component of the system. Instead, the client is required to know the GMap file ID, and the indexing lookup phase that the client would normally require is bypassed.

6.9 Client API

The client API is very simple in implementation. This is one of the first things written, as it was required for the first testing phases of the indexer, which was developed first. The client API was written using Java (to make it OS independant), and uses the XML-RPC package at <http://ws.apache.org/xmlrpc> [9].

The authentication client was the first section to be developed. This uses XML-RPC as a client to connect to the indexer. See section 6.14.1 for more information.

Given that the file transceiver, authentication, RPC handlers, and data transfer handlers are complete, the API is written except for the actual interface to the client code. The API includes functions to receive authentication certificate, write a file to the SAN, read a file from the SAN, lock a file on the indexer, read file Meta-Data on the indexer, and possibly few more.

6.10 Client Frontend (TUI)

The client is developed as a packaged API in Java classes. This API is demonstrated by a simple command prompt, with all of the developed functionality implemented as manual commands. This frontend was written to be scriptable, so that standard input could be redirected.

The shell is able to start a connection, hold parameters, and run simple commands with verbose event feedback. The user should use the "connect" command first, and always "disconnect" when completed. Functionality such as FMap handling, etc. is not included, as it is not supported yet on the prototype.

6.11 Client Scripting

In order to do QA, and test the system to make sure it works with many concurrent processes, we use scripts to test the system. These scripts call the client frontend (described in section 6.10), in order to generate events such as downloading, uploading, deleting, and indexer command execution. Full sessions are completely scriptable. Shell commands operate synchronously, so that commands do not clobber each other.

6.12 Control Protocol

The control protocol was developed as specified in section 5.4.2. This implements a fully functional UDP arbitrary path protocol for messaging, RPC, and latency calculations. It is implemented as a common library between the Indexer and storage servers.

6.13 DTX Protocol

The client uses a TCP DTX implementation for transfers between itself and SAN nodes. The transmitting implementation connects to a single node. However, the receiving system must handle multiple simultaneous downloads, and a post-receive reconstruction phase.

The receiver handlers must also associate each chunk received with its position in the linear destination file. The simple client implementation uses an in-memory cache for speed, and development simplicity. However, given that this system is designed for large archival files, a non-volatile cache must be at least theoretically feasible.

The non-volatile cache may be implemented as on-disk queues of chunks, partially sorted when received, and concatenated upon completion. Using this method, twice the memory is required on-disk, in order to assure worst-case reconstruction is possible.

6.14 Security

6.14.1 Authentication

Authentication is a necessary stage in order to test any concurrency stages and issues. For this purpose only, a very simple user based authentication stage is implemented in the indexer. This authentication module uses the indexer's database to store a simple associative table. The architecture mentions the use of secure key encryption for authentication, but the prototype will

not implement this level of security. Keys which are implemented (and required for identification signing) are passed without any obfuscation or encryption.

Once the client has its certificate with which to identify itself in future transactions, it is assumed that the authentication stage is complete. The client should now supply this certificate in any future RPC calls.

6.14.2 Encryption

In order to keep this client simplified, we will not implement any encryption. The purpose of this, which is to prevent MIM attacks, unauthorized clients and storage servers, etc. is out of the picture. For this implementation, we will assume there are no malicious users.

6.14.3 Certification

While certification is normally very important in an untrusted environment, the prototype will assume that no malicious users can access the system. Therefore, there will be absolutely no certification, other than that which identifies a client or SAN node (see section 6.14.1). Certificates are created as UUIDs, based on the client's MAC address, a timestamp, and random data.

6.15 Development Timeline

6.15.1 Checkpoint 1: September 27, 2006

Checkpoint one marked the beginning stages of development. This includes a few fundamental sections of the indexer, and storage server. The XML-RPC system for the indexer was completely finalized. The client handler calls were also implemented, including the certificate system and UUID library calls. The authentication system was created based on MySQL tables, in order to associate certificates with users. The database controller was also begun for this purpose.

Test scripts were used by creating XML-RPC calls with an `xmlrpc-c` utility program, which verified that certificates were created successfully.

In this stage, the client is not be developed at all. Instead, scripts were used to test the indexer, by wrapping the `xmlrpc` executable included in the `xmlrpc-c` program. These wrappers were just simple certificate requests, and exist in the `src/tests/test1.pl` file.

The UDP control protocol was started at this stage, but development is still heavily needed. The SAN con-

troller for the indexer was started, for the round-robin node checking thread.

Changes to the design were made, so that SAN XML-RPC was not required. Instead, requests are handled completely over the DTX and Control protocols.

Actions which are legal at this point include:

- The indexer will boot and initialize properly.
- Clients can connect and authenticate properly while receiving a certificate for identification.
- Internal file operations requested by client return void results but are valid actions.

The first checkpoint was initially the bulk of development, attempting to get all design issues resolved at the very beginning. Unfortunately, the design changed radically along the development of the prototype. This included using LVM modules to do more than initially expected, but slowed down the development progress. Checkpoint one never completed the basic framework for both the Indexer and Storage server.

The Indexer/SAN UDP Control Protocol turned out to be a major project, involving much of the time in development of checkpoints one and two, and partially into three. This protocol was extended to handle RPC capabilities, and is lacking certain aspects which were not seen as necessary in the prototype.

6.15.2 Checkpoint 2: October 11, 2006

In this section, the data transmission protocol between the client and SAN nodes was to be completed. Development on the client application is also begun.

The client program at this stage includes a simple one-operation which connects, attempts to write, and disconnects. No shell is developed.

Checkpoint two was used to finish up what was incomplete in checkpoint one, for the most part. Additions were minimal, and is mostly comprised of work on the control protocol.

6.15.3 Checkpoint 3: November 1, 2006

Checkpoint three became mostly an extension of checkpoint two, developing the control protocol in order to guarantee stability, and functionality. The need for a full RPC implementation was recognized and initially developed.

The storage server gained much work on this stage, to make the indexer and nodes communicate properly. No data transfers were developed, however, many fundamental features in order to make the entire system communicate together were developed.

Checkpoint three included the development of the LVM module importing framework. At first, the capabilities of LVM modules seemed slim, but the issue was resolved with a different style of linking. The LVM framework is tested with a skeleton LVM, called `lvmskel`.

The client was developed in a hard-coded state, only enough for basic testing of the write file operation, which is currently the only one having any significant work. The design of `fop.write` was modified largely, to accommodate required design changes seen in checkpoint one. The write file operation is still under major development at the end of this checkpoint.

6.15.4 Checkpoint 4: November 29, 2006

The beginning of this checkpoint is a realization of a crunched timeline for the project. Decisions involving cutting extra work out are made, and the basic read/write file operations are still in queue. The development of two basic LVMs, striping and mirroring, were to be originally completed at this point. This was not possible, however, due to the fact that the skeleton LVM is still under heavy development.

The write operation is complete at this stage, and the client is now a scriptable terminal. The programs are tested on all platforms which are targeted, and run as expected. The read file operation is under heavy development, as the design for reading algorithms was changed quite radically.

6.15.5 Checkpoint 5: December 9, 2006

This section of development included heavy work on the testing platform, and developing test scripts. Work was done in order to reduce debugging time, and verify that the work at this point is globally executable. Major errors were debugged and resolved in this checkpoint, including many control protocol issues, and redesign of the read protocol along with certain patches.

6.15.6 Checkpoint 6: January 29, 2007

This section concludes the research project as per credit, at Purdue University. The read functionality was completed, including dynamic LVM-to-sibling

command and argument passing. This allows LVM modules to do intelligent and efficient file transfers on reading.

The mirroring LVM was completed with two separate reading methods; one multi-threaded multiple nodes to client, and one single node to client. Redundancy upkeep was not implemented for neither mirroring or striping (the skeleton LVM).

The functionality for restoration was not completed for the research project. Although planned, the time involved in developing this functionality to a constructive point was not feasible. Much of the utility code which would be used to do reconstruction is written, as part of reading and writing data files. The difference

between reconstruction and what is currently available is mainly in the destination of the read/write (towards another node), and the targeted data to read.

The DTX library was not implemented fully for the siblings, as receiving data (for reading) is a complex procedure (see the client's DtxReceiver class). This was not necessary for our purposes, of communicating with the client. This is however a very important requirement for node upkeep.

Restoration, including proper monitoring by the Indexer, would require a very significant codebase to complete to the extent of basic operation. Unfortunately, only slight hints of code towards this idea were completed in the prototype.

References

- [1] Chandramohan A. Thekkath, Timothy Mann, Edward K.Lee, Frangipani: A Scalable Distributed File System, Systems Research Center, Digital Equipment Corporation 130 Lytton Ave, PaloAlto, CA 94301 <http://www.nondot.org/sabre/os/files/FileSystems/Fkangipani%20-%20A%20Scalable%20Distributed%20F>
- [2] David Teigland, Ken Preslan, Matthew O'Keefe An Overview of The Global File System, <http://www.globalfilesystem.org>
- [3] Steve D. Pate, UNIX Filesystems: Evolution, Design, and Implementation, Indianapolis, Indiana: Wiley Publishing, Inc., 2003.
- [4] Edward K. Lee, Chandramohan A. Thekkath, Petal: Distributed Virtual Disks, Systems Research Center, Digital Equipment Corporation 130 Lytton Ave, PaloAlto, CA 94301 <http://www.thekkath.org/papers/petal.pdf>
- [5] John Henry Hartman, The Zebra Striped Network File System, Univ. of California at Berkeley, 1994. <http://www.cs.arizona.edu/~jhh/papers/thesis.pdf>
- [6] Randolph Y. Wang, Thomas E. Anderson, xFS: A Wide Area Mass Storage File System, Computer Science Division, University of California Berkeley CAj 94720 <http://now.cs.berkeley.edu/Papers2/Postscript/wwos93.ps> (PDF)
- [7] Cygwin: Linux environment for Windows. <http://www.cygwin.com/>
- [8] Sourceforge, XML-RPC C and C++ implementation <http://xmlrpc-c.sourceforge.net>
- [9] Apache, XML-RPC Java implementation <http://ws.apache.org/xmlrpc>
- [10] Sourceforge, XML-RPC TCL implementation <http://sourceforge.net/projects/xmlrpctcl>