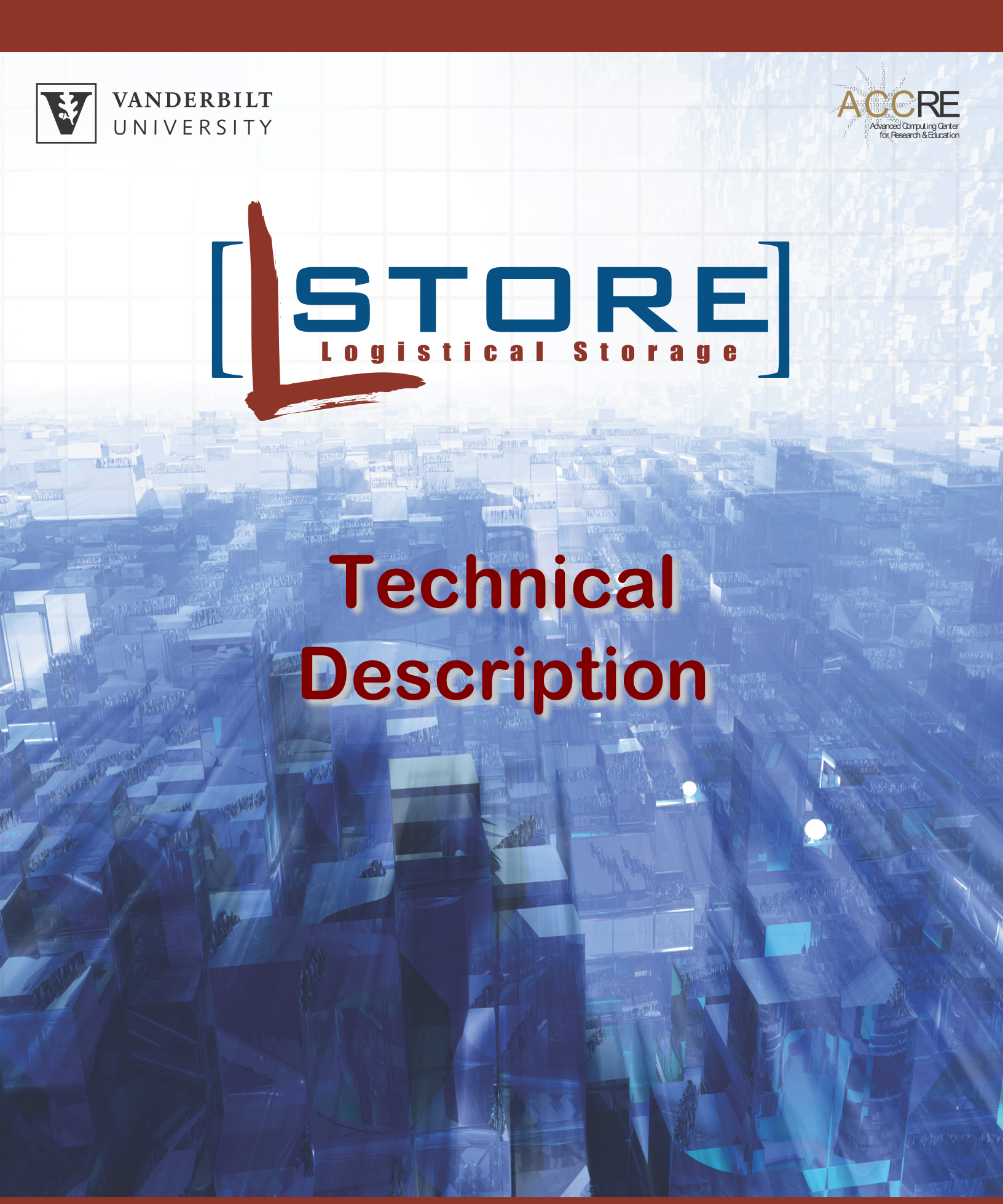
VANDERBILT UNIVERSITY

ACCRE Advanced Computing Center for Research & Education

[LSTORE] Logistical Storage

Technical Description



Enabling Researcher-Driven Innovation and Exploration

L-Store Technical Description

Version 1.0
Last Updated: 11/05/2013

Contents

1. Overview

The purpose of this document is to provide an overview of L-Store and it's underpinnings in Logistical networking and consists of a technical description of the storage hardware and the L-Store virtual file system software.

In this document there are several key terms, which will be frequently used. For the readers convenience these terms are defined in Table 1. L-Store is an integrated suite of storage and networking technologies for creating topologically embedded, topology aware storage nodes, or *depots,* meant to implement a *WAN-enabled (World Wide) network*. This is an important distinction from many other storage methods, which only work on a LAN (i.e. within an institution). L-Store's purpose is to provide a foundation for shared storage infrastructure that dramatically improves the ability of data intensive communities to collaborate in the wide area, especially by bridging the technical and administrative obstacles between national and international sources of data, and the site specific clusters and desktops where researchers and students do most of their work.

Table 1: Useful Definitions

Term	Definition
L-Store	The overall networking hardware and software suite that combine the individual depots into a single data cloud.
Depot	A server with minimal data management software and a heterogeneous collection of hard drives.
exNode	Contains the metadata (file size, location of data on depots, etc.) associated with a given file. Similar to a Linux inode.
Logistical Input/Output (LIO)	A command line software suite for reading and writing data in L-Store. Allows for fine control of the data flow.
Logistical File System (LFS)	A mountable file system, which allows the user to access all of REDDnet as if it were an external hard drive.

By deploying such depot clusters at geographically distributed sites and at different network (or project) "tiers," communities can create facilities for data intensive collaboration (e.g. a global "drop box" with automatic data logistics capabilities) that increase the productive utilization of wide area links, local area networks, and disparate computational resources. More specifically, L-Store services and facilities enable the community to:

- Optimize the rapid movement and timely placement of data for a wide range of scenarios, including policy-controlled data sharing (Beck 2003), replication (Huadong 2006), caching (Ding 2003) and high-performance, high-resilience data streaming (Atchley 2003, Bhat 2004, Plank 2003).
- Relieve application developers of the need to explicitly stage data from within their applications and scripts. L-Store provides this functionality in a data logistics service layer that supports more

effective data handling and simplifies programming; and it includes client libraries and tools that make it easy and transparent for traditional applications to get the benefits of these capabilities.

- Empower communities to work more cooperatively with their local IT organization to provision needed local storage resources. L-Store uses an application-neutral form of scalable storage virtualization that not only interoperates and peers with storage nodes in local, regional, national or global deployments, but which offers infrastructure providers a viable path for storage consolidation that can rapidly incorporate new hardware innovations and largely avoid vendor lock-in.

The availability of L-Store and its potential for immediate, high impact on the design needs of large scale or distributed collaborations rests on the fact that the software components that it combines already exist as a result of prior research and experimentation in the area of Logistical Networking (LN) (Beck 2000). LN is the use of storage to decouple data generation and consumption from end-to-end data transfer, generalizing the paths available through time and space. The central components of L-Store such as the Internet Backplane Protocol (IBP) (Beck 2002, Plank 1999, Plank 2001) have all been tested, and in some cases fully deployed in several large projects hosted at Vanderbilt University including: The Compact Muon Solenoid (CMS**;** http://cms.web.cern.ch) experiment for the Large Hadron Collider (LHC**;** http://home.web.cern.ch/about/accelerators/large-hadron-collider) and the Vanderbilt TV News Archive.

There are two distinct data management modes that will be addressed in this document: Point to point transfers and large scale data distribution and storage. The point to point transfer mode is a unidirectional transfer of data at rapid speeds from a collection of source depots to a collection of destination depots. The large scale data distribution and storage mode is much more concerned with efficiently and reliably storing data for long periods of time, as well as, making the data available to the cloud. One might consider this the Netflix model, where the goal is good service to our customer. Although other methods of distributing data already exists, at the petabyte scale this is a far from trivial problem to solve, and a significant portion of this document will address the issues created by this amount of data.

Similarly, there are different methods for various types of users to interact with the L-Store system, providing capacity for different types of use cases. One method is as a mounted file system which uses normal Unix/Linux commands and where files can be accessed normally by applications. This is the primary way that most users would interact with the system. One could also use this method as the backend for a web server or other higher level application. Another method uses a native command line tool set. This method would be used for large, rapid data transfers from site to site. It would also be useful for overall data management, where controlling the physical location or doing low-level caretaking of the data is important.

This document is broken down into three main sections. The first section is this overview. The second is a technical description of the L-Store hardware and software. The third section will look at the theoretical and practical problems involved with network and data management at the petabyte scale. This document concludes with appendices with additional L-Store technical information, a glossary, and references.

2. What is L-Store?

2.1. L-Store Hardware

The depot is the fundamental hardware unit for storage. A typical depot consists of a chassis containing a power supply, one to many hard drives, an off the shelf cpu, and a 10 Gb network card. One hard drive contains a minimal Ubuntu Linux based operating system, and the rest of the drives are formatted for use as storage. The final component is the IBP server software, which manages all the data on a depot. The hard drives can be a heterogeneous set in both size and vendor, making lifecycle management or replacement easy.

The second benefit is simplified and natural expandability and upgradability. Old hard drives can be replaced with new larger and faster hard drives. The L-Store system allows the addition of new hardware without throwing away the old hardware, naturally expanding and updating the capabilities of the storage cloud as time progresses. This benefit is already paying significant dividends on currently installed systems. During its brief three year lifetime, the current 2.5 petabye CMS L-Store installation has expanded by a factor of three in terms of raw storage, and the mix of installed hardware has become quite heterogeneous.

2.2. L-Store software overview

The Storage for distributed collaborations typically consists of an array of distinct storage elements—each within separate administrative domains—that may be connected together with a common Grid transfer and security mechanism. Data transfers are organized around files, which are copied between sites. A Storage Cloud, the other hand, can be distributed in nature, will look the same from all access locations, and defines a single administrative domain for all its data. Data transfers within a storage cloud can, like a typical file system, be organized around data blocks arbitrarily located within the cloud.
The power and flexibility of L-Store can be traced to Logistical Networking (LN), a distributed storage architecture designed explicitly to offer a scalable solution to problems of data logistics in distributed computing in the WAN context. Many of the advantages of LN derive from the way in which it handles file metadata. Conventional file systems operate on a file abstraction that consists of two kinds of stored information: the actual data that is the contents of the file; and structural metadata, which we also call a data mapping. When a file is written to a file system, its contents are divided into pieces (blocks) and each block is written and stored separately. The map to where each block is stored and other state information is the structural metadata. In UNIX, this metadata is called the inode. Typically, the user does not have access to this metadata.

LN uses a generalized abstraction of the inode, a container called the exNode. Information about a distributed file's data mappings is placed in the exNode. In addition to the file's structural metadata, arbitrary non-structural metadata, called attributes, can be stored in the exNode. The generality of this abstraction allows us to combine storage resources from many sources, implemented by many different technologies. Files stored in this distributed mix of hardware, along with their metadata, can be manipulated by users through widely used interfaces with familiar semantics.

As discussed in Section 2.1, the fundamental unit of storage in LN is the depot. A depot can be a single disk or a collection of disks in a server. In LN, the pieces or blocks of a file are called allocations.

As currently implemented in L-Store, exNodes use one basic storage access protocol—the Internet Backplane Protocol (IBP). IBP provides a generic, best-effort service that allocates, reads, writes and manages allocations on network-addressable storage (the depots). The exNode data mappings capture the way in which the (possibly replicated) segments of a data extent are mapped to IBP allocations. By factoring the data movement and storage and data-mapping aspects of the file abstraction, the LN architecture exposes the structural metadata in a form that can be safely and directly managed and manipulated by client processes and by services acting on behalf of the client. IBP allows for third party transfer of data between depots relieving users of the burden of moving data around the Internet, as other middleware tools can do this on their behalf as seen in Figure 1.

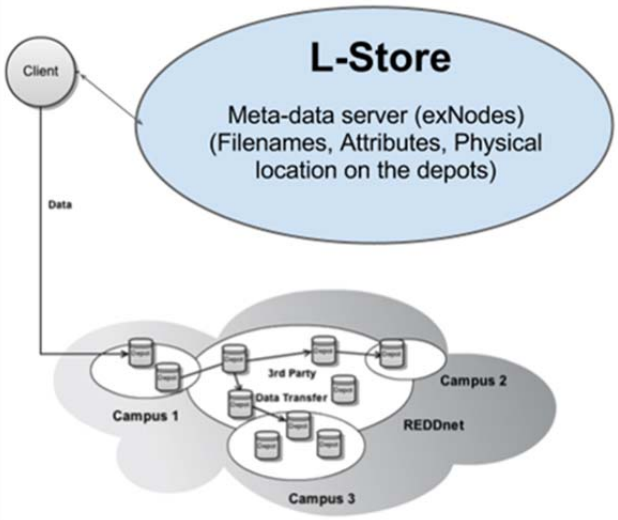


Figure 1. The client accesses metadata in the L-Store namespace and then locates the actual data across many different depots at different locations.

The L-Store software stack implements a complete virtual file system. L-Store uses IBP as the underlying abstraction of distributed storage. It manages, monitors, and maintains data distributed to IBP depots, and supports RAID-like mirroring and striping of data for performance and fault tolerance.

L-Store supports the complete lifecycle management of hardware. No downtime is required to add disk storage, which can be added on the fly with the space becoming immediately available. As hardware is added, allocations can be redistributed to maintain uniformity in data distribution. Hardware can be retired, resulting in the automatic migration of data off the retiring hardware. IBP supports heterogeneous disk sizes and storage hardware. As a result, L-Store can grow based on demand, using the best technology. This has become a routine and common occurrence in the multi-petabyte L-Store storage element used at the Vanderbilt CMS Tier2 center.

There are two main methods for interacting with L-Store: command line tools, and a FUSE based disk mount. The Logistical Input/Output (LIO) command line tools provide the most complete and efficient way to access the L-Store data. These tools are designed to replicate familiar Linux commands such as cp, du, find, fsck, ln, ls, mkdir, mv, rm, rmdir, and touch. A person with a reasonable familiarity of these Linux commands can easily transition to using the LIO command line tools. The LIO tools also provide commands for L-Store specific tasks such getting and setting file attributes, extending the lifetime of a file, directing output to standard out or input to standard in, or inspecting the health and physical location of files. These tools are easily incorporated into shell scripts and the like allowing for straightforward access, manipulation, and administration of files in L-Store.

The second access method is even more user friendly and transparent than the LIO tools. L-Store takes advantage of a kernel module that is readily available on most Linux distributions and Mac systems, the File system in Userspace (FUSE) kernel module (http://fuse.sourceforge.net/). L-Store provides a FUSE-

based interface called the Logistical File system (LFS). With minimal configuration, and using a single command, LFS can be mounted on an investigators machine as if it were an external hard drive. This mount point allows the user to carry out analysis as if the data was located on his or her local machine.

Instances of L-Store have been in production use for over three years at the Advanced Computing Center for Research and Education (ACCRE), and it is the ACCRE team who are the primary developers of L-Store software. The CMS repository at ACCRE currently stores over a petabyte of CMS data, and it has been pushed hard by CMS users. There are, however, significant development efforts that are being undertaken to assure that L-Store meets the evolving needs of CMS.

2.3. L-Store Architecture

L-Store is comprised of a set interoperable building blocks defined by a collection of abstract functional interfaces. Each functional interface targets a distinct core service. In most cases, there exist multiple implementations of a core service with the various implementations combined together in unique ways to provide additional functionality. This building block approach provides an extensible framework to allow L-Store to be adapted in the future as technology and demands change.

A quick overview of these building blocks is provided below with a more in-depth discussion available in Appendix A:.

- The ***Object Service (OS)*** provides the traditional file system semantics for manipulating files and directories. It also provides support for arbitrary metadata to be associated with any object.
- Object access is controlled through the ***Authentication and Authorization Services***.
- The ***Resource Service (RS)*** is responsible for mapping resource requests to physical resources and monitoring the health of physical resources.
- The ***Data Service (DS)*** is responsible for block level data movement and storage. Currently IBP is the only implementation supported.
- The ***Segment Service*** consists of software drivers that control the logical layout of data on physical resources. These drivers could be designed to interface to other repositories, for example iRODS, S3, or WebDAV for example providing whole file access. They can also perform block-level I/O using the *Data Service* to perform sophisticated fault tolerance and caching schemes.

3. What are the challenges of wide-area network and data management at the petabyte scale?

The operation a large distributed storage facility for data intensive collaboration involves many challenges. Reliability, performance, and ease of access are obviously important. L-Store has demonstrated its ability to provide these attributes in its use by application domains such as CMS and the Vanderbilt TV News archive. We discuss below other challenges that our experience has taught us are important.

Any time one changes a parameter by an order of magnitude new challenges arise. Managing multi-petabyte data sets is no different. Moving a few terabytes of data between remote sites can be done in under a day with existing tools and hardware. Ten Gb/s or greater networking is not required, and one can most likely use traditional hardware or software based RAID systems to insure data integrity. At the

petabyte scale, reliable high performance networks become vital and standard data integrity solutions such as RAID5 and even RAID6 become less viable.

3.1. Why is data integrity such a big issue at petabyte scales?

Data integrity becomes a much more significant issue at petabyte scales due to increasing probabilities of data loss and array rebuild times. At these scales, unrecoverable read errors start to become statistically significant.

Traditional RAID array rebuild times can span days when using multi-terabyte hard drives. One can overcome this by using a distributed RAID array which is discussed later.

When modeling failure modes it is usually assumed that drive failures are uncorrelated but in our experience that is not the case. Typically all the drives in an array are not just from the same build batch but will have sequential serial numbers. Manufacturing defects or firmware bugs lead to systematic and correlated problems. In addition, because rack space is typically a precious commodity one tends to use dense disk enclosures which can suffer from mechanical vibrations and broken circuit traces on connectors.

3.1.1. Data integrity Schemes

Data integrity schemes typically employ either data mirroring or some form of RAID. Data mirroring is nothing more than keeping multiple copies of the data stored on different devices at different locations. Mirroring is great for read dominated workloads since all replicas can be used to service user requests. Write performance is slowed by the making of all the replicas. Mirroring is not very space efficient which can be quite costly at the petabyte scale. It is also not very good at providing data integrity as can be seem from Figure 2. We assumed a 1% failure probability in generating this plot. Changing this probability will change the scale of the y-axis but the relative positions of the different schemes will not change dramatically.

The other standard approach is to employ some form of RAID. In this case one stores the data and additional parity information that can be used to reconstruct unrecoverable bit errors or drive failures. The two most common forms for RAID are RAID5 and RAID6. RAID5 uses one additional disk for storing parity and is can survive a single device failure. RAID6 has two additional disks for parity and can survive two drive failures. Reed-Solomon encoding is the generalization or RAID5 and RAID6 to support arbitrary numbers of drive failures. There are other RAID encoding schemes but these are the ones we will focus on without loss of generality.

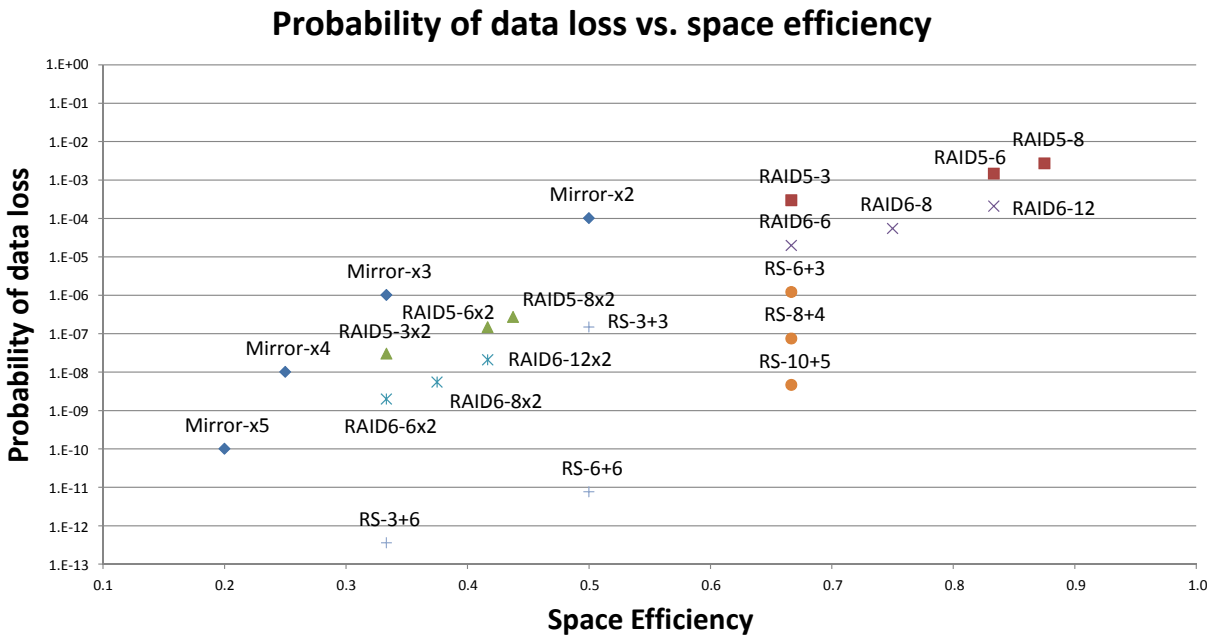


Figure 2. Probability of data loss vs. space efficiency for various data integrity schemes. Space efficiency is defined as (unique data)/(data+parity). We assumed a 1% probability of disk failure, changing this assumption will change the scale of the y-axis but not dramatically change the relative positions of the results for each scheme. The ideal configuration is in the lower right corner and would have a low chance of data loss and make good use of space. Mirror-xN signifies N replicas of data. RAID5-N and RAID6-N correspond to N total drives, data and parity, used in the array. The RAID6-Nx2 correspond to RAID6 arrays using N total drives that are replicated. RS-D+P represents generic Reed-Solomon using D data disk and P parity disks. The Y-axis should be scaled by the number of drive arrays to calculate the actual probability. For a fixed storage capacity (1 petabyte, for example) the number of drive arrays required will vary depending on the data integrity scheme used.

L-Store has implemented generic Reed-Solomon encoding along with several other data integrity schemes. We typically recommend 6+3 Reed-Solomon (RS-6+3) encoding -- 6 data disks and 3 parity disks. More generally we recommend the use of the RS-d+(d/2) family of configurations which can be seen in Figure 2 as RS-6+3, RS-8+4, and RS-10+5. This family uses 2/3 of the total space for data with the remainder for parity and has excellent reliability. RS-6+3 has the same reliability as keeping 3 copies of the data but uses only half the space.

3.1.2. Array Rebuild Times

As alluded to previously, recovering from the failure of a multi-terabyte disk drive can take days. We have witnessed this first hand in a 500 terabyte GPFS file system used to provide home disk space for a large campus cluster. The possibility of correlated drive failures (for the reasons described in the introduction of Section 3.1) increases the probability that additional drive failures could occur, resulting in unrecoverable data loss. Figure 3 shows the probability of data loss due to the loss of additional disks as a function of the rebuild time for three RAID configurations. RAID5-6 provides very little protection in this scenario. The RADI6-8 configuration is better since two additional drives are required in order to lose data. Even so, the chances of data loss are uncomfortably high if it takes a few days for the repair. The RS-6+3 is substantially more reliable even if repair takes a few days but keeping the rebuild times down to a few hours helps minimize the chance of data loss. The graph shows a distinct knee for this

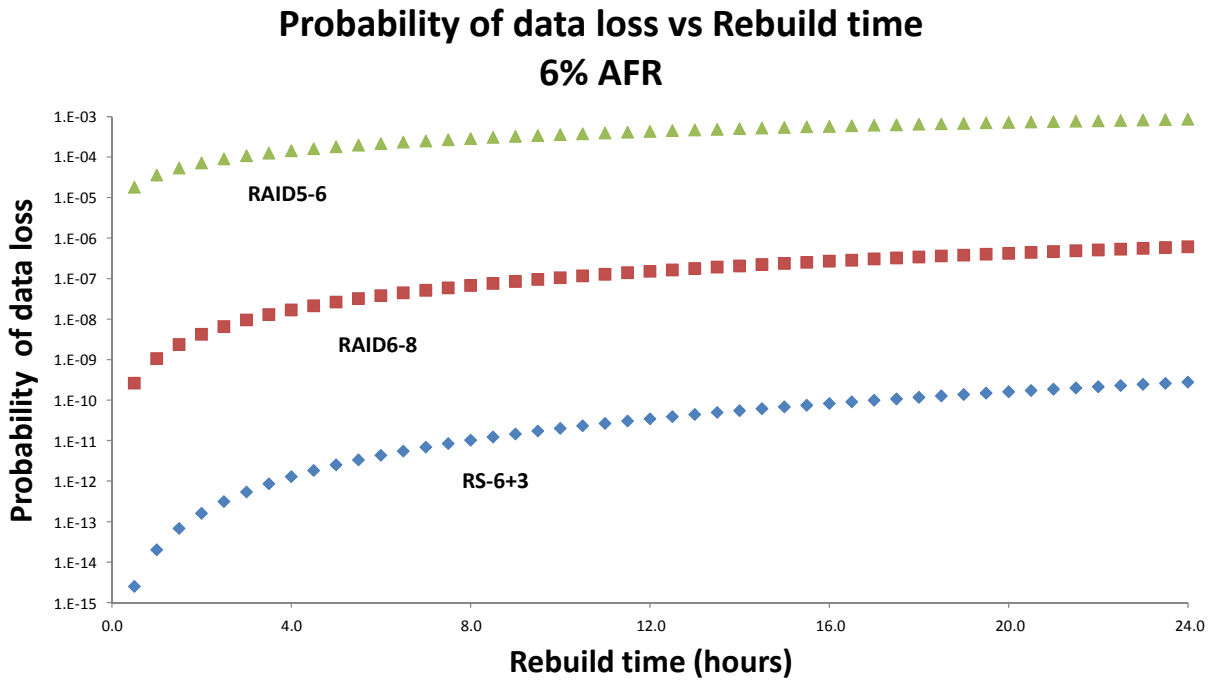


Figure 3. Probability of additional drive failures occurring during the initial drive rebuilds leading to data loss for three common RAID configurations. Ideally repairs should take as little time as possible but keeping them down to a few hours almost as good for RS-6+3. The plot assumes a 6% Annualized Failure Rate (AFR) based on the literature. The probability given is for a single array.

configuration occurring in the first couple of hours. The plot assumes a 6% Annualized Failure Rate (AFR) which is taken from the literature (Pinheiro 2007;Schroder 2007) for 3 and 4 year AFR estimates.

3.1.3. Distributed RAID Arrays

Traditional RAID arrays completely reconstruct a *single* failed drive on a *single* replacement drive. The use of a single replacement drive is a major factor in the rebuild time. Traditionally the entire drive is reconstructed with no regard for used vs. unused space. If the array is active with user I/O requests, this will greatly increase the rebuild time.

Distributed RAID arrays are designed to overcome these limitations. Instead of using the whole disk the disk is broken up into many smaller blocks. These blocks are combined with blocks on other disks creating many small logical RAID arrays utilizing a large subset of the available drives as shown in Figure 4. These distributed logical RAID arrays are based on space that is actually used. The free space on each drive can be used to store the newly reconstructed data. This allows for a large number of drives being read and written to simultaneously providing significantly faster rebuild times. For L-Store each file and associated parity is placed on a random collection of drives based on the fault tolerance scheme used and data placement criteria. We routinely rebuild a single 2TB in a couple of hours using a single host to perform the data reconstruction. Adding more hosts causes the rebuild time to proportionately decrease.

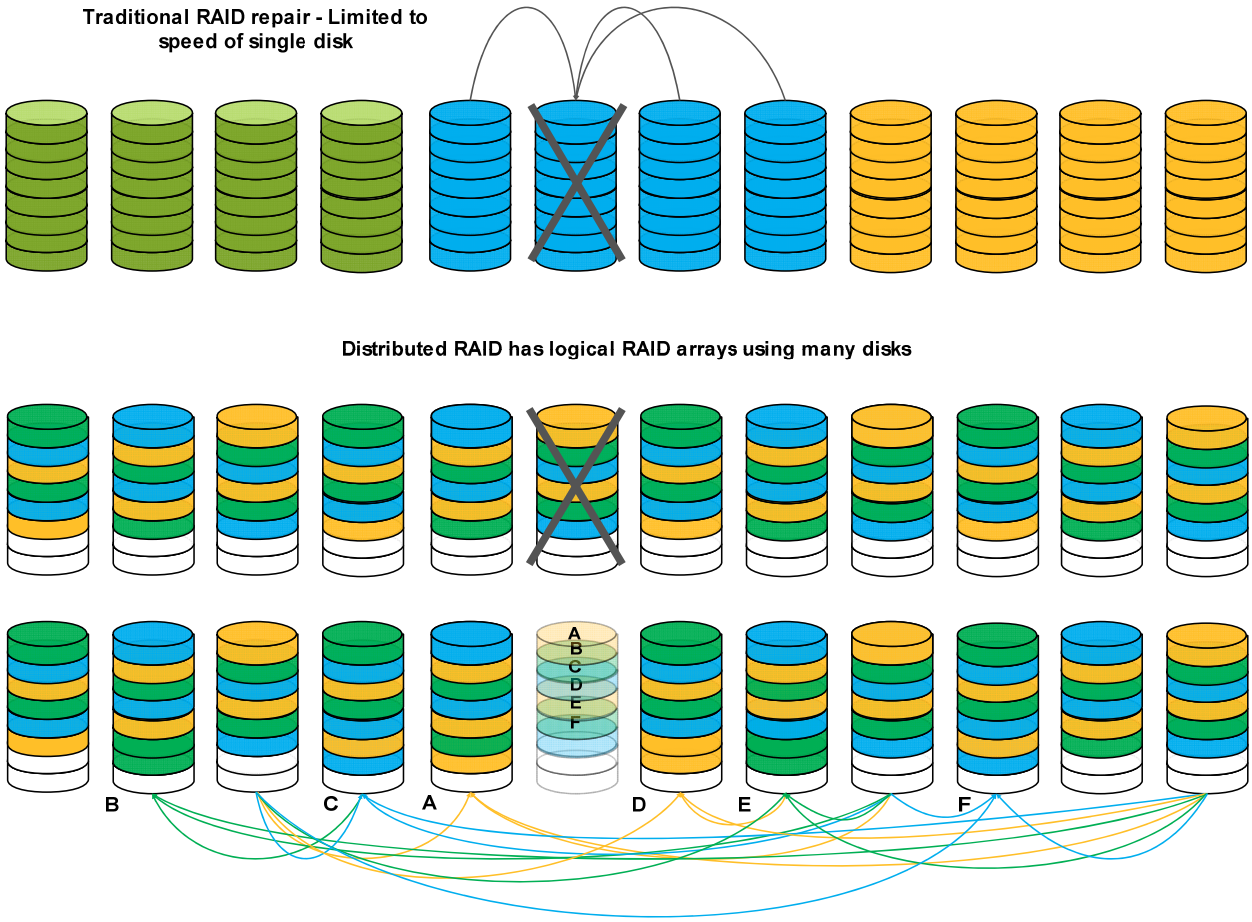


Figure 4. Diagram showing a traditional RAID array rebuild vs. using a distributed approach.

3.2. What is the difference between the LAN and WAN contexts?

Managing and providing data access over a wide area network can be significantly more challenging than over a local area network. One often has complete control over the local area network, whereas one has little control over a wide area network. Latency, firewalls, asymmetric routes and alternating routes with significant bandwidth differences, outages, and congestion can be essentially ignored in the LAN context but are gremlins waiting to pounce in the WAN. Some of these problems are demonstrated in Figure 5, which shows the data transfer rates between NCSA and Vanderbilt. The ability to handle these difficulties is one of the key tasks for L-Store.

One needs a data management policy framework that can cope with these gremlins. Network aware transfer tools are one important way to mitigate an uncertain network landscape. Tools such as Phoebus (http://www.internet2.edu/performance/phoebus/) exist for this purpose. Quoting from the Phoebus website: “Phoebus is an environment for high-performance optical networks that seamlessly creates various adaptation points in the network to maximize performance. By splitting the network path into distinct segments, Phoebus minimizes the impact of packet loss and latency by leveraging the best performance attributes of each network segment. Using an end-to-end session protocol, transport and

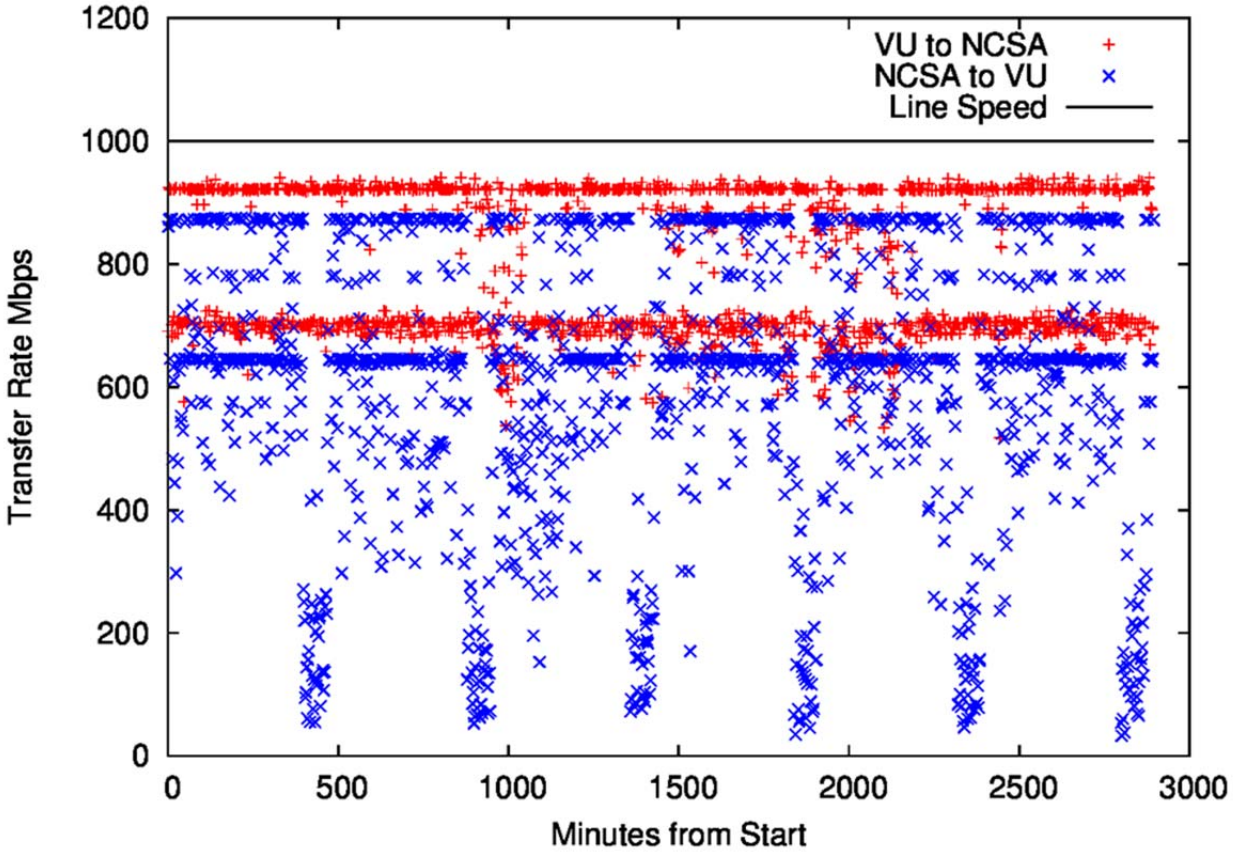


Figure 5 A two day bi-directional network test between Vanderbilt University (VU) and NCSA. There is a small asymmetry between the two data flow directions. There are also clearly two stable routes between these two end-points with different transfer rates, as well as a significant bandwidth degradation that occurs only in the NSCA to VU direction on an 8 hour repeating cycle.

signaling adaptation points can be controlled and better performance is possible." The core functionality of Phoebus is already incorporated into the L-Store architecture. In order to take full advantage of these low level building blocks, development work is underway to build higher level Phoebus command and control into the L-Store data management framework. L-Store team members are also collaborating on NSF funded projects (DYNES and ANSE) working to develop tools to enable the dynamic allocation of dedicated circuits over a shared network connection. The ability to use these sorts of third party tools is built into both the messaging and network infrastructure layers of L-Store.

3.3. Why do we use the "Web-like" architecture that we use?

Large scale parallel applications arose from the scientific community, who required tightly coupled shared-data centric applications connected together by low-latency dedicated networks. This highly coupled design leads to a concurrency model that is brittle when problems such as increased latency, slow responses, dropped connections, or application segmentation faults occur. These issues are the norm for today when using commodity hardware. These applications work best when they can "share everything" such as when they run on a single compute node and can share memory and other resources. At the opposite extreme is the "share nothing" approach, where the parts of an application run independently of each other and share no common resources, relying on communication protocols

to synchronize as needed. We have adopted an approach that lies somewhere in between these two extremes. On a single server, we use a tightly coupled approach to squeeze out every ounce of performance. Similar to the Web, we adopt a "share very little" approach when coordinating work between collections of distributed computers. This approach minimizes the use of global and local locks and emphasizes asynchronous task execution leading to loosely coupled networks which scale much more easily. This idea is most obvious in the use and design of IBP which is modeled on the narrow waist underpinning all network transfers – the internet protocol.

The L-Store approach focuses on providing primitives or building blocks for application developers to use in order to construct whatever communication schemes are needed for their application. The goal is to provide applications the freedom to design schemes based on their unique requirements rather than having to work around a built in, generic set of assumptions made by the L-Store developers.

Appendix A: Technical Description of L-Store Architecture

A.1. Plugin Architecture

L-Store is designed from the ground up to be extensible with support for dynamically loaded plugins. The plugin framework provides a grab bag context that applications can fill with whatever is needed. This could be simple variables, complex data structures, or dynamically loaded functions. This grab bag is then passed as an argument for most object creation routines along with a handle to the configuration file and section or stanza to use for instantiation. The configuration files uses a variation of the standard INI file format with additional support for nested inclusion of files and repeated identically named sections.

The type of service instantiated is completely controlled by the configuration file. To better understand how this is accomplished let us walk through a simple example. A version of the configuration file stripped down to just the essential fragments for illustration purposes is shown in Example 1.

Example 1. Stripped down configuration file

```
[lio]
timeout = 60
max_attr_size = 1Mi
ds = ibp
rs = rs_remote_client
tpc_cpu = 4
tpc_unlimited = 10000
os = osfile
cache = cache-amp
#cache = cache-lru
user=tacketar

[rs_simple_client]
type=simple
fname=/etc/lio/rid.cfg
dynamic_mapping=1
check_interval=60

[rs_remote_client]
type=remote_client
rs_local = rs_simple_client
child_fname = rid-client.cfg
remote_address = tcp://127.0.0.1:6713
dynamic_mapping = 1
check_interval = 3600
check_timeout = 60
```

By default all L-Store routines load the *lio* section or stanza but this is easily overridden. It is mainly a collection defining which stanzas of the various services to load. We are going to focus on the sequence

for instantiating the Resource Service which is controlled via the "rs" key above. In this case it has the value "rs_remote_client". So we are supposed to load the *rs_remote_client* but in order to do that we need to determine the plugin to use. This is done by looking at the "type" key in *rs_remote_client*. In this case we need to instantiate a RS of type "remote_client". We look in the global grab bad and see if we find a match. If so, we call the creation function and return it.

It does not end there, though. We can continue down the rabbit hole since a RS of type "remote_client" also takes as one of its configuration options a RS service it uses locally. The remote service is just for getting a list of resources. It's not used to do the actual allocations. Just for resource discovery. The local resource service to use is stored in the "rs_local" key which points to the "rs_simple_client" stanza. The process now repeats itself. We look at the "type" key and see that rs_simple_client is to use the "simple" grab bag object for instantiation.

There are two ways objects and services are installed. For the common services they are automatically included in the binary and nothing special has to be done. The other option is to dynamically load the plugins at runtime. This is accomplished through one or more *plugin* stanzas similar to the ones in Example 2. Each plugin has four required variables. The first two control how they are placed into the grab bag. They define the stanza or section, *section* key, and the key or name, *name* key. This is similar to the way they would be accessed or written in the INI file format, section and key.

Example 2. Plugin stanza.

```
[plugin]
section=segment_load
name=file
library=/etc/lio/plugins/segment_file.so
symbol=segment_file_load
```

A.2. Object Service (OS)

This is the workhorse layer for providing traditional file system semantics. The OS handles the creation and removal of objects. An object is just a file or directory. Symbolic and hard links are supported as well. Each object can have an arbitrary collection of metadata or attributes attached, both virtual and traditional (See Section A.2.1). Symbolic linking of attributes between objects is also supported. Read and Write object locking is supported through the *os.lock* virtual attribute.

A.2.1. Attributes

A traditional attribute is a key/value pair where the key and value are simply stored or retrieved. A virtual attribute is an alias for a piece of C computer code. This code is executed for all read/write operations associated with the attribute name. Virtual attributes would be considered micro-services using iRODS terminology. They are not designed to perform complex tasks. That is what the Messaging Queue Framework, Section A.8, is for. Instead virtual attributes are designed to be quick, light weight plugins to perform simple tasks.

There are two types of virtual attributes – *fixed name* and *prefix matched*. The type of virtual attribute is specified when it is "plugged-in." A *fixed name* attribute is the simplest. It provides a direct mapping between attribute name and plugin. The attribute os.link is a fixed virtual attribute. Querying this attribute returns the object type, for example file, directory, or link. A *prefix matched* attribute means the virtual attribute name is a *prefix* of the attribute name requested by the user. For example *os.timestamp* is a prefix virtual attribute and would be triggered by a request to access the attribute

os.timestamp*.my.date*. In fact it gets triggered anytime the first 12 characters of the attribute requested correspond to "os.timestamp". It is easiest to understand the differences by looking at some examples.

The sequence to resolve an attribute is as follows: The OS first scans the list of fixed name virtual attributes for a match. If none occur, then it scans the prefix matched virtual attributes looking for a match. If still no match a lookup is performed for a traditional attribute with that name.

Attributes use a dot separated namespace. The native attributes supported by the OS all begin with "os" and are listed below:

Fixed virtual attributes:

- os.lock – Retrieve an object's read/write locks
- os.link – Retrieve an object's symbolic link
- os.link_count – Returns how many times the object referenced
- os.type – Returns the object type (file, directory, symbolic link, etc)
- os.create – Returns the object creation time

Prefix matching virtual attributes:

- os.attr_link – Returns an attributes symbolic link
- os.attr_type – Returns the type of attribute (virtual, symbolic, text, etc)
- os.timestamp – Sets or retrieves a timestamp attribute

Example 3 shows the output of the *os.link* fixed virtual attribute. In Example 3, /lio/lfs is the L-Store mount point and we have created a symbolic link of bar->foo which shows up when probing the os.link attribute. Note that foo returns (null) as expected because it is not a symbolic link.

Example 3. Settings and accessing attributes.

```
root@depot1:/lio/lfs/ncsa1/testing# echo "bar" > foo
root@depot1:/lio/lfs/ncsa1/testing# ln –s foo bar
root@depot1:/lio/lfs/ncsa1/testing# ls -l
total 0
lrwxrwxrwx 1 root root 3 Mar 21 15:43 bar -> foo
-r--r--r-- 1 root root 4 Mar 21 15:43 foo
root@depot1:/lio/lfs/ncsa1/testing# lio_getattr -al os.link foo bar
object=/ncsa1/testing/foo
  os.link=(null)

object=/ncsa1/testing/bar
  os.link=foo
```

In Example 4, we explore working with virtual attributes. Now let us create a timestamp just for my use. Let us call it *my.date* and I would like to add a little text to it to let me know who created it which was "me". I could just store my system date in the attribute *my.date* along with the "me" but in a distributed environment it is probably better to use a synchronized clock that we can all agree on. In this case, we will use the clock in the OS. To do that, we will need to use the *os.timestamp* virtual attribute. Since this

is a prefix matched virtual attribute it gets triggered anytime the virtual attribute is a prefix of the attribute being specified. In this case, that is *os.timestamp*. *Os.timestamp* takes the rest of the attribute name passed and creates a normal traditional attribute using that name. The epoch time, in seconds, is stored in the traditional attribute along with provided text, separated by the pipe, "|", symbol. For Example 4, we want that to be *my.date*. This means we need to form the composite attribute *os.timestamp.my.date* and use that when setting the attribute.

Example 4. Setting an attribute by incorporating another attribute.

```
root@depot1:/lio/lfs/ncsa1/testing# lio_setattr -as os.timestamp.my.date='me' foo
root@depot1:/lio/lfs/ncsa1/testing# lio_getattr -al my.date foo
object=/ncsa1/testing/foo
  my.date=1363899327|me
```

Notice that the lio_setattr command created the *my.date* indirectly. The *os.timestamp* virtual attribute appended our extra text to the timestamp.

A.2.2. OS Implementations

Currently the workhorse OS implementation is os_file. This uses a traditional disk file system to store all metadata. As a result this implementation does not work between nodes. In order to overcome this limitation we have added the os_remote_client and os_remote_server implementations. These allow the OS from a client to connect to a remote OS acting as a server. The OS remote server will then use the os_file driver for the actual backing store. This combining of existing building blocks to provide new functionality is a real strength of L-Store and is done routinely.

The current os_file driver has inherent limitations since it is isolated to a single machine and uses a traditional file system to store data. One can get pretty good scalability using an SSD for the disk. Likewise using the linux rsync and inotify tools one can achieve high availability if the primary OS server dies. We do this for the current version of L-Store and plan on adding it to the new version as well. In order to scale to billions of files one needs another approach. We have been looking at various NoSQL databases and plan on using one of them as the core for a new NoSQL OS. Currently the most promising is Apache Cassandra.

A.3. Authentication and Authorization (AuthN and AuthZ)

The primary authentication system in development is relatively traditional using a username for identification and a password, key, or certificate as credentials to vet identity. Also in the works is a utility that will work in conjunction with this authentication module that can be used to create an authentication session. This tool would securely cache the user's identity and credentials for a limited duration and environmental scope (for example: 1 hour and only within a particular login session on the user's computer). This tool will give the user the convenience of being able to perform operations on the L-Store system without having to resupply authentication credentials with every operation. This tool fulfills the same role in L-Store that ssh-agent does in the SSH suite.

Alternate authentication systems will also be supported and a great deal of work is being done to carefully design an authentication framework that can support modules for all future authentication and user management models. Authentication and identity management is greatly more complicated in a

distributed system, such as L-Store, than in a local one. Such a system must properly enforce security policy while operating across administrative domains and also have the flexibility to allow integration with a variety of external authentication/user management systems. Additionally distributed systems must be capable of working within untrusted environments, with untrusted clients, and over untrusted networks.

A user's identity in a remote/distributed file system, unlike in a local file system, is not the same as the user's operating system account on the local computer. A user thus has both a local identity and a remote file system identity. Generally that remote identity would be claimed and vetted for use with more than a single operation so that identity must be retained but with appropriately restricted access. In many cases, a one-to-one mapping of remote identity to local identity is sufficient, however being able to restrict the remote identity credentials to a single application process or session is needed in some use cases. Additionally a local user on the client machine may need to take on multiple remote identities, or it may be desirable to allow multiple users on the client machine to share a single remote identity, either fully or in a limited or delegated fashion. To add to the complexity there may be other user identity systems to integrate with, for example a LDAP single sign-on service. Integration could take the form of either mapping the external service's identities to identities used by the distributed file system or using the service as the file system's own user identity authority. In designing the authentication framework for L-Store, we take the view that supporting all of the above scenarios should be possible.

Collaborating with the CMS project and operating as an OSG (Open Science Grid) site has given the ACCRE team direct experience with some complex security and user management scenarios. CMS researchers juggle many identities and operate between many administrative domains when doing work interactively, when managing batches of long-running jobs, and when storing and accessing datasets. A X.509 certificate based system provides an identity that can be used across grid computing sites for Grid services but beyond that the sites are heterogeneous. Researchers often have separate accounts on clusters at a handful of sites. These accounts must get mapped to a Grid account to access Grid services and in some cases when Grid services are accessed at remote sites these grid identities themselves get mapped onto site specific accounts. Grid credentials are capable of being delegated through creating proxy certificates and this is used for creating sessions through time-limited credentials, delegating access to batches of computational jobs that may run at a variety of sites as a variety of users, and arranging third party transfers of data. At the Vanderbilt site, the L-Store system has to manage access to the CMS datasets for local and remote Grid users. In short it is vital to have a flexible and robust authentication framework so L-Store can integrate into complex systems.

The authorization module in development is non-traditional in design and implementation. The module is designed to be enormously flexible and powerful to enable the use of rich and unique authorization policies. However the ability of the module to behave just like familiar authorization systems to the user is also priority. The module is designed to provide a superset of the capabilities of common, traditional authorization systems and is designed to be easily configured to replicate traditional behavior. This authorization module uses an embedded Prolog logic engine to process authorization rules and make authorization decisions.

The authorization hooks in the object service make authorization requests using a tuple composed of the action proposed, the object subject to the action, and the credentials presented, where object is a generalization that encompasses files and attributes, and the credentials are vetted by the authentication module. Based on the request tuple, the authorization rules, and any relevant metadata

(associated with the object such as a file's owner and permissions attributes and the attributes of a containing folder, or associated with the credentials such as an attribute containing a user's group membership) the module will either allow or deny the action proposed.

The authorization rules are not hard coded into the module but rather they are configurable both system wide and on a domain/folder basis and can be reconfigured on a running system. Rule sets are being created to model standard POSIX (Unix/Linux) style permissions and NFSv4 ACLs (an open standard nearly the same as the ACL systems used by OSX and Windows operating systems). Authorization rules can potentially be used to create whole custom authorization systems (for example a role based access control system) or extend the common authorization systems (for example adding the capability for a file owner to share a file with a colleague by emailing them a hyperlink containing a key for anonymous, read-only, and perhaps time-limited access).

Large binary objects are physically stored outside of the Object Service on Depots but the data is only accessible via handles to the data in the exNode. These handles to the data provide limited capabilities to read or modify the data. Thus authorization policy is enforced by returning different views of the exNode with different capability handles based on the action approved by the authorization system. For example if just read-only access is approved by the authorization system then only a version/view of the exNode containing solely read-only handles to the physical data will accessible. See Section A.6.7 for more on exNodes.

Consult **Error! Reference source not found.** for the complete timetable of authentication and authorization system development.

A.4. Resource Service (RS)

The RS has two functions: Maintain a list of available resources and map user requests to physical resources. A resource or RID has three mandatory attributes (*rid_key*, *ds_key*, and *host*) and an arbitrary collection of user defined attributes. The *rid_key* is unique, so dynamic mapping can be enabled to automatically remap resources. For example changing the hostname or moving the resource to another depot. The data service key, *ds_key*, is the value passed to the data service in order to reserve the space. Technically *host* is not mandatory but the most common failure group is host based. It is best to stripe data across depots if possible. Another common attribute is *site* or *lun* to allow for replication or data movement between remote sites. It's easiest to understand the difference by looking at an actual RID definition as shown in Example 5.

Example 5. RID example definition.

```
[rid]
rid_key=1301
ds_key=illinois-depot1.reddnet.org:6714/1301
host=illinois-depot1.reddnet.org
lun=ncsa
```

In this case the *rid_key* is just the RID id, 1301. The RID is located in the depot with the *host* name illinois-depot1.reddnet.org. The *ds_key* is an amalgam of this information along with the IBP server port, 6714, and is passed directly to the DS. The DS then parses the information and uses it to locate the resource. There is an additional attribute named *lun* with the value *ncsa*. This is used to signify that the depot is located at NCSA. Additional attributes can be added as needed.

A resource query is used to find viable locations to place data. The resource query itself is a text string representing a postfix boolean expression. The available operators are AND, OR, NOT, KV_EXACT_MATCH, KV_PREFIX_MATCH, and KV_ANY. The names are pretty self–explanatory (KV =Key/Value). There are a couple of modifiers that can be applied to the KV operators: KV_UNIQUE, and KV_PICKONE. The purpose of KV_UNIQUE is straightforward; make sure the KV selected is unique. KV_PICKONE is normally used in conjunction with the KV_PREFIX_MATCH operator. It is designed to pick and use a single answer for all subsequent requests. For example assume we have a bunch of resources some with *lun=vu_accre* and others with *lun=vu_physics*. We want all the data to be stored at either ACCRE or physics but not both. To do this we use the KV_PREFIX_MATCH with the KV_UNIQUE modifier. The resource query is associated with a segment and stored in the exNode. The segment driver passes query to the RS along with other information, like the number of resources and size, to find suitable matches. See Section A.7 for an example of the interplay between the RS and the various other services.

The workhorse RS implementation is rs_simple which is designed to work in a single process space, thus it is not shared between processes. For that, one uses the rs_remote_client and rs_remote_server drivers. This allows all clients to use a common RS. Like the OS versions in Section A.2.2, the rs_remote_* drivers use the rs_simple underneath to do the heavy lifting. Rs_simple supports dynamic reloading of the configuration file. One simply needs to touch the configuration file to trigger reloading the configuration. If dynamic mapping is enabled any resource host changes are automatically picked up on the next read or write operation. The rs_remote_server automatically detects the change in its underlying RS and propagates it to all the rs_remote_clients registered with it. The RS service tracks space usage and if the host is up, it adjusts resources based on this information.

A.5. Data Service

All actual disk I/O is handled in this service. Currently ds_ibp is the only implementation and uses IBP as the underlying storage mechanism. This IBP implementation will be the focus of this section. IBP provides a best-effort storage service with stronger guarantees being enforced by higher levels of the software stack. In the L-Store case that is the segment service's responsibility.

As noted, IBP implements a primitive storage service that is the foundation of LN. As the lowest layer of the storage stack that is globally accessible from the network, its purpose is to provide a generic abstraction of storage services at the local level, on the individual storage node, or depot. Just as IP is a more abstract service based on link-layer datagram delivery, so IBP is a more abstract service based on blocks of data (on disk, memory, tape or other media) that are managed as "byte arrays." By masking the details of the local disk storage — fixed block size, different failure modes, local addressing schemes — this byte array abstraction allows a uniform IBP model to be applied to storage resources generally (e.g. disk, ram, tape). The use of IP networking to access IBP storage resources creates a globally accessible storage service.

As the case of IP shows, however, in order to have a shared storage service that scales globally, the service guarantees that IBP offers must be weakened. To support efficient sharing, IBP enforces predictable time multiplexing of storage resources. Just as the introduction of packet switching into a circuit switched infrastructure dramatically enhanced the efficient sharing of the "wires," IBP supports the time-limited allocation of byte arrays in order to introduce more flexible and efficient sharing of "disks" that are now only space multiplexed. When one of IBP's "leased" allocations expires (per known schedule or policy), the storage resource can be reused and all data structures associated with it can be deleted. Forcing time limits puts transience into storage allocation, giving it some of the fluidity of

datagram delivery; more importantly, it makes network storage far more sharable, and easier to scale. An IBP allocation can also be refused by a storage resource in response to over-allocation, much as routers can drop packets; and such "admission decisions" can be based on both size and duration. The semantics of IBP storage allocation also assume that an IBP storage resource can be transiently unavailable. Since the user of remote storage resources depends on so many uncontrolled, remote variables, it may be necessary to assume that storage can be permanently lost. Thus, IBP is also a “best effort” storage service.

To enable stronger storage services to be built from IBP without sacrificing scalability, LN conforms to classic end-to-end engineering principles that guided the development of the Internet. IP implements only weak datagram delivery and leaves stronger services for end-to-end protocols higher up the stack; similarly, IBP implements only a “best effort” storage service and pushes the implementation of stronger guarantees (e.g. for availability, predictable delay, and accuracy) to end-to-end protocols higher up the network storage stack. In its weak semantics, IBP models the inherent liabilities (e.g. intermittent unreachability of depots, corruption of data delivered) that inevitably infect and undercut most attempts to compose wide area networking and storage. But in return for placing the burden for all stronger services on the end points (i.e. on the sender/writer and receiver/reader), this approach to network storage tolerates a high degree of autonomy and faulty behavior in the operation of the logistical network itself, which leads directly to the kind of global scalability that has been a hallmark of the Internet’s success.

Since IBP allocations are designed to be transient you may be wondering how to get around this limitation. This is accomplished by periodically contacting the depot and extending the allocations duration. We call this process “warming” the file. If the depot refuses to extend the expiration, then data is migrated to another location.

Another issue commonly encountered in large scale storage is silent bit errors. The most common reference to silent bit errors is for a drive changing a bit’s state that is not detected by the disk drive. Given manufacturer specified unrecoverable bit error rates of 1 in 10^{14} or 10^{15} and the size of available drives, these do occur. The REDDnet CMS group sees these errors infrequently. But a more common silent bit error is from a drive or computer crash resulting in garbage being stored of portions of an allocation. The file system may replay the log and correct some of these errors but it does not catch all of them. This is even harder in a distributed context since the IBP write operation will have completed from the remote applications standpoint but before the operating system had the ability to flush the data to disk. One option would be to only return success to the remote client after a flush to disk had occurred but that would greatly impact performance.

Another option is to interleave block-level checksum information with the data. This has some impact on disk performance but much less than the alternative. The tradeoff is much greater processing power to calculate the checksums. This is done on the depot only which has plenty of excess computing power. Enabling block-level checksums is done at the allocation’s creation time. When an allocation has block-level checksums enabled every read and write operation involves comparison to the existing checksum to verify data integrity. If an error occurs, it immediately notifies the remote application allowing it to properly deal with the issue. This normally triggers either a soft or hard error, typically in the segment_lun driver, which is corrected by another layer. These errors are kept track of and used to update the *system.soft_errors* and *system.hard_errors* attributes. These are used, in turn, to trigger more in depth inspections of the object.

A.6. Segments

The various segment drivers are responsible for processing user I/O requests and mapping that to physical resources. It can use the DS and RS services in order to do this. The segment drivers support the traditional I/O operations read, write, truncate, remove, and flush but also inspecting the data for integrity, insuring the RS query is properly enforced, and cloning data between segments. There are many different segment drivers implemented and they are commonly combined together, see section A.6.7, to extend functionality. The segments themselves are all contained within the exNode with the exNode controlling how the various segments are coupled together and presented to the application.

A.6.1. Interfacing with other repositories

Segment drivers can be used to interface with other repositories. For example iRODS, Amazon S3, WebDAV, to name a few. The simplest example is mapping an L-Store file to a local file. This is done using the segment_file driver discussed in Section A.6.2. But due to the ease of extending L-Store functionality using the provided plugin framework, adding support for other repositories is well defined. Once implemented they can then be combined together with other segment drivers to provide additional functionality, for example combining the segment_cache driver with segment_file to enable local caching of a file.

A.6.2. segment_file

This is the only segment that does not use the DS or pass the DS on to a lower level. It simple maps an object to a physical file on a local disk and is more of a testing driver.

A.6.3. segment_cache

Caching segment driver. The segment_jerase driver does not allow reading and writing of arbitrary offsets and sizes. It can only perform I/O on fixed size blocks. The cache driver handles the impedance mismatch between what the user wants and segment_jerase needs by buffering data. There are two different types of caching implemented. The traditional Least Recently Used (LRU) method and a variation of Adaptive Multi-stream Preteching (AMP; Gill 2007).

A.6.4. segment_jerase

The primary driver providing fault tolerance. The driver implements all the functionality provided by Jim Planks's Jerasure library (Plank 2008). The most notable being the Reed-Solomon encoding.

A.6.5. segment_log

This driver implements a log-structured file. A log structured file only supports append operations, *i.e.* data is never overwritten. Instead it is appended to the file along with a new extent defining what was appended. This allows one to "replay" the log and create different versions of the file at any point in the log. The log segment is comprised of three other segments:

- Base – Normal flat address space used as the base for the file to apply the changes
- Log extents – This just contains all the extents (offset and length pairs) for changes.
- Log data – All the data corresponding to the extents.

The log extents are kept separate from the log data to facilitate quick access over the WAN since the extents can be read in large reads with the log data being accessed as needed. Any range requests not in the log is retrieved from the base.

This driver has additional functionality to add additional logs on top of the current log and also merge existing logs with the base. This driver provides the foundation for handling file versioning, snapshotting, and regional caching.

A.6.6. segment_lun

This driver takes a collection of allocation and turns them into a logical unit or LUN. Similar to what a hardware based RAID appliance does. It takes as input the number of virtual disks, chunk size, optional shift, and the RS query to control data placement. It then takes that information and converts it into a flat address space for use. Normally a chunk size of 16k is used. See Figure 6.

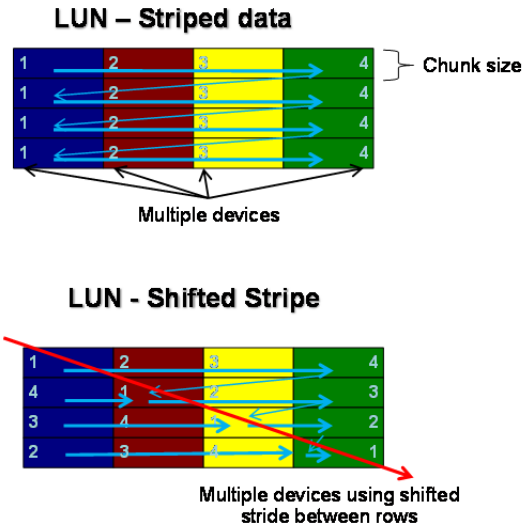


Figure 6. Examples of the segment_lun driver data layout. In each example 4 allocation or devices are used. The arrows show the logical byte ordering of the data. The data can be shifted (bottom diagram) in order to interleave data and parity on the same device or allocation.

A.6.7. ExNodes

ExNodes are the segment containers. It is unusual for an exNode to contain a single segment. It is much more likely to contain many segments organized in a hierarchy to provide more complex functionality. The default exNode, using Reed-Solomon 6+3, has 3 segments. The topmost being a cache segment to allow an arbitrary offset and length to be accessed. The cache segment will convert the arbitrary offset and length and map it into a whole set of pages that the Jerase segment can handle. The Jerase segment then breaks up the request into much smaller chunks, typically 16k, and intermingles with it a 4-byte page id which is used to detect out of sync allocations. Each of these chunks are now 16k + 4 bytes in size. This is then passed down to the LUN driver which maps it to individual allocations and issues the actual I/O operation. The *lio_signature* command can be used to probe the exNode structure. Example 6 shows the result of executing the lio_signature command on the file foo from Example 3.

Example 6. Using lio_signature to inspect the structure of an exNode.

```
root@depot1:/lio/lfs/ncsa1/testing# lio_signature foo
cache()
jerase(
   method=cauchy_good
   n_data_devs=6
   n_parity_devs=3
   chunk_size=16384
   w=-1
)
lun(
   n_devices=9
   n_shift=0
   chunk_size=16388
)
```

A.7. Putting it all together

To illustrate how the different services and layers fit together we will use the file *foo* created earlier and migrate the data from its initial location on Illinois-depot1.reddnet.org to depots at Vanderbilt, reddnet-depotXX.reddnet.org. Since we have plenty of individual depots there we will add the constraint that all allocations in a stripe reside on unique hosts. The exNode attribute is called *system.exnode*. In Example 3, we used the LIO command line tool, *lio_getattr*, but we could have just as easily used the native Linux *getfattr* command. This command translates carriage returns into their octal value "\012", so the output all fits on a line. For ease of readability, we will use the sed command to translate it back to carriage returns as shown in Example 7.

Example 7. Showing the physical location of data blocks for file foo

```
root@depot1:/lio/lfs/ncsa1/testing# getfattr -n system.exnode foo | sed -e 's/\\012/\n/g' | grep read_cap
read_cap=ibp://illinois-depot1.reddnet.org:6714/1303\\#HYZYqP8Kr084ZhCn5R4Xo-qbEBOXTWHw/1193237927050283380/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1309\\#zaX4pV7S+Rzfh+CIQF7PWBU0XU3ip5Cl/8142985625835886494/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1305\\#FLBccPC2cVdOZvw+SbIIyv5q8ZS2jZm6/1327633227782700733/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1301\\#fiskXAQtMOWzO+h8sex13RPrNaSJ-s81/1552463313520192294/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1311\\#wDHS5s320cmSff3erlKd1l14U+EN71Jn/4545713708264805990/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1310\\#bV0zRWCtoyo901YRQLTszYsECMtu5Jh/11111074597559848935/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1307\\#JMXuOjkH-wr0MYu5t2r2Ak7D2yIIiPL8/7595365900388334614/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1306\\#GxuHi-XHWTl8vEV16yrIB2Zu1TttJU2z/5873569038756943851/READ
read_cap=ibp://illinois-depot1.reddnet.org:6714/1308\\#bUqNFlXFu5qvV2NIO8IIsRE9NLNE9l6u/1467926925278802402 2/READ
```

We are just pulling out the IBP Read capabilities, which describe where the data blocks are located and how to access them. As you can see in the highlighted region of Example 7, all the data is sitting where we think it is on illinois-depot1.reddnet.org. Now let us change the RS query to use unique VU depots. The RS query is contained in the *query_default* tag for the LUN segment stanza located in the exNode. For that we have written a couple of simple scripts, change_location.sh, and query_change.sh shown in Examples 8 and 9 respectively.

Example 8. change_location.sh

```
#!/bin/bash

if [ "${1}" == "" ]; then
  echo "${0} [ncsa1|ncsa2|vandy] file"
  exit
fi;

case "${1}" in
 "ncsa1")
   qnew="simple:1:rid_key:1:any:67;1:host:1:illinois-depot1.reddnet.org:1;3:any:1:any:1"
   ;;
 "ncsa2")
   qnew="simple:1:rid_key:1:any:67;1:host:1:illinois-depot2.reddnet.org:1;3:any:1:any:1"
   ;;
 "vandy")
   qnew="simple:1:host:1:any:67;1:lun:1:vandy:1;3:any:1:any:1"
   ;;
 *)
   echo "Invalid destination"
   exit;
   ;;
esac

query_change.sh ${2} ${qnew}
```

Example 9. query_change.sh

```
#!/bin/bash

if [ "${1}" == "" ]; then
  echo "${0} file new_query"
  exit
fi;

lio_getattr -al system.exnode ${1} | sed -e "s/query_default=.*/query_default=${2}/g" > /tmp/query.$$
lio_setattr -af system.exnode=/tmp/query.$$ -p ${1}
rm /tmp/query.$$
```

Change_location.sh has three predefined RS query strings:

- nsca1 – Locate data on unique RIDS using only Illinois-reddnet1.reddnet.org.
- ncsa2 – Locate data on unique RIDS using only Illinois-reddnet2.reddnet.org
- vandy – Locate data on unique RIDS and unique hosts using the “vandy” LUN which is comprised of depots with hostnames reddnet-depotN.reddnet.org.

Based on the user option it selects one of the sites and then calls query_change.sh to make the change. Query_change.sh use the lio_getattr/lio_setattr commands along with sed to make the change. At this

point you may be wondering, “How did “foo” initially get stored on Illinois-depot1?” This is controlled by the parent directory’s exNode. Any objects created inside the directory, directly inherit the parent exNode. If you look in the LFS mount you will see three directories: nsca1, ncsa2, and vandy. Each of these directories has a different RS query controlling the data placement. Since “foo” was created in nsca1 it inherited the ncsa1 RS query string and placed the data there.

The effect of change_location.sh can be seen in Example 10.

Example 10. Simple data management using change_location.sh script.

```
root@depot1:/lio/lfs/ncsa1/testing# lio_getattr -al system.exnode foo | grep query_default
query_default=simple:1:rid_key:1:any:67;1:host:1:illinois-depot1.reddnet.org:1;3:any:1:any:1
root@depot1:/lio/lfs/ncsa1/testing# change_location.sh vandy foo
root@depot1:/lio/lfs/ncsa1/testing# lio_getattr -al system.exnode foo | grep query_default
query_default=simple:1:host:1:any:67;1:lun:1:vandy:1;3:any:1:any:1
```

In Example 10, the first command just pulls out the attribute *default_query* tag and displays it. We then execute the *change_location.sh* script, and another run another query to shows that the segment locations have changed as shown in the highlighted region. The format of the query is not important for Example 10 so the format will be discussed only briefly. The query is comprised of a collection of “,” separated tuples. The tuples are individual operations broken into the following “:” separated format:

OP:key:KEY_OP:val:VAL_OP

The OP, KEY_OP, and VAL_OP are integers representing the various operations mentioned earlier. The *key* and *val* are text string arguments that KEY_OP and VAL_OP use. For some operations these values are not used. In that case , “any” is normally used to signify this.

Now let us check things using *lio_inspect*:

Example 11. Using lio_inspect to detect an error in location.

```
root@depot1:/lio/lfs/ncsa1/testing# lio_inspect -i 20 -o inspect_quick_check foo
2605765806391400866: Inspecting file /ncsa1/testing/foo
2605765806391400866: Cache segment maps to child 7897007012983383082
7897007012983383082: jerase segment maps to child 7731792479974313281
7897007012983383082: segment information: method=cauchy_good data_devs=6 parity_devs=3 chunk_size=16384 used_size=147492 mode=1
7897007012983383082: Inspecting child segment...
7731792479974313281: segment information: n_devices=9 n_shift=0 chunk_size=16388 used_size=147492 total_size=147492 mode=1
7731792479974313281: Checking row (0, 147491, 147492)
7731792479974313281:  slun_row_size_check: 0 0 0 0 0 0 0 0 0
7731792479974313281:  slun_row_placement_check: -102 -102 -102 -102 -102 -102 -102 -102 -102
7731792479974313281: status: FAILURE (0 max dev/row lost, 0 lost, 0 repaired, 9 need moving, 0 moved)
7897007012983383082: status: FAILURE (0 devices, 0 stripes)
ERROR  Failed with file /ncsa1/testing/foo.  status=20 error_code=0
------------------------------------------------------------------
Submitted: 1   Success: 0   Fail: 1
ERROR Some files failed inspection!
```

In Example 11, we use the lio_inspect command to show more details about the file. We use the option, "*-o inspect_quick_check"*, to tell lio_inspect to check for errors, and also enabled more detailed information, "*-i 20*". The numbers on the left correspond to the segment ID of which there are 3. The last number refers to the resource allocation. Notice the slew of "-102" errors on the slun_row_placement_check. That means the data is good but it is in the wrong location, which is what the change_location.sh script did. Effectively, we told the resource allocation segment that it should be located on a Vanderbilt server, but the actual data is still stored on the illinois-depot1 data server. Correcting this mismatch is easy, and allows us to do basic data management as shown in Example 12.

Example 12. Correcting mismatched data location.

```
root@depot1:/lio/lfs/ncsa1/testing# lio_inspect -i 20 -o inspect_quick_repair foo
2605765806391400866: Inspecting file /ncsa1/testing/foo
2605765806391400866: Cache segment maps to child 7897007012983383082
7897007012983383082: jerase segment maps to child 7731792479974313281
7897007012983383082: segment information: method=cauchy_good data_devs=6 parity_devs=3 chunk_size=16384 used_size=147492 mode=4
7897007012983383082: Inspecting child segment...
7731792479974313281: segment information: n_devices=9 n_shift=0 chunk_size=16388 used_size=147492 total_size=147492 mode=4
7731792479974313281: Checking row (0, 147491, 147492)
7731792479974313281:    slun_row_size_check: 0 0 0 0 0 0 0 0 0
7731792479974313281:    slun_row_placement_check: -102 -102 -102 -102 -102 -102 -102 -102 -102
7731792479974313281:    slun_row_placement_fix: 0 0 0 0 0 0 0 0 0
7731792479974313281: status: SUCCESS (0 max dev/row lost, 0 lost, 0 repaired, 9 need moving, 9 moved)
7897007012983383082: status: SUCCESS (0 devices, 0 stripes)
Success with file /ncsa1/testing/foo!
---------------------------------------------------------------------
Submitted: 1   Success: 1   Fail: 0

root@depot1:/lio/lfs/ncsa1/testing# lio_getattr -al system.exnode foo | grep read_cap
read_cap=ibp://reddnet-depot9.reddnet.org:6714/3301\#PohVY7F1aD9rdou3m5Q71iSPscLJyPQ5/5103495629362325813/READ
read_cap=ibp://reddnet-depot1.reddnet.org:6714/3005\#xcoQuoPynFs0Frwc47MzISFo6+YPIGUQ/11519844186520566116/READ
read_cap=ibp://reddnet-depot3.reddnet.org:6714/3085\#lDJnQZUta2Sm3AI2QOtsk2VqF0ssRbSs/7281358841736548979/READ
read_cap=ibp://reddnet-depot8.reddnet.org:6714/3278\#4kSH5AVFbLeDgVsufH9NhG2eQCOTHIe4/12830054284326113551/READ
read_cap=ibp://reddnet-depot5.reddnet.org:6714/3150\#2vlq6369cK-iLFxDdtI4s6ZGJN59BrHw/10565121898315427550/READ
read_cap=ibp://reddnet-depot4.reddnet.org:6714/3124\#ZzBqKNeA0KgP0lrQmcWLZJm20lEQtZeT/2408774212259981079/READ
read_cap=ibp://reddnet-depot6.reddnet.org:6714/3187\#gh4ytNLnKGF+y75DSCz-DaZN-xbAuHtg/14267407348897555811/READ
read_cap=ibp://reddnet-depot7.reddnet.org:6714/3222\#dEbDyQ2xEBH2PRsDuisTgIqdIdxrmiWI/2302119019249640684/READ
read_cap=ibp://reddnet-depot2.reddnet.org:6714/3068\#7duw3cTjx-GyIwogs4FWIB91Pk8zGwBG/15574321931192607090/READ
```

In Example 12, we changed the inspection option from "quick-check" to "quick_repair". This causes the lio_inspect tool to move the actual data from illinois-depot1 to several Vanderbilt depots (reddnet-depots 1-9). This time you see the same "-102" error on the placement check, as highlighted in Example 11, but on the next line you see the data migration was successful which is signified by the "0's" for all the allocations in the stripe. Just to be safe we do a check of the exNode by looking at the read capabilities again using lio_getattr just like we did in the previous Example 7. The data blocks have been moved to the Vanderbilt LUN and each allocation is on a different depot, as highlighted in Example 12.

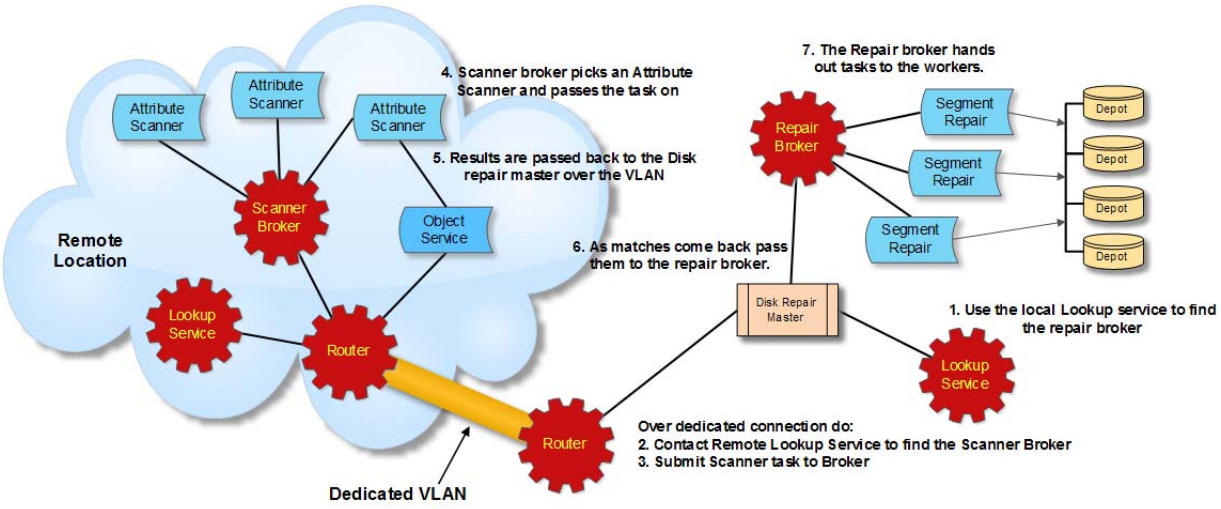


Figure 7. Example MQ data flow usage for repairing a failed drive as described in Section A.8.

A.8. Message Queue Framework (MQ)

How individual compute nodes connect and find one another greatly impacts not just the software design but also controls the scalability. Message queues were designed to provide an asynchronous communication method allowing the sender and receiver to check the queue at different times. By making the communication asynchronous it opens the door for much greater application and task concurrency. Common communication patterns also start to arise which can be templated and standardized making development much easier. The simplest pattern is a direct peer-to-peer communication. But one can make much more sophisticated patterns publish-subscribe to track types of events. Overlay networks can be created to route around network problems or move high priority traffic to a dedicated network.

RabbitMQ is currently being used for the previous generation of L-Store to perform consistency checks, warm allocations, calculate checksums, and perform drive repairs. One of the big drawbacks of RabbitMQ is its Java centric support. When writing the current version of L-Store we evaluated many of the MQ libraries available. We wanted an MQ toolkit that was lightweight, language agnostic, had a broad user base, and easy to use. Ultimately we decided on using ZeroMQ. ZeroMQ has its own idiosyncrasies, but they could be worked around. In keeping with the rest of the L-Store design, instead of directly using ZeroMQ we instead made an abstract MQ interface that the rest of L-Store uses. Behind the scenes ZeroMQ is used but the abstraction means we can replace it in the future and without having to change the whole code base. During the evaluation process we tested many patterns in ZeroMQ – heartbeating, overlay routing, publish/subscribe, and several more. This exposed many of the idiosyncrasies mentioned earlier, and we came up with work-arounds to mesh ZeroMQ into the generic task operator (GOP) framework that underpins the whole concurrency model in L-Store. This is similar to an RPC call but with task queues allowing you to consume tasks as they complete, or all at once in the end, or only process failed tasks. Nested Queues are also supported. All tasks run through the GOP framework – thread tasks, MQ tasks, and IBP operations. This makes coding things much easier since you do not have to marshal up a connection to a host or spawn a thread. You just call the particular GOP task constructor and execute it. The current version does not have all the functionality yet, namely

interior MQ nodes for routing and brokers are only prototyped. Adding them is straightforward and will be done in the near future when we decentralize the Warmer and Inspection operations as seen in **Error! Reference source not found.**.

Figure 7 is designed to provide a feel for what can be accomplished with the MQ layer. In this example data is stored at multiple sites with the OS located in a single location. The remote site has a disk failure and it needs to be repaired. Conceptually this entails doing a scan of the file system looking for any eXnodes that use the failed drive and then run the repair process on the file, lio_inspect. The remote site is in the process of doing large scale data movement so a dedicated VLAN is set up to the primary site for high priority traffic. In this case, it would be metadata operations. Below is the sequence of events that would transpire.

1. The Disk Repair Master (DRM) contacts the local MQ lookup service to see which nodes offer a broker repair service.
2. Now the DRM does the same thing at the remote site looking for an Attribute Scanner broker. Since this is high priority traffic it jumps on the dedicated VLAN using the MQ router nodes. It could go directly to the remote lookup service, but would be competing with the data transfer traffic.
3. After finding an Attribute Scanner broker the DRM submits the attribute scanning task to the broker. The broker finds a worker bee attribute scanner and assigns it the task.
4. The Attribute Scanner then executes the task: Scan the entire file system looking at the system.exnode attribute. If the exNode has the resource in question, notify the DRM.
5. Since this is high priority, the results are sent back over the dedicated VLAN.
6. The DRM then sends the filename to the Repair Broker found in step 1.
7. The Repair Broker then assigns the task to a Segment Repair worker in a round robin fashion. Additional Segment repair workers can be added to expedite the repair time.

There are many other things that can be done with the MQ layer. This is just designed as an example. One real world example, we plan on implementing, is a local caching service. The idea is to use local depots to cache frequently used data storing the augmented exNode in a local OS. This would be accomplished by encapsulating the existing exNode with a new log segment, segment_log driver, using the normal file exNode as the base. Using this approach frequently accessed data would be stored in the log on local depots. Anything not in the log would fall through to the base segment and be pulled from the remote site.

Deciding on what to cache locally would be done by instrumenting the segment read/write operations to publish any I/O operations via MQ with a Local Segment Caching service subscribed to the data stream mining it for hot spots. When a hot spot is detected it would trigger the creation of a log segment as described above and prefetch the active data. The new eXnode could then be used on subsequent opens and I/O operations.

In summary, the MQ layer provides a pluggable generic task execution framework, similar in concept to micro-services in iRODS. Micro-services in iRODS focuses on providing a client/server task execution framework which would be classified as a subset of L-Store's MQ framework.

Glossary

ACCRE -- Advanced Computing Center for Research and Education. A supercomputing facility located at Vanderbilt University. Members of the ACCRE team are the primary developers of L-Store software and are among the main authors of this document.
Allocation – Space reserved for a file on an individual depot. This information is contained in the exNode. Access to the data is controlled by read, write, and manage capabilities. Each allocation has a unique set of capabilities.
Attribute -- A keyword / value pair that is associated with a file. There are two distinct types: traditional and virtual.
CMS -- Compact Muon Solenoid experiment for the Large Hadron Collider (LHC). Vanderbilt is a Tier2 data center for CMS. The massive data storage and access needs of CMS are what continue to drive the development of the L-Store software package. The previous version of L-Store has been in full production mode with CMS for the last 3 years.
Data block – It defines how much of a given allocation is used by a particular file. A file is defined by one or more allocations.
Data Service (DS) – This service is responsible for block level data movement and storage. Currently IBP is the only implementation supported.
Depot -- A server with minimal data management software and a heterogeneous collection of hard drives.
exNode – Contains the metadata (file size, location of data on depots, etc.) associated with a given file. An exNode is the L-Store version of a Linux/Unix inode. It contains the data blocks and the information needed to reassemble the data blocks into a file.
Logistical Input/Output (LIO) -- A command line software suite for reading and writing data in L-Store. It is an interface to L-Store that allows for fine control of the data flow.
Logistical File System (LFS) -- A mountable file system that allows the user to access all of the L-Store data as if it were an external hard drive. This is the preferred method for users and applications to interact with the L-Store namespace.
Logistical Networking (LN) – A system architecture concept in which the use of storage to decouple data generation and consumption from end-to-end data transfer, generalizing the paths available through time and space.
L-Store -- The unified software namespace that connects all the depots together into a cloud. The exNodes are part of L-Store.
Metadata -- Data associated with a file that describes it in some way. This can include file size, location of the hardware resource, access time, checksum information, arbitrary tagging information etc.
Object Service (OS) -- Provides the traditional file system semantics for manipulating files and directories. It also provides support for arbitrary metadata to be associated with any object.
Reed-Solomon – A way of encoding data that is the generalization or RAID5 and RAID6 to support arbitrary numbers of drive failures. Often denoted RS-D+P where D is the number of data disks and P is the number of parity disks.
Resource Service (RS) -- is responsible for mapping resource requests to physical resources and monitoring the health of physical resources
Segment – A driver that controls data placement and layout, and optionally interactions with the rest of the services.

Segment Service -- This service consists of software drivers that control the logical layout of data on physical resources. These drivers could be designed to interface to other repositories, for example iRODS, S3, or WebDAV for example providing whole file access. They can also perform block-level I/O using the Data Service to perform sophisticated fault tolerance and caching schemes.
Traditional Attribute – Is a key value pair associated with a file and stored on the filesystem.
Vanderbilt TV News Archive – A searchable archive of network nightly news telecasts, and a major user of the L-Store storage system.
Virtual Attribute – Attribute that is calculated on demand from other attributes. This is effectively a small script that gets activated by an attribute query.

References

Atchley S.,Soltesz S., Plank J.S., and Beck M., "Video IBPster," Future Gener. Comput. Syst., vol. 19, no. 6, pp. 861-870, 2003.

Beck M., Moore T., and Plank J.S., "An End-to-end Approach to Globally Scalable Network Storage," in Proceedings of SIGCOMM 2002. Pittsburgh, PA, 2002, pp. 339-346

Beck M., et al., "Information Security on the Logistical Network: An End-to-End Approach." Proceedings of Security in Storage Workshop, 2003. SISW '03. Proceedings of the Second IEEE International, 31-31 Oct. 2003, 2003.

Bhat V., et al., "High Performance Threaded Data Streaming for Large Scale Simulations," in Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing: IEEE Computer Society, 2004, pp. 243-250.

Ding J., et al., "Remote Visualization by Browsing Image Based Databases with Logistical Networking," in Proceedings of SC2003. Phoenix, AZ, 2003.
Gill B., Bathen L., “AMP: Adaptive Multi-stream Prefetching in a Shared Cache,” ,” 5th USENIX Conference on Fil and Storage Technologies, pp. 185-198.

Huadong L., Beck M., and Jian H., "Dynamic co-scheduling of distributed computation and replication." Proceedings of Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on, 16-19 May 2006, 2006.

"Internet2 Performance Tools Case Study: REDDnet’s Distributed Storage Challenges," Internet2, pp. 2, 2009. http://www.internet2.edu/performance/200904-CS-RED.pdf.

Pinheiro E, Weber W, Barraoso L., “Failure Trends in a Large Disk Drive Population,” 5th USENIX Conference on Fil and Storage Technologies, pp. 17-28.

Plank J.S., et al., "The Internet Backplane Protocol: Storage in the Network." Proceedings of NetStore99: The Network Storage Symposium, Seattle, WA, 1999.

Plank J.S., Simmerman S, and Schuman C.D., “Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2,”, University of Tennessee Technical repot CS-08-627, August, 2008.

Plank J.S., et al., "Managing Data Storage in the Network," IEEE Internet Computing, vol. 5, no. 5, pp. 50-58, September/October, 2001.

Plank J.S., Atchley S., Ding Y., and Beck M., "Algorithms for High Performance, Wide-area Distributed File Downloads," Parallel Processing Letters, vol. 13, no. 2, pp. 207-224, June, 2003.

Schroeder B, and Gibson, G, “Disk failures In the real world: What does an MTTF of 1,000,000, hours mean to you?,” 5th USENIX Conference on Fil and Storage Technologies, pp. 1-16.