The Logistical File System: A Network File System Designed for Scalable Resource Sharing

Alex Bassi, Micah Beck, Erika Fuentes, Terry Moore, James S. Plank
Logistical Computing and Internetworking Laboratory
University of Tennessee

Abstract — This paper describes the technologies and design ideas that underlie the *Logistical File Systems* (*LoFS*). LoFS is close to a traditional distributed file system in structure and in the class of operations it supports, but it is designed to preserve the easy deployability and scalability across the administrative boundaries that have been the pillars of the Web's success. The leading idea behind the design of LoFS is that in order to implement a real file system that nonetheless preserves the strengths of the Internet of model resource sharing, one has to apply that model to the storage resources needed to implement file system operations, so that they are exposed and shareable to the global network. Systems that do not expose the underlying resources used to implement file system operations can implement remote access to file system operations, but they cannot distribute many important functions of the file system itself.

1. Introduction

After more than a decade of unprecedented growth, the World Wide Web has transformed the landscape of networked information systems. It has easily dominated the arena of wide area information sharing that seemed, at the beginning of the 90's, to be the natural domain of more capable and mature file sharing technologies, such as the Andrew File System [1]. This development was all the more surprising since comparison shows that the weaknesses of the Web's design (e.g. limited *ad hoc* caching capabilities, little or no security, lack of scalability in important dimensions, tendencies to performance degradation, etc.) were significant, and many of its strengths (e.g. free distribution, intuitive user interface) could be (and have been) duplicated by more capable competitors [2]. But what could not have been duplicated, and the key, we believe, to the Web's unparalleled success, was the *Internet model of resource sharing* at the base of its design

As evidenced in structure the IP stack, the Internet Model was created to facilitate the sharing of network bandwidth for the purpose universal communication among an international community of indefinite size, and is therefore designed to be as open (i.e. lightly controlled) and easy to use as possible. It's very weaknesses, according to some established criteria, have had the compensating virtue of making it easy to deploy and highly scalable across the administrative boundaries of traditional systems. The Web builds on this same foundation. The fundamental contribution of the Web over previous networked information systems was not the notion of the URL as a file name with globally uniform semantics (AFS and other distributed file systems have that feature), but the use of such names in hyperlinks that were globally valid across *administrative boundaries*. When the Web was introduced users found that if they could put up a Web server (and this was very easy to do), *anyone* could link to the files they wanted to share and vice versa. It seemed as if the Web's "fuzzy pointers" could point anywhere.

The universality and generality that this model to resource sharing achieves has a power that is undeniable. But in order to achieve it, the Web architecture makes a series of compromises that from the traditional systems point of view seem drastic:

1. The default semantics of URLs are that they represent unprotected, read-only data. The protection mechanisms that have been introduced are based on passwords and certificates, and so are not inherently limited in administrative scope.
2. Local interpretation of URLs exposes a specific portion of the host directory structure to the network.
3. The Web defines a restricted set of operations on URLs that can be implemented in a globally scalable network.

The thesis of this paper is that a different set of design choices can be made that results in a distributed file system that can scale up to the global Internet, but without accepting the compromises accepted by the Web. We believe that the file system that results, which we call the *Logistical File System* (*LoFS*) is

closer to traditional distributed file systems in structure and in the class of operations it supports, but preserves the easy deployability and scalability across the administrative boundaries that have been the pillars of the Web's success. We call this file system "logistical" because it builds on our work in *logistical networking*, which is described elsewhere [3].

The leading idea behind the design of LoFS is that in order to implement a real file system that nonetheless preserves the strengths of the Internet model, one has to apply the Internet model to *the resources (primarily storage) needed to implement file system operations, so that they are exposed and shareable to the global network*. Systems that do not expose the underlying resources used to implement file system operations can implement remote access to file system operations, but they cannot distribute many important functions of the file system itself.

Accordingly, in designing LoFS we have followed a "bottom up" design philosophy that today is more familiar in design of network protocol stacks than in the design of operating systems. At the bottom of the "network storage stack" is the *Internet Backplane Protocol (IBP)*, which is a mechanism created to enable sharing of exposed storage resources across the network on the Internet paradigm. Since, as we describe below, IBP uses a model of storage with weak semantics, in order to support the kind of strong file abstraction that LoFS requires, we have developed a data structure represents *aggregate* storage resources and allows us to layer a file abstraction with strong semantic properties on top of a weak underlying storage resource that does not generally provide them. We call this data structure the *exNode* because it is analogous to the Unix *inode*, but scoped for the wide area network. Finally, the top layer of the network storage stack is LoFS, which is designed to be a log-based file system that can leverage the power of Internet resource sharing that the lower layers make available. The discussion below follows the same bottom up design as the technology being described, explaining each layer in turn and providing details of applications and experiences with these technologies.

2. Background: The Internet Protocol and the Internet Backplane Protocol

The unique capabilities of LoFS will derive from the foundation of exposed resource sharing on which it builds, i.e. from the Internet Backplane Protocol. IBP is a mechanism developed for the purpose of sharing storage resources across networks ranging from rack-mounted clusters in a single machine room to global networks. [3-5] To approximate the openness of the Internet paradigm for the case of storage, the design of IBP parallels key aspects of the design of IP, in particular IP datagram delivery. This service is based on packet delivery at the link level, but with more powerful and abstract features that allow it to scale globally. Its leading feature is the independence of IP datagrams from the attributes of the particular link layer, which is established as follows:

- Aggregation of link layer packets masks its limits on packet size;
- Fault detection with a single, simple failure model (faulty datagrams are dropped) masks the variety of different failure modes;
- Global addressing masks the difference between local area network addressing schemes and masks the local network's reconfiguration.

This higher level of abstraction allows a uniform IP model to be applied to network resources globally, and it is crucial to creating the most important difference between link layer packet delivery and IP datagram service. Namely,

> *Any participant in a routed IP network can make use of any link layer connection in the network regardless of who owns it. Routers aggregate individual link layer connections to create a global communication service.*

This IP-based aggregation of locally provisioned, link layer resources for the common purpose of universal connectivity constitutes the form of sharing that has made the Internet the foundation for a global information infrastructure.

IBP is designed to enable the sharing of storage resources within a community in much the same manner. Just as IP is a more abstract service based on link-layer datagram delivery, IBP is a more abstract service based on blocks of data (on disk, tape or other media) that are managed as "byte arrays." The independence of IBP byte arrays from the attributes of the particular *access layer* (which is our term for storage service at the local level) is established as follows:

- Aggregation of access layer blocks masks the fixed block size;
- Fault detection with a very simple failure model (faulty byte arrays are discarded) masks the variety of different failure modes;
- Global addressing based on global IP addresses masks the difference between access layer addressing schemes.

This higher level of abstraction allows a uniform IBP model to be applied to storage resources globally, and this is essential to creating the most important difference between access layer block storage and IBP byte array service:

> *Any participant in an IBP network can make use of any access layer storage resource in the network regardless of who owns it. The use of IP networking to access IBP storage resources creates a global storage service.*

Whatever the strengths of this application of the IP paradigm, however, it leads directly to two problems. First, in the case of storage, the chronic vulnerability of IP networks to Denial of Use (DoU) attacks is greatly amplified. The free sharing of communication within a routed IP network leaves every local network open to being overwhelmed by traffic from the wide area network, and consequently open to the unfortunate possibility of DoU from the network. While DoU attacks in the Internet can be detected and corrected, they cannot be effectively avoided. Yet this problem is not debilitating for two reasons: on the one hand, each datagram sent over a link uses only a tiny portion of the capacity of that link, so that DoU attacks require constant sending from multiple sources; on the other hand, monopolizing remote communication resources cannot profit the attacker in any way, it can only harm the victim. Unfortunately neither of these factors hold true for access layer storage resources. Once a data block is written to a storage medium, it occupies that portion of the medium until it is deallocated, so no constant sending is required. Moreover it is clear that monopolizing remote storage resources can be very profitable for an attacker and his applications.

The second problem with sharing storage network-style is that the usual definition of a storage service is based on processor-attached storage, and so it includes strong semantics (near-perfect reliability and availability) that are difficult to implement in the wide area network. Even in "storage area" or local area networks, these strong semantics can be difficult to implement and are a common cause of error conditions. When extended to the wide area, it becomes impossible to support such strong guarantees for storage access.

We have addressed both of these issues through special characteristics of the way IBP allocates storage:

- *Allocations of storage in IBP can be time limited.* When the lease on an allocation expires, the storage resource can be reused and all data structures associated with it can be deleted. An IBP allocation can 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. Forcing time limits puts transience into storage allocation, giving it some of the fluidity of datagram delivery.
- *The semantics of IBP storage allocation are weaker than the typical storage service*. Chosen to model storage accessed over the network, it is assumed that an IBP storage resource can be transiently unavailable. Since the user of remote storage resources is depending on so many uncontrolled remote variables, it may necessary to assume that storage can be permanently lost. Thus, *IBP is a "best effort" storage service*. To encourage the sharing of idle resources, IBP even

supports "volatile" storage allocation semantics, where allocated storage can be revoked at any time. In all cases such weak semantics mean that the level of service must be characterized statistically.

Because of IBP's limitations on the size and duration of allocation and its weak allocation semantics, IBP does not directly implement reliable storage abstractions such as conventional files. Instead these must be built on top of IBP using techniques such as redundant storage, much as TCP builds on IP's unreliable datagram delivery in order to provide reliable transport.

3. The Internet Backplane Protocol

3.1 The IBP API

The IBP API is in many respects typical of network file systems, with calls for allocation, access and management as summarized in Table 1 below and discussed in some detail in the API documentation [5]. The unique aspects of IBP are reflected most directly in the IBP_allocate call.

In most conventional file systems, file creation entails the creation of an entry in a file directory under a client-supplied name. The directory entry represents the ability to indefinitely allocate data space through write operations, generally up to some limit imposed either by the system or on a per-user basis, and to later read from that data space. In some specialized file systems, file attributes such as physical layout or staging policy between disk and tape can be specified [6-8].

The IBP_allocate call differs in a number of ways. First of all, it is much like a typical memory allocation operation such as the C library malloc in that it allocates writable storage space but does not create a visible directory entry. It returns a set of capabilities that can be used as an opaque credential for later read, write and management operations on the allocated space from any Internet-connected client. Most important, it allows the specification of attributes that model a number of uses for storage other than conventional file space.

- A number of different write semantics are available: append, truncate, FIFO queue, circular queue. Note that there is currently no support for overwriting other than by truncating the entire file to zero length, as this requires external synchronization when shared among multiple writers. These write semantics support non-file applications such as the implementation of Unix-like pipes in the network. To reflect this generality, we refer to the space allocated as a byte array to reflect that fact that it is more general than either a file or a communication buffer.
- Allocations can be weakened in a number of ways reflecting a more lightweight approach to permanence and reliability than is typical for network file systems. In particular, allocations may be time-limited, representing a lease that expires at some known point in the future, and they may be volatile, meaning that they represent an allocation of free space that the server can revoke at any point in the future. It is intended that by allowing depots to grant weaker allocations while remaining within the bounds of correct functioning, IBP will enable the sharing of resources that would otherwise be held for private use only.

The other IBP API calls fall into two groups: those that operate on an IBP allocation, and one that operates on an IBP depot.

1. **Operations on allocations**. **`IBP_allocate`** returns a set of three capabilities: a read capability, a write capability and a manage capability. Each of these are required for different subsequent API calls.
    a. **`IBP_store, IBP_load, IBP_copy, IBP_mcopy.`** The **`IBP_store`** and **`IBP_load`** are syncrhonous read and write operations that return when the requested data transfer has occurred or when a specified timeout expires. They take a write and read capability as argument respectively. **`IBP_copy`** allows a third-party copy between depots without requiring that the data be retrieved by the client, and takes both a read and a write

capability as arguments. The **IBP_mcopy** call models generalized point-to-multipoint communication. It takes as argument a single read capability (source) and a set of write capabilities (destinations) as well as an "operation" parameter and a set of operation-specific arguments. These operations are implemented by depot "plug-in" modules called "data movers" (see section 3) and are intended to support flexible exploration of new and non-standard ways of transferring data between endpoints[9].

b. **`IBP_manage`** takes the manage capability as an argument, and implements increment and decrement operations on two reference counts maintained for each allocations: a read count and a write count. If the write count reaches zero, the allocation can be treated as read-only, and if the read count reaches zero, the allocation can be deleted. This call is also used to query the state of an allocation, and to request modification of some of its basic characteristics (such as extending the lease).

2. **Depot management** calls require no capability, but can be protected by password.
   a. **`IBP_status`** has two sub-commands: **`inquire`** and **`change`**. **`Inquire`** allows a client to query a depot about its total stable and volatile storage, the amount of both storage categories used, and the maximum allowed duration. **`Change`** allows the client to change these parameters.

Storage Management	Data Transfer	Depot Management
`IBP_allocate, IBP_manage`	`IBP_store, IBP_load IBP_copy, IBP_mcopy`	`IBP_status`

Table 1

3.2 IBP Implementation

Depot — The main IBP depot architecture goals are flexibility, reliability and performance. The current implementation (1.0) is multi-threaded for performance, with a pool of threads created at boot time. The code base is shared between Unix/Linux/OS X and Win32 versions, with OS-specific I/O calls encapsulated in a library that has two implementations (win-lib and unix-lib).

Client Library — The IBP Client Library, offered in a few different versions and systems, was designed to be flexible, to ease the implementation of future changes to both the API and the protocol, to be very maintainable code and to be extremely robust and fault-tolerant. The library is sepearted into two different modules: the API2P Module and the Communication Module (ComModule). API2P translates the API command into Communication Units, which are abstract data types that specify the communication and its characteristics (direction, semantics of the message, the message itself or the expected message). Then, the ComModule allows the execution of the communication. No analysis of the message is made at this level, the API2P module being responsible to interpret the message and to take the appropriate action. This design allows easy changes to the API (as it's seen as a sequence of communication units) and to the protocol. On top of it, the communication module, being completely independent, will be re-used in future depot implementations, cutting developing time.

Protocol — The current version of our protocol is a very direct encoding of the API as an architecture-independent RCP using a per-call TCP connection. As more challenging requirements arise, we are considering a redesign of this protocol, perhaps using some of the protocol encapsulation tools, such as BXXP, that are currently under discussion for standardization with the IETF [10].

Testing — A flexible test language and interpreter were developed to enable extensive testing and performance measurement of the depot by both developers and users. These tools [11] allow a complete and extensive test of our software and of the protocol semantics.

4. Data Movers

Since the primary intent of IBP is to provide a common abstraction of storage, it is arguable that third party transfer of data between depots is unnecessary. Indeed, it is logically possible to build an external service for moving data between depots that access IBP allocations using only the `IBP_load` and `IBP_store` calls. However, such a service would have to act as a proxy for clients, and this immediately raises trust and performance concerns. The `IBP_copy` and `IBP_mcopy` data movement calls were provided in order to allow a simple implementation that avoids these concerns. However, software architectures based on external data movement operations are still of great interest to us.

The intent of the basic `IBP_copy` call is to provide access to a simple data transfer over a TCP stream between depots. `IBP_mcopy` is a more general facility, and can provide access to operations that range from simple variants on simple TCP-based data transfer to highly complex protocols using multicast or other advanced network facilities. In all cases, the caller is responsible for determining whether the requested operation is appropriate to the depot's network environment, and for any error strategy should the data movement call return in failure.

The data mover is a plug-in module to an IBP depot that is activated either by an **`IBP_mcopy`** call or by an **`IBP_datamover`** call. The second call is not an API call, but an internal call made by the sending IBP depot. The sending depot is responsible for invoking a Data Mover plug-in on the receiving depot, and it accomplishes this by **`fork`**ing a data mover control process that sends an **`IBP_datamover`** request, causing the receiving depot to **`fork`** a symmetric data mover control. Sending and receiving control processes then **`exec`** the appropriate Data Mover plug-ins for the requested operation and these cooperate to perform the operation, then the plug-in at the sending depot replies to the client and then both plug-ins terminate. The figure 1 illustrates this process.

The Data Mover software architecture can support a wide variety of operations, including:

- Point-to-multipoint through simple iterated TCP unicast transfers
- Point-to-multipoint through simultaneous threaded TCP unicast transfers.
- Unreliable UDP point-to-mulitpoint utilizing native IP mulitcast
- Reliable point-to-multipoint utilizing native IP multicast
- Fast, reliable UDP data transfer over private network links [12]

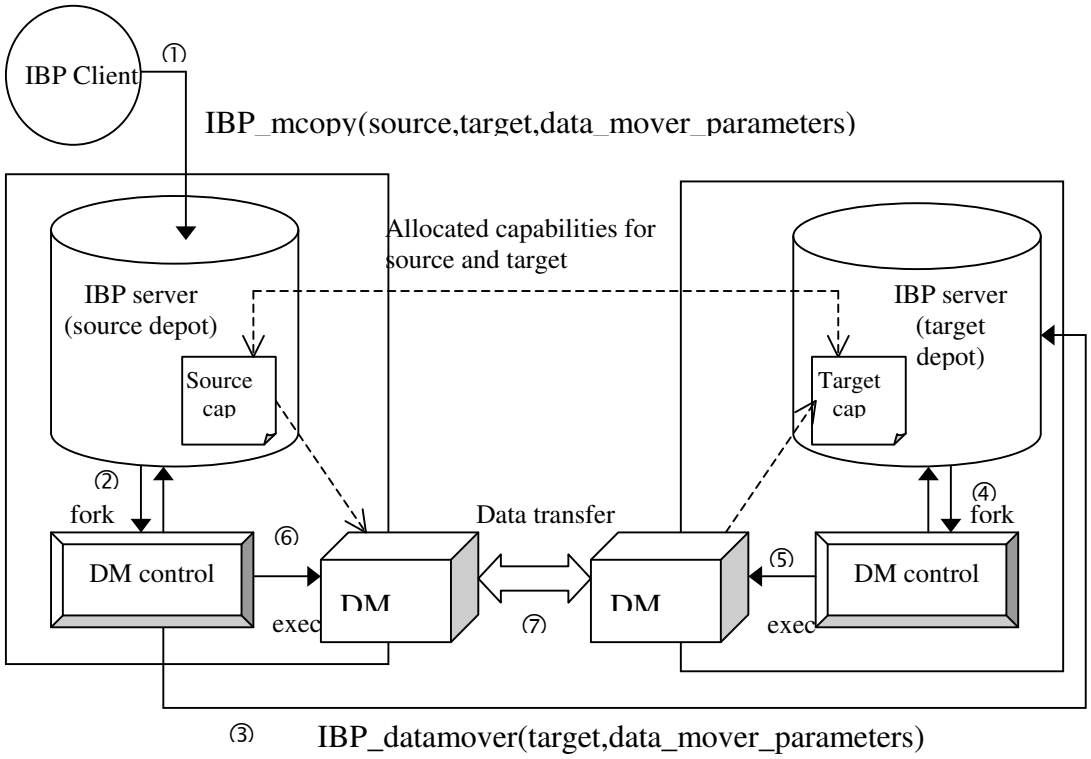

Figure 1: Data Mover Control and Transfer Coordination.

5. Experience and Applications

Our method in developing the Internet Backplane Protocol is based on implementation and experimentation. A number of simple applications from within our own research group have formed the basis of the experience that has guided our work. In addition, a few external application groups have picked up on early implementations of IBP and contribute by both tracking our releases and giving us needed feedback for future developments. However, it is only with the upcoming release of the exNode library and serialization (Section 6) and ultimately the Logistical File System (Section 7) that we believe a wide application community will find the supporting tools necessary to adopt an IBP-based methodology.

- IBP-mail is a system that uses IBP to transmit and deliver mail attachments that require storage resources beyond the capacity of standard mail servers[13]. To use IBP-Mail, the sender first uploads the attachment into a suitable IBP server and then forwards the read capability to the receiver, who can use it to download the attachment (Figure qqq). An initial version of IBP-mail used custom CGI scripts to chose the depot and implement the upload and download functions. The capability was transmitted between sender and receiver embedded in a form that invoked the receiving CGI. This custom architecture has been replaced by a generic mechanism for storing IBP capabilities (see section 6 on the exNode) which allows them to be sent between sender and receiver as a simple file attachment in a standard serialized format. The manipulation of files stored in IBP format is now handled through generic tools operating on the serialized exNode data structure, and so today IBP-Mail is not so much an application as a way of using those standard tools together with the standard MIME attachment facility in e-mail.
- NetSolve is a distributed computation tool created by Cassanova and Dongarra [14] to provide remote invocation of numerical libraries for scientific code. One shortcoming of the initial NetSolve architecture is that it is stateless: a series of calls to the same server cannot, for instance,

cache arguments or results in order to avoid unnecessary data movement in subsequent calls. One of the approaches taken to optimize NetSolve was to use an IBP depot at the server to implement a cache under the control of the client. Such short-lived use of IBP for value caching is able to make good use of even volatile and time-limited allocations [15].

- TAMANOIR [16] is a project developed by the RESAM lab of the Lyon I University in the field of Active Networking. It is a framework that allows users to easily deploy and maintain distributed active routers in a wide area network. IBP depots will be among the standard tools available to services implemented within the TAMANOIR framework, along with othe basic tools such as the routing manager and stream monitoring tools. It will also be used to implement distribution and caching of services (distributed as Java byte-code modules) that are loaded by TAMANOIR on-demand, freeing TAMANOIR to manage only is own internal cache of services.

6. The exNode: Aggregating IBP Storage Resources to Provide File Services

Our approach to creating a strong file abstraction on the weak model of storage offered by IBP continues to parallel the design paradigm of the traditional network stack. In the world of end-to-end packet delivery, it has long been understood that TCP, a protocol with strong semantic properties (e.g., reliability and in-order delivery) can be layered on top of IP, a weak datagram delivery mechanism. In spite of the weak properties of IP datagram delivery, stronger properties like reliability and in-order delivery of packets can be achieved through the fundamental mechanism of retransmitting IP packets. Retransmission controlled by a higher layer protocol, combined with protocol state maintained at the endpoints, overcomes non-delivery of packets. All non-transient conditions that interrupt the reliable, in-order flow of packets can then be reduced to non-delivery. We view retransmission as an *aggregation* of weak IP datagram delivery services to implement a stronger TCP connection.

The same principle of aggregation can be applied in order to layer a storage service with strong semantic properties on top of a weak underlying storage resource that does not generally provide them, such as an IBP depot. Examples of aggregating weaker storage services in order to implement stronger ones include the following:

- Reliability — Redundant storage of information on resources that fail independently can implement reliability (e.g. RAID, backups).
- Fast access — Redundant storage of information on resources in different localities can implement high performance access through proximity (e.g. caching) or through the use of multiple data paths (e.g. RAID [6]).
- Unbounded allocation — Fragmentation of a large allocation across multiple storage resources can implement allocations of unbounded size (e.g. files built out of distributed disk blocks, databases split across disks).
- Unbounded duration — Movement of data between resources as allocations expire can implement allocations of unbounded duration (e.g. migration of data between generations of tape archive).

In this exposed-resource paradigm, implementing a file abstraction with strong properties involves creating a construct at a higher layer that aggregates more primitive IBP byte-arrays below it. To apply the principle of aggregation to exposed storage services, however, it is necessary *to maintain state that represents such an aggregation of storage allocations*, just as sequence numbers and timers are maintained to keep track of the state of a TCP session. Fortunately we have a traditional, well-understood model to follow in representing the state of aggregate storage allocations. In the Unix file system, the data structure used to implement aggregation of underlying disk blocks is the *inode* (*intermediate node*). Under Unix, a file is implemented as a tree of disk blocks with data blocks at the leaves. The intermediate nodes of this tree are the inodes, which are themselves stored on disk. The Unix inode implements only the aggregation of disk blocks within a single disk volume to create large files; other strong properties are sometimes implemented through aggregation at a lower level (e.g. RAID) or through modifications to the

file system or additional software layers that make redundant allocations and maintain additional state (e.g. AFS, HPSS) [1, 8].

Following the example of the inode, we have chosen to implement a single generalized data structure, which we call an *external node,* or *exNode*, in order to manage of aggregate allocations that can be used in implementing network storage with many different strong semantic properties (Figure 2). Rather than aggregating blocks on a single disk volume, the exNode aggregates storage allocations on the Internet, and the exposed nature of IBP makes IBP byte-arrays exceptionally well adapted to such aggregations. In the present context the key point about the design of the exNode is that it has allowed us to create an abstraction of a network file to layer over IBP-based storage in a way that is completely consistent with the exposed resource approach.

We plan to use the exNode as the basis for a set of generic tools for implementing files with a range of characteristics. Because the exNode must provide interoperability between heterogeneous nodes on a diverse Internet, we have chosen not to specify it as a language-specific data structure, but as an abstract data type with an XML serialization. The basis of the exNode is a single allocation, represented by an Internet resource, which initially will be either an IBP capability or a URL. Other classes of underlying storage resources can be added for extensibility and interoperability.

The important elements to be developed are libraries that implement generic requirements such as large size (through fragmentation), fast access (through caching), and reliability (through replication). Applications requiring these characteristics should be able to obtain them even without having available individual IBP depots that implement those specific characteristics – simply using the APIs should be sufficient if aggregate resources that are available for use somewhere on the network. The exNode data structure will be a basis for interoperability within the logistical networking API, and the XML serialization will be the basis of interoperability between network nodes.

Since our intent is to use the exNode file abstraction in a number of different applications, we have chosen to express the exNode concretely as an encoding of storage resources (URLs or IBP capabilities) and associated metadata in XML. If the exNode is placed in a directory, the file it implements can be imbedded in a namespace. But if the exNode is sent as a mail attachment, there need not be a canonical location for it. The use of the exNode by varying applications will provide interoperability similar to being attached to the same network file system.

The exNode metadata must be capable of expressing at least the following relationships between the file it implements and the storage resources that constitute the data component of the files state:

- The portion of the file extent implemented by a particular resource (starting offset and ending offset in bytes)
- The service attributes of each constituent storage resource (e.g. reliability and performance metrics, duration)
- The total set of storage resources which implement the file and the aggregating function (e.g. simple union, parity storage scheme)

Despite our emphasis on using an exposed-resource approach, it is natural to have the exNode support access to storage resources via URLs, both for the sake of backward compatibility and because the Internet is so prodigiously supplied with them. It is important to note, however, that the flexibility of a file implemented by the exNode is a function of the flexibility of the underlying storage resources. The value of IBP does not consist in the fact that it is the only storage resource that can be aggregated in an exNode, but rather that it is by far the most flexible and most easily deployed.

6.1 The exNode API

The exNode API is a standard interface for creating, communicating and manipulating the exNode data structure.

- **xnd_create**, **xnd_destroy** are standard data structure constructor/destructor operations.
- **xnd_serialize**, **xnd_deserialize** write/read the standard XML serialization of the exNode data structure to/from a file descriptor (see section 6.2).
- **xnd_add_mapping**, **xnd_delete_mapping** add/delete a mapping from the exNode.
- **xnd_query**, **xnd_enum_next**, **xnd_enum_end**, **xnd_build_exNode** are query operations. **xnd_query** returns the a enumeration data structure representing the set of mappings whose range intersects with a specified target range. The enumeration can be traversed using xnd_enum_next or destroyed using **xnd_enum_end**. **xnd_build_exNode** creates a new exNode from the set of mappings that comprise an enumeration.
- **xnd_size** returns the aggregate extent of all the mappings in an exNode.

This minimal exNode API can be extended in a number of ways that have been left out of this account for the sake of clarity, and to keep from having to introduce additional structure. Some of these extensions include:

- Queries can be much more complex, specifying ranges of data and time, and returning sets of storage resources with associated metadata to direct the process of retrieving data.
- Mappings can be annotated to specify read-only or write-only data.
- As storage allocations expire or become unavailable it will be necessary to manage the exNode by finding and deleting mappings, and this will require additional mapping management calls.
- By associating a mapping with a set of storage specifiers and an aggregation function, it is possible to model group allocations such as RAID-like error correction.
- By defining metrics on the location or performance or other characteristics of different storage allocations it is possible to inform the user of the exNode which of multiple alternatives to choose.

6.2 The exNode XML Serialization

The mobility of the exNode is based on two premises:

1. it is possible to populate the exNode exclusively with network-accessible storage resources
2. the exNode can be encoded in a portable way that can be interpreted at any node in the network

Today, XML is the standard tool used to implement portable encoding of structured data, and so we are defining a standard XML serialization of the exNode. The serialization is based on the abstract exNode data structure, and so allows each node or application to define its own local data structure.

7. LoFS: The Logistical File System

A simple IBP-based file system that implements a directory structure and data storage completely within IBP has been developed [17], using an *ad hoc* modified Apache web server to act as the trial application that accesses its source files through it. In the realm of this experiment, we restricted updates to complete replacement of a file, allowing atomic updates to be implemented through the directory. A very interesting feature of this project can be identified in the possibility of having a local IBP depot as data cache, to improve performance. The preliminary tests results show a good potential, but more tests need to be conducted in order to have valid results.

Our Apache-based file system was a prototype on which to test the IBP implementation and robustness. A true distributed file system, built on wide-area resources, must have at its core the ability to deal with the commonplace occurences of failures and unbounded network latencies. Our implementation will be based on two core functionalities:

- The exNode as the main metadata type.
- A log-structured approach to storing data

The exNode is explained above. Log-structured file systems (LFS's) were invented in the late 80's as a way to improve the performance of file system writes[18, 19]. Instead of overwriting data in place, which results in disk writes that can be scattered across the disk, data is appended to logs, which are flushed to disk *en masse*, resulting in a more efficient use of contiguous disk blocks. This of course results in dead data spread throughout the filesystem, which must be reclaimed by a separate *gargabe collector* process. Many implementations of LFS's demonstrated improved overall filesystem performance[20, 21].

As an unexpected benefit, LFS's were found to have other desirable properties, such as extremely efficient failure recovery[19], the ability to deal smoothly with compression on the storage substrate[22], and the ability to ease synchronization worries when replication is added to the file system[23]. This latter property is what makes LFS's attractive to LoFS.

Recently, a storage system called Swarm has been developed at the University of Arizona[24], which implements a *storage server* layer, intended to be a layer between raw or network-attached storage, and file system clients. With Swarm, clients produce append-only logs as in a LFS, which then become striped across multiple storage servers with RAID Level 5 parity encoding to tolerate the loss of any server in the collection. Like in a typical LFS, file updates result in log records which are appended to new logs, and the updated data eventually becomes cleaned by a cleaner thread.

This structure has the basic elements needed to implement file systems on the wide area, since it avoids the synchronization problems (typically solved by holding locks) of updating data in place, and the log-based structure eases the task of rebuilding state following a failure. It is a storage service, however, and not a full-blown file system because it leaves issues of naming, sharing and security up to the clients. The Swarm researchers have built prototype file systems for Swarm on a local area network[25].

We plan to start with the methodology of Swarm as our base design for LoFS, using the exNode data structure as the basic metadata block. Beneath this foundation, we will use IBP servers to provide the basic storage services, taking the place of standard disks and network attached storage devices. On top of this foundation, we must incorporate the following majors changes:

1. Unlike raw and network-attached disks, IBP byte arrays have time limits. A simple way of incorporating this into the file system is to have another thread, much like the cleaner thread, that refreshes time limits, and if that is not possible, copies the data to another IBP byte array.
2. Many file systems implement striping and/or replication for performance improvements on more tightly coupled networks than the wide area. On the wide area, cache-based replication, and logistically scheduled replication (as is performed in a rudimentary fashion by IBP Mail) will be a necessity, and will have to be incorporated into LoFS.
3. As such, files in LoFS will consist of replicated blocks, and coding blocks (like parity blocks in RAID) that enable clients to rebuild file blocks when they are unavailable. The coding will be based on Reed-Solomon coding[26], which allows for a system to provide m coding blocks for *n* blocks of data, and then to tolerate the failure of {\em any} *m* blocks (note, RAID Level 5 parity is equivalent to Reed-Solomon coding with *m* equal to 1).

Once these changes are in place, we may experiment with the use of time-limited IBP storage on the wide-area as the basis for a network-wide file system.

8. Related Work

IBP occupies an architectural niche similar to that of network file systems such as AFS [1] and Network Attached Storage appliances [27], but its model of storage is more primitive, making it similar in some ways to Storage Area Networking (SAN) technologies developed for local networks. In the Grid community, projects such as GASS [28] and the SDSC Storage Resource Broker [29] are file system overlays that implement a uniform file access interface and also impose uniform directory, authentication and access control frameworks on their users.

9. Conclusions

Validation of the claimed scalability of the logistical model of file system design based on IBP requires a research program based on implementation and extensive experimentation and experience through deployment. For this reason, we are commited to an aggressive schedule of code release, and all released code can be freely downloaded from our project website `http://icl.cs.utk.edu/ibp`. Today, the depot software runs on a variety of Unix-based operating systems and the Win32 platform and the client library runs on these platforms as well as Java. Porting to other systems is being pursued as interested user communities or interesting experimental opportunities are identified. The test suite and IBP application-building tools are also available. A specification of the exNode serialization and a preliminary version of the exNode library are scheduled for release before publication of this paper. The L-Bone is a service available to any IBP depot administrator, and the Logistical File System is currently under development, and the design of content distribution systems based on IBP is being studied.

An underlying thesis of our research program in the confluence of networking and storage technologies is that community-based resource sharing is one of the important factors that distinguishes the Internet from other communication networks. In formulating the Internet Backplane Protocol as a common mechanism for the sharing of storage resources, we are making a conscious attempt to emulate key aspects of IP networking while generalizing them to a new domain. While the challenges are considerable, and success is far from guaranteed, the potential reward is a new phase in the Internet revolution involving not just end-to-end communication but also the management of distributed state ranging from communication buffers to distributed files with the myriad policies and algorithms used by distributed applications of all sorts applied to a common underlying infrastructure. The ultimate goal is not simply to enable distributed file systems to scale across administrative domains in the manner of the Web, but to achieve a similar level of deployability for distributed systems of all sorts, ultimately integrating distributed computational resources (process cycles) to create a Logistical Computing and Internetworking infrastructure provisioned with the fundamental troika of distributed resources: bandwidth, storage and computation.

10. References

[1] J. H. Morris, M. Satyanarayan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," Communications of the ACM, vol. 29, no. 3, pp. 184-201, 1986.

[2] M. Satyanarayanan and M. Spasojevic, "AFS and the Web: Competitors or Collaborators?," Operating Systems Review, vol. 31, no. 1, pp. 18-23, 1997.

[3] M. Beck, T. Moore, J. Plank, and M. Swany, "Logistical Networking: Sharing More Than the Wires," in Active Middleware Services, vol. 583, The Kluwer International Series in Engineering and Computer Science, S. Hariri, C. Lee, and C. Raghavendra, Eds. Boston: Kluwer Academic Publishers, 2000.

[4] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski, "The Internet Backplane Protocol: Storage in the Network," presented at NetStore99: The Network Storage Symposium, Seattle, WA, 1999.

[5] A. Bassi, M. Beck, J. Plank, and R. Wolski, "Internet Backplane Protocol: API 1.0," Department of Computer Science, University of Tennessee, Knoxville, TN, CS Technical Report, ut-cs-01-455, March 16, 2001 2001. http://www.cs.utk.edu/~library/2001.html.

[6] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," ACM Computing Surveys, vol. 26, pp. 145-185, 1994.

[7] H. Garcia-Molina and K. Salem, "Disk striping," in 2nd International Conference on Data Engineering: IEEE, 1986, pp. 336-342.

[8] R. W. Watson and R. A. Coyne, "The Parallel I/O Architecture of the High-Performance Storage System (HPSS)," presented at IEEE Mass Storage Systems Symposium, 1995.

[9] M. Beck, T. Moore, and J. S. Plank, "Exposed vs. Encapsulated Approaches to Grid Service Architecture," presented at 2nd International Workshop on Grid Computing, Denver, CO, Nov. 12, 2001, 2001.
[10] M. T. Rose, "The Blocks Extensible Exchange Protocol Core," Internet Engineering Task Force, IETF RFC, RFC3080, March 2001. http://www.ietf.org/rfc/rfc3080.txt.
[11] A. Bassi and X. Lee, "Internet Backplane Protocal - Test Language 1.0," Department of Computer Science, University of Tennessee, Knoxville, TN, CS Technical Report, ut-cs-01-454, June 2001. http://www.cs.utk.edu/~library/2001.html.
[12] M. Beck and E. Fuentes, "A UDP-Based Protocol for Fast File Transfer," Department of Computer Science, University of Tennessee, Knoxville, TN, CS Technical Report, ut-cs-01-456, June 2001. http://www.cs.utk.edu/~library/2001.html.
[13] W. Elwasif, J. Plank, M. Beck, and R. Wolski, "IBP-Mail: Controlled Delivery of Large Mail Files," presented at NetStore99: The Network Storage Symposium, Seattle, WA, 1999.
[14] H. Casanova and J. Dongarra, "Applying NetSolve's Network Enabled Server," IEEE Computational Science & Engineering, vol. 5, no. 3, pp. 57-66, 1998.
[15] D. C. Arnold, S. S. Vahdiyar, and J. Dongarra, "On the Convergence of Computational and Data Grids," Parallel Processing Letters, forthcoming.
[16] J. Gelas and L. Lefevre, "TAMANOIR : A High Performance Active Network Framework," in Active Middleware Services, vol. 583, The Kluwer International Series in Engineering and Computer Science, S. Hariri, C. Lee, and C. Raghavendra, Eds. Boston: Kluwer Academic Publishers, 2000.
[17] A. Burton and M. Beck, "Creating a File System with Caching Using Internet Backplane Protocol," Department of Computer Science, University of Tennessee, Knoxville, TN, CS Technical Report, ut-cs-01-465, July 19 2001. http://www.cs.utk.edu/~library/2001.html.
[18] J. K. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: A Case for Log-structured File Systems," Operating Systems Review, vol. 23, no. 1, pp. 11-27, 1989.
[19] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation fo a Log-Structured File System," Operating Systems Review, vol. 25, no. 5, pp. 1-15, 1991.
[20] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," in Conference Proceedings, Usenix Winter 1993 Technical Conference. San Diego, 1993, pp. 307-326.
[21] J. T. Kohl, C. Staelin, and M. Stonebraker, "HighLight: Using a Log-structured File System for Tertiary Storage Management," in Conference Proceedings, Usenix Winter 1993 Technical Conference. San Diego, 1993, pp. 435-447.
[22] M. Burrows, C. Jerian, B. Lampson, and T. Mann, "On-line Data Compression in a Log-structured File System," in Fifth International Conference on Architectural Support for Programming Languages and Operating Systems: ACM, 1992, pp. 2-9.
[23] J. H. Hartman and J. K. Ousterhout, "The Zebra Striped Network File System," Operating Systems Review -- 14th ACM Symposium on Operating System Principles, vol. 27, no. 5, pp. 29-43, 1993.
[24] I. Murdock and J. H. Hartman, "Swarm: A Log-Structured Storage System for Linux," in Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference. San Diego, 2000.
[25] J. H. Hartman, I. Murdock, and T. Spalink, "The Swarm Scalable Storage System," in 19th International Conference on Distributed Computing Systems (ICDCS), 1999, pp. 74-81.
[26] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," Software - Practice & Experience, vol. 27, pp. 995-1012, 1997.
[27] G. Gibson and R. V. Meter, "Network Attached Storage Architecture," Communications of the ACM, vol. 43, no. 11, pp. 37-45, 2000.
[28] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke, "GASS: A Data Movement and Access Service for Wide Area Computing Systems," presented at Sixth Workshop on I/O in Parallel and Distributed Systems, May 5, 1999, 1999.

[29] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," presented at CASCON'98, Toronto, Canada, 1998.