# QoS issues with the L-Store distributed file system

Alan Tackett, Bobby Brown, Laurence Dawson,
Santiago de Ledesma, Dimple Kaul, Kelly McCaulley, and Surya Pathak,
Advanced Computing Center for Research and Education,
Vanderbilt University, Nashville, TN

As the flood of data associated with leading edge instruments, sensors, and simulations continues to escalate, the challenge of supporting the distributed collaborations that depend on these huge flows becomes increasingly daunting. The chief obstacles to progress on this front lie less in the synchronous elements of the process, which have been reasonably well addressed by new global high performance networks, than in the asynchronous elements, where appropriate shared storage infrastructure seems to be lacking. To address this problem, the Advanced Computing Center for Research and Education (ACCRE) at Vanderbilt is developing a flexible storage framework called Logistical Storage (L-Store). L-Store is conceptually designed as a complete virtual file system. Designing a file system distributed over the WAN necessitates a rethinking of the traditional file system components to incorporate the more varied QoS issues that arise in the distributed environment. This paper discusses several of the different components and various QoS issues that arise.

## 1 Introduction

Storing and sharing large volumes of data across geographically separated locations is a difficult problem several different communities face today. The primary challenge is to cut across organizations, private firms, hospitals, and universities in a transparent, distributed, and secure fashion, utilizing the collective bandwidth of the internet efficiently, while also providing easy access. Starting from hospitals that deal with large amounts of sensitive patient data (security), to groups requiring billions of small files (metadata), to petabytes of high energy physics data (volume), different sets of storage and access mechanisms are required.

There are numerous QoS and fault tolerance issues when distributing a file system over the WAN that one must take into account. Below are listed several of the most prominent issues. How L-Store addresses these issues will be the focus of the rest of this paper.

**Availability.** Is the service available for use? If a portion of the network goes down it shouldn't take the whole file system with it.

**Data and metadata integrity.** Guarantees need to be in place insuring the data sent from the client and what is stored are the same. End-to-end conditioning is required.

**Performance in both metadata (transactions/s) and data transfer(MB/s).** Each community has a different blend of theses performance measurements. In the High Energy Physics community a task typically works with a few large (100MB-1GB) files but in the Proteomics world it is vastly different. Their typical data set is comprised of tens of thousands of very small (less than 1K) files. One should be able to tune the L-Store system based on the communities needs.

**Security of both metadata and raw data.** In addition to normal role based authentication and authorization some of the metadata and data stored may need to be encrypted based on the community. LN supports transfer over an SSL encrypted socket and also AES encryption of the actual data before sending.

**Fault tolerance of both metadata and data.** One can use replication to provide redundancy for metadata losses but simple replication is inefficient and cost prohibitive for the data. L-Store breaks a file up into multiple blocks that are scattered out to the various storage devices on the WAN. As a result L-Store must be able to handle not just a simple drive failure but also an entire storage appliance. We go one step further and support multiple appliance failures.

The rest of the paper describes the L-Store framework. It first starts off in section 2 discussing Logistical Networking which provides the underpinnings of L-Store. The last section discusses the L-Store architecture.

## 2 Logistical Networking

*Logistical Networking* (LN) technology was explicitly designed to solve the general problem of providing distributed storage resources for data intensive research among distributed collaborators belonging to diverse application communities. Since solving this problem means addressing the same basic issues of interoperability and deployment scalability that confronted the designers of the Internet, the developers of LN followed an architectural approach that, though applied to storage instead of bandwidth resources, closely tracks the design of the IP stack[1]. This model, which is widely recognized to be fundamental to the Internet's enduring success, has been called the "hourglass architecture"[2, 3]. The basic idea of the hourglass architecture is to build around *a highly generic common service* that mediates between the basic physical resources that need to be shared (network bandwidth, storage) and all the applications that want to use those resources. Such a common service needs, on the one hand, to be generic

enough so that it can run on almost any underlying media and can be easily implemented over new technology as it emerges. On the other, it must also be useful for almost any kind of application that builds on it, and leave application developers free to apply extra resources to adapt it for new or more specialized purposes. Like IP's generic, best effort datagram service, IBP was designed to be a highly generic, best effort storage service at the "thin waist" of LN's network storage stack (Figure 1).

### 2.1 IBP

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 [4, 5]. 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.

One can also think of IBP as providing internet level "malloc()" and "free()" calls for storage. This simple concept provides the underpinnings to meet several of the QoS requirements: *availability* – allocations can be made anywhere one has the appropriate permissions; *data transfer performance* – one can stripe a file across multiple physical storage devices to increase bandwidth; and *data fault tolerance* – for example make allocations at "off-site" locations or for storing erasure blocks.

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

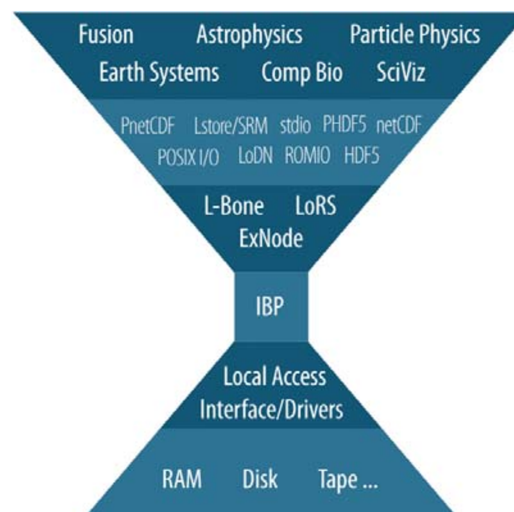


Fig. 1: Hourglass architecture for storage. IBP provides the generic common service enabling interoperability across different media (layers below) and between different middleware and applications (layers above)

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 [1]. IP implements only weak datagram delivery and leaves stronger services for end-to-end protocols higher up the stack which are required; 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 which are required for the *data integrity* QoS. 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. Below we describe the essential middleware above IBP that makes this possible.

### 2.2 exNodes: Interoperable Network Files

While IBP provides network-accessible storage, it does so in a best-effort manner to ensure scalability. Individual IBP allocations are of limited size (2 GB or less) and of limited duration (days, weeks or years as determined by the depot's owner). To mask these limitations, LN utilizes a portable file descriptor, the *exNode*. Modeled after the Unix inode, it provides a mapping from the logical extent to the physical storage and it allows for the annotation of storage. To overcome limited IBP allocation sizes, the exNode may contain multiple IBP allocations to form larger data extents. To overcome intermittent unavailability inherent in network services and support the *availability* QoS requirement, the exNode permits replication (i.e. multiple copies of the same data) on different IBP depots to improve *fault-tolerance* (Figure 2).

The exNode data structure contains the data mappings and global metadata. Each mapping contains pointers to an IBP allocation as well as mapping-level metadata and a function. The pointers or capabilities determine how the users can interact with the storage (i.e. read, write, extend the duration, delete, etc.). The mapping metadata describes how the data fits into the larger data extent. Depending on the needs of the application, the metadata may describe the mapping as a specific piece of a linear file (i.e. the metadata contains offset and length of data in the larger file), the metadata may be a set of keys in a database or matrix, or the metadata may indicate that the data is a error correction coding block (i.e. similar to a parity block in RAID-5 sets). Applications are free to define and use the metadata in any way necessary.

The global and mapping metadata is arbitrarily extensible by the user and/or application. Each metadata item is a triplet with keyword, value and type. The types include text strings, whole numbers (i.e. 64-bit integers), real numbers (i.e. 64-bit doubles), and nested lists of metadata. The function describes how the data was manipulated before it was stored. In order to use the stored data, the retrieving application will need to use the inverse of the function. The purpose of the mapping function is to allow implementation of end-to-end services such as checksums for error-detection, encryption for data security, compression for lower storage usage and faster transport, and error-coding for higher fault-tolerance while using less storage than replication. User applications may define and implement additional end-to-end services as well. Although the exNode library does not implement any of these end-to-end services, the LoRS client described below do.

While the exNode implements a network file, it is important to note that, to allow portability between users and various operating systems (OS), the exNode can be stored as an XML file. Since XML is a simple,

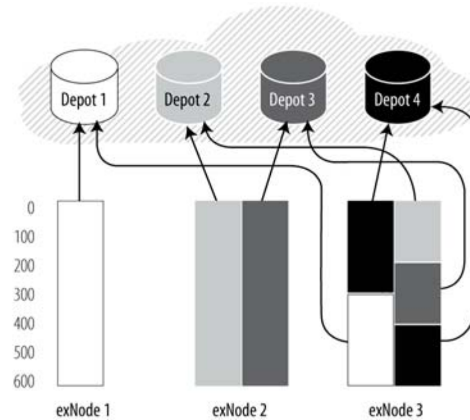


Fig. 2: Sample exNodes of a 600MB file with different replication strategies

structured text document, users can easily share it and it is does not depend on any OS-specific features. If a user sends an exNode to another person, it is as if the user has given the receiver a soft link (or shortcut in Windows or alias in Mac OS) to the file without the security concerns of giving them access to their local filesystem.

The open-source exNode library is available in both C and Java implementations [6].

### 2.3 End user tools

The Logistical Runtime System (LoRS) provides a set of command line tools (available for Windows, MacOSX, Linux and most Unix systems), a Tcl/Tk based graphical user interface (GUI), and an application programming interface (API) and library that can be used by other applications to use IBP storage. With LoRS, exNodes can easily be used to implement network files and other storage abstractions with a wide range of characteristics, such as large size (through fragmentation), fast access (through caching), and reliability (through replication). Applications requiring these characteristics can obtain them from the aggregate pool of storage resources without having to have any single IBP depot that offers them separately.

Finally, the LoRS tools employ the features of the lower levels to implement a number of end-to-end guarantees on the data, including MD5 checksums, encryption, compression, and erasure coding. This is crucial for meeting the *data integrity* QoS requirement.

## 3 L-Store

As mentioned earlier L-Store implements a complete virtual file system using LN as the underlying abstraction of distributed storage and the Chord Distributed Hash Table (DHT)[7] developed for peer-to-peer systems as a scalable mechanism for managing metadata. L-Store is designed to provide: virtually unlimited scalability in both raw storage and associated file system metadata; a decentralized management system; security; role-based authentication and

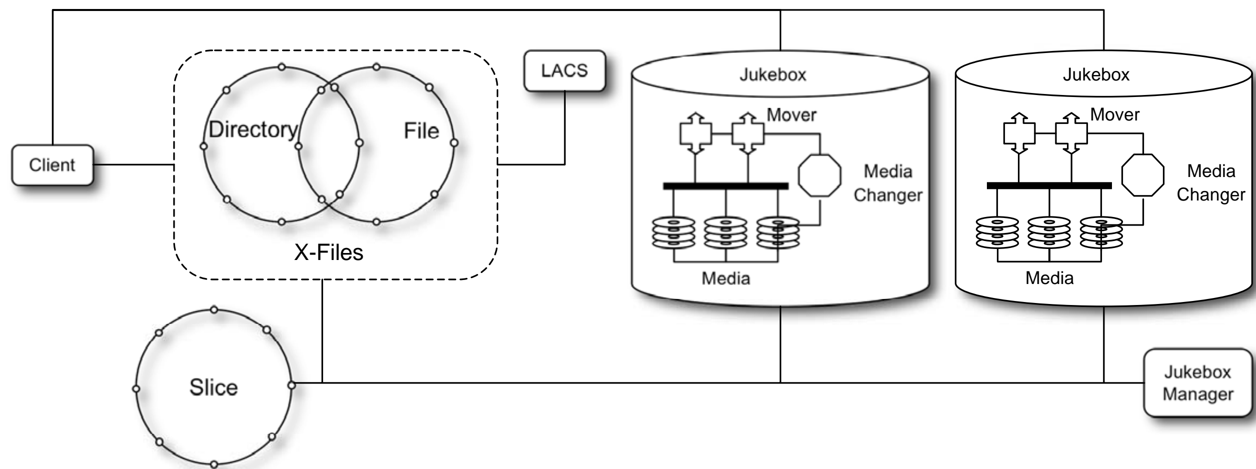


Figure 3: L-Store system architecture

authorization; policy-based data management; fault tolerant meta data support; user controlled replication and RAID-like striping of data on a file and directory level; scalable performance in both raw data movement and metadata queries; a virtual file system interface in both a web and command line form; and support for the concept of geographical locations for data migration to facilitate quicker access.

### 3.1 Distributed File System Metadata

Research in LN addresses scalability issues in the implementation of files, but not in the creation of directories and file systems. In order to create a fully scalable virtual file system, L-Store has married the LN implementation of files to the Chord distributed lookup protocol for metadata management. Chord supports a single operation: given a key, it maps the key onto a node. In our case the "key" is a hash of the directory and file name discussed later.

Chord uses a fixed length key that is a SHA-1 hash of the application key, for L-Store this is a directory or file name. The hash function evenly distributes the keys throughout the available key space and each Chord node is responsible for a fraction of the key space. In order for a Chord node to route key requests outside their assigned key range, each node needs $O(\log N)$ information about other Chord nodes, where $N$ is the number of Chord nodes. This allows the *metadata performance* (transactions/s) to scale independent of the amount of data stored. A set of Chord nodes spanning a given key space is referred to as a Chord ring. For L-Store purposes a Chord node is a process that manages a portion of the Chord key space allowing multiple Chord nodes to run on the same physical machine. A client only needs to know a Chord node on the ring to perform a key lookup.

Replication of Chord nodes in the ring is critical for keeping continuity in case of a node failure and satisfying the *availability* QoS and *meta data fault tolerance*. This is accomplished by having each node in the ring replicate its keys to one or more nodes. The collection of backup nodes for a given active node form a failure set. All the nodes in a failure set know of each other and can assume control in the event of a node failure.

### 3.2 Architecture

L-Store is designed to provide a clean functional delineation with each function designed to scale independently of the other building blocks. Figure 3 shows the L-store components and how they fit together. The best way to understand this is to see how a file is absorbed into L-Store. First each file is associated with a directory, name, list of attributes, and authentication/authorization objects. Then through the use of IBP the file is broken up into multiple

"slices" and stored on different "media."

**LACS**: The L-Store Authorization Control System is based on the "Policy Machine" paper of D.F. Ferraiolo[8]. This provides a solid framework for handling arbitrary access control policies to meet the *metadata security* QoS needs. One will notice from the diagram that a client never actually accesses the LACS servers. Instead the X-File servers cache access policies for files they own. This way LACS is only used for changes in policies.

**X-Files – File system metadata and processes:** The X-File machines have two separate Chord rings: one for handling directories and the other for the actual file information, including exNodes.

**Directory Ring:** The directory ring ascribes each directory a unique directory ID (DID). The DID is constructed by hashing the full path of the directory.

**File Ring**: After the files parent DID is resolved it is then combined with the hash of the filename (FID) and stored in the file ring: FID→DID + hash(filename).

**Slice Ring:** IBP splits a file up into multiple slices. Each of these slices is given a unique Slice ID (SID), and stored as part the file's metadata in the X-Files database. Separating the slice metadata, which include information about the storage media, from the file information gives L-Store the ability to detect hot spots on the medium and automatically replicate or move slices to eliminate the hot spot.

**Jukebox:** L-Store has intentionally separated out where the media is stored from how it is accessed. This allows one to treat tape, disk, DVD, or memory in the same way. A Jukebox is composed of three components – media, movers, and a media changer. A Jukebox can have multiple media and movers.

**Media Changer:** The media changer is responsible for keeping track of media space usage and attributes, tracking media, and setting up the connection between media and movers. The Jukebox Manager probes the Media Changers inventory for satisfying specific media attribute requests.

**Movers and Media:** A mover presents an IBP depot to the client and is strictly for moving data to and from the different media. For directly-attached disks there would typically be a fixed mover; for example, the machine attached to the disk. In SAN environments, the relationship between mover and media can be more dynamic.

**Jukebox Manager:** The Jukebox Manager is designed to keep track of each media items' different attributes. There can be multiple Jukebox Managers, each keeping track of one or more Jukeboxes. A Jukebox Manager handles requests for different attributes and space requirements made for all "write" operations.

## 3.3 Extended Attributes

Traditional file and directory permissions – read, write, and execute for the user, group, and others – do not take into account the distributed storage infrastructure L-Store provides. For this reason we have extended the usual list to support these additional attributes on a per file or per directory basis: replication across multiple sites; erasure encoding of data for fault tolerance; maximum storage duration; content addressable storage for slices; and storage locality coordinates. These attributes can be set on a per-directory basis, with each uploaded file inheriting the attributes from the directory unless a per-file value is explicitly specified.

**Dataflow:** This feature allows for the migration of data across sites and media, maintaining copies as needed at intermediate storage locations. For example, a user can choose to keep files on nearby disks and after a specified time automatically migrate the data to tape for archiving. Likewise data can be migrated around the country to different sites, each one performing a separate analysis, before allowing the data to migrate on to the next site. This can also be used to route data through multiple sites using multiple paths in order to obtain higher bandwidth utilization and fault tolerance.

**Links between different L-Store systems:** Most file systems support the concept of soft links between directories. This is an extremely useful feature that is easily accommodated in L-Store.

**Locality:** In order to maximize throughput one strives to place the data near where it is being accessed. So, what constitutes "nearby"? One could use something like the Network Weather Service, which provides bandwidth and latency information, or one could use the geographical location of the DNS owner. Most of the time one is not concerned with whether two machines are in the same data center or across the country. What is critical is the latency between the machines. In a POP it is normal to have adjacent machines on different networks resulting in a large latency.

## 3.4 Data fault tolerance

Due to the distributed nature of the data it is critical the system support a much higher *data fault tolerance* than a simple RAID-5 or RAID-6 drive failure. At the very least network hiccups can cause multiple storage appliances to "disappear" making the files unavailable. As a result L-Store is designed to handle the complete failure of multiple storage appliances. This is accomplished using the Weaver erasure codes[9]. These erasure codes are simple to implement and have the same efficiency as mirroring (50%). The table provided in the reference supports up to 10 failures. The full table is implemented in L-Store along with the normal LoRS options of replication and striping. The user decides how critical the their data is on a file-by-file basis. Encoding and reconstruction are simple $O(n)$ operations where $n$ is the fault tolerance. Data and parity are kept separately so there is no decoding cost for reading data. As a result all encoding and reconstruction are done on the storage depots under the guidance of L-Store. This is accomplished using the Network Functioning Unit (NFU) supported in the IBP depot.


[1] M. Beck, T. Moore, and J. S. Plank, "An End-to-end Approach to Globally Scalable Network Storage." In Proceedings of SIGCOMM 2002, Pittsburgh, PA, August 19-23, 2002.

[2] L. Kleinrock and e. *al*, "Realizing the Information Future, The Internet and Beyond," NRENAISSANCE Committee Computer Science and Telecommunications Board, Washington D.C., 1994.

[3] L. Peterson and B. Davie, *Computer networks: a systems approach*, 3rd ed. Boston: Morgan Kaufmann Publishers, 2003.

[4] A. Bassi, et al., "The Internet Backplane Protocol: A Study in Resource Sharing," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002).* Berlin, Germany: IEEE, 2002.

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

[6] *exNode Library*, http://loci.cs.utk.edu/modules.php?name=Downloads&d_op=getit&lid=142, 2006.

[7] F. Dabek, et al., "Building Peer-to-Peer Systems With Chord, a Distributed Lookup Service." In Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), May, 2001.

[8] D. Ferraiolo, S. Gavrila, V. Hu, and D. R. Kuhn, "Composing and Combining Policies under the Policy Machine." In Proceedings of the ACM SACMAT 2005, Stockholm, Sweden, June 1-3, 2005.

[9] J. L. Hafner, "WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems," *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December, 2005, http://www.usenix.org/events/fast05.