

# Mobile Management of Network Files

Alex Bassi, Micah Beck, Terry Moore

**Abstract-- Common opinion holds that a precise definition of the concept of middleware is elusive because it is highly dependent on one's design perspective regarding application environments and system architectures. The approach to the mobile management of network files discussed in this paper, which involves issues of process mobility and architecture/OS independent execution, represents one such a perspective. Our previous work in the area of logistical networking has focused on the Internet Backplane Protocol (IBP), a technology for shared network storage that can scale in terms of the size of the user community, the aggregate quantity of storage that can be allocated, and the breadth of distribution of service nodes across network borders. To achieve this end we have followed a layered, bottom-up design philosophy that draws on the engineering principles well known from the design of the Multics operating system, RISC microprocessors, and most especially the Internet protocol stack. In this paper we introduce the exNode, a data structure intended to provide the basis for reliable and efficient implementation of a file abstraction on top of the primitive storage service defined by IBP and discuss its application in network-based file management.**

**Index Terms-- File system, Mobility, Network, Storage**

## I. INTRODUCTION: A POINT OF VIEW ON NETWORK STORAGE MIDDLEWARE

Common opinion holds that a precise definition of the concept of middleware is elusive because it is highly dependent on one's design perspective regarding application environments and system architectures [1, 2]. The approach to the mobile management of network files discussed in this paper, involving issues of process mobility and platform independent execution, represents one such a perspective.

Our work in the area of *logistical networking* has focused on creating technology for shared network storage that can scale in terms of the size of the user community, the aggregate

This work is supported by the National Science Foundation Next Generation Software Program under grant # EIA-9975015 and the Department of Energy Next Generation Internet Program under grant # DE-FC02-99ER25396

Alex Bassi is a Research Associate, Innovative Computing Laboratory, Computer Science Department, University of Tennessee, Knoxville, TN 37996-3450. telephone: 865-974-9972, e-mail: abassi@cs.utk.edu).

Micah Beck is Research Associate Professor, Innovative Computing Laboratory, Computer Science Department, University of Tennessee, Knoxville, TN 37996-3450. telephone: 865-974-3548, e-mail: mbeck@cs.utk.edu).

Terry Moore is Associate Director Innovative Computing Laboratory, Computer Science Department, University of Tennessee, Knoxville, TN 37996-3450. telephone: 865-974-5886, e-mail: tmoore@cs.utk.edu).

quantity of storage that can be allocated, and the breadth of distribution of service nodes across network borders [3]. To achieve this end we have followed a layered, bottom-up design philosophy that draws on the engineering principles well known from the design of the Multics operating system, RISC microprocessors, and most especially the Internet protocol stack [4]. To parallel the design of the IP stack, it was clear that the lowest globally accessible network layer in the network storage stack should provide an abstraction of *access layer resources* (i.e. storage services at the local level) that does at least the following two things:

- *Enable scalable Internet-style resource sharing* — The abstraction must mask enough of the peculiarities of the access layer resource (e.g. fixed block size, differing failure modes, and local addressing schemes) to enable lightweight allocations of those resources to be made by any participant in the network for their limited use and regardless of who owns them.
- *Expose underlying storage resources in order to maximize freedom at higher levels* — The abstraction should create a mechanism that implements only the most indispensable and common functions necessary to make the storage usable *per se*, leaving it otherwise as primitive as it can be; all stronger functions should be built on top of this primitive layer. The goal of providing essential functionality while keeping the semantics of this layer as weak as possible is to expose the underlying resources to the broadest range of purposes at higher layers, thereby fostering ubiquitous deployment and freeing developers to innovate.

Our survey of standard network storage systems showed that each of them fails in some degree to satisfy one or both of these criteria. So to address the need for primitive management functionality at the bottom of the network storage stack, we have created and implemented the *Internet Backplane Protocol (IBP)*.

IBP is a mechanism that supports the management of shared network storage through an abstraction that leaves the underlying resource as exposed as possible [3, 5]. Each IBP *depot* (server) provides access to an underlying storage resource for any client that connects to it. In order to enable sharing, the depot hides details such as disk addresses, and supplies a very primitive capability-based mechanism to safeguard the integrity of data it stores. At the same time, IBP's weak semantics and low overhead model of storage are designed to expose depot resources in a way that allows all kinds of more complex structures, such as asynchronous networking primitives and file and database systems, to be

built on top of the IBP API. Just as many in the networking community view “middleware” as anything above the IP layer, we view software constructed on top the IBP layer as network storage middleware.

The importance of this middleware becomes evident when you realize that its first essential layer needs to provide a data structure to support the existence of *files*. Now a file is not just a group of disk blocks. At a minimum a file is a set of disk blocks plus a data structure that implements the file abstraction. But the IBP client contains no such data structure. The disk blocks of IBP depots that would hold the data content of a typical file are not accessible to user applications through a standard file system interface, but must be accessed through IBP’s primitive mechanism. Since the most universal and intuitive abstraction of storage is a *file*, however, it is obvious that we must create a file abstraction on top of IBP that uses its low level functionality to deliver the kind of strong properties users automatically expect in a typical file, such as unbounded size and duration. Moreover, in order to support advanced applications of various kinds (e.g. new kinds of highly flexible and scalable overlay and edge networks), this new file structure needs to provide both *location independence* and *mobility across system and network boundaries*, whether or not the actual data on the disk moves as well.

The data structure we call the *exNode* (*external node*) represents our solution to this problem. In this paper we begin by examining the relationship between file data structures and processes in order to motivate the approach we take to creating a file abstraction for the network storage stack. After describing in more detail IBP’s exposed approach to storage resources, we discuss how our implementation of the *exNode* data structure creates a file abstraction that can use IBP’s primitive mechanism. Finally, we describe how the *exNode* provides mobile file management as a kind of highly structured mobile process, giving examples of its application, sketching out future directions for development, and concluding with some remarks about its potential significance.

## II. THE DUALITY BETWEEN FILES AND PROCESSES

In thinking through the problem of how to implement files in the context of the network storage stack, instead of asking the question “What is a file?” we found it more illuminating to ask “What is the relationship between the abstraction of a file and the abstraction of a process?” Historically this duality between files and processes has arisen from time to time in the design of operating systems. In message passing operating systems such as Demos [6] we model access to any resource (e.g. a file) as a request sent to the process that manages that resource. The identity of the manager is often hidden through the use of a capability that is opaque to the user. This opacity makes it reasonable to think of each file as if a distinct process managed it, even if those processes are in fact implemented by a monolithic file manager (possibly multithreaded, but not one

thread per stored file). The state of each “file-process” is in fact encoded in file system data structures, notably the file table entry and data blocks in cache and on the disk. (Fig. 1)

Conversely, operating systems such as Unix, which take the file descriptor as their central abstraction, reverse the relationship, modeling access of all stored state (e.g. of a process) as operations on a file. The identity of the portion of the operating system that implements those operations is hidden through the use of a *file descriptor* that is opaque to the user. Because each process has its own stored state, it is reasonable to consider each process as implementing a file. Yet the contents of each “process-file” is in fact encoded in a special process management data structure, notably the task control block and the memory image in main memory and swap space.

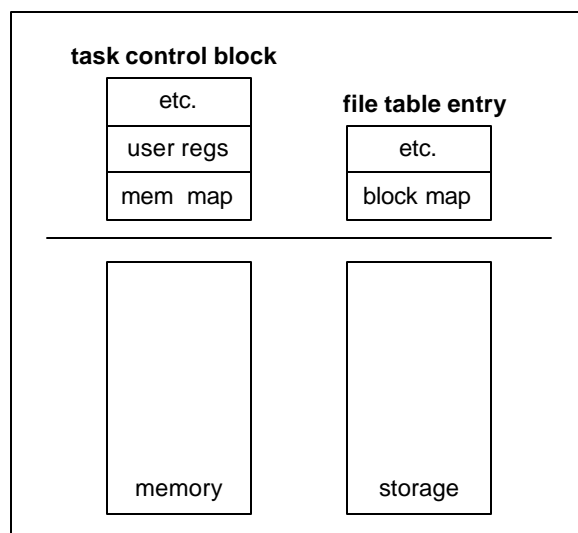


Fig. 1 File vs. Process

This seemingly reversible relationship between files and processes derives from the fact that both files and executing processes deal with *state that changes over time*. In the case of a file, there are two components of the state: 1) a *control component* that resides in data structures that determine the behavior of the file system and are maintained by the file system; and 2) a *stored data component* that resides in disk blocks, and is accessed and modified only through external commands to the file system. In the case of a process, there is no stored data component, but its control component is expanded to include a stored program and a large, unstructured memory space (see Figure 1). Thus the apparent duality occurs because we can always look at the state of each one from the point of view of the other:

- The active component of a file can be viewed as the state of a simple process
- If we treat the changing state of the large, unstructured memory space of a process as if it were a data component, it can be treated as a dynamically changing file

It is important to notice, however, that the correspondence

is not equally complete in both directions. While a file can easily be modeled as a process, when we model a process as a file *we must omit the portion of its state that does not map to memory*, notably the processor registers and the contents of the task control block. The process abstraction is more general than the file abstraction.

It is natural, then, to think of a file system as consisting as a collection of processes, each one of which is responsible for the implementation of a single file. The state of each such process is the control component of the state of the corresponding file. The bulk of a file's control state, corresponding to the inode, implements a mapping from the logical byte extent of the file to specific blocks stored on disk, i.e. to the data component of the files state. Because disk blocks are typically addressable only within a single processor or a storage area network [7], it is not clear how such a file-process could be usefully viewed as a mobile agent. But as we shall see below, the combination of IBP's exposed approach to access layer storage resources (sec. 3) and the exNode's method of encoding the control state of the file-process (sec. 4) loosens the bindings between files and particular local resources, allowing the control state to be mobile.

### III. EXPOSING NETWORK STORAGE WITH THE INTERNET BACKPLANE PROTOCOL

As noted above, in endeavoring to build a network storage stack, we drew on the analogy with the IP paradigm and applied a bottom up design approach, beginning with the lowest network layer. Now one of the well-known perils of this approach consists in succumbing to the temptation to implement high-level functionality at such a low layer, thereby losing the generality and predictability essential to the freedom that application designers need [8]. When application designers are unable to use low-level functionality because it is encoded in high level constructs that were created with different specific purposes in view, the temptation is to reimplement the lower layer, usually replacing the old high-level constructs with a set better adapted to the new purposes at hand. The result is balkanization.

Indeed, when you look at the current world of network storage, balkanization is what you see. While we will not argue for it here, we believe that the leading approaches to network storage — FTP and HTTP, network file systems like NFS and AFS [9], Network Attached Storage [10], Metastorage systems like Global Access to Storage Services (GASS) [11] an Storage Resource Broker (SRB) [12] — all push high-level functionality to an inappropriately low level. To recover the generality we believe is essential at the bottom of the network storage stack, we looked to establish a more primitive mechanism at that layer.

The Internet Backplane Protocol (IBP) is a network service that exposes an abstraction of storage resources to the network that is similar to the disk block [3, 5]. The primitive unit of IBP storage allocation is called a *byte array*. As an

abstraction of storage, the IBP byte array is at a higher level than the disk block (a fixed size byte array), and is implemented by aggregating disk blocks and using auxiliary data structures and algorithms. Abstracting away the size of the disk block, a byte array amortizes the overhead of allocation across multiple blocks. If we consider storage services at the disk block level to be the equivalent of “scalar” operations within a processor, then byte arrays allow a kind of “vectorization” of operations. Though our aim was to make the IBP storage service as exposed as possible, this level of encapsulation was considered indispensable to hide the most specific underlying characteristics of the access layer (physical medium and OS drivers) and to amortize per-operation overhead across multiple blocks.

Byte arrays can be used in various ways, and when they are so used we sometimes designate them in a way that emphasizes that fact. For example, when byte arrays are used in data communication, we call them *data buffers*. As a network service, IBP clients can allocate data buffers from IBP depots as freely as a file system allocates free disk blocks or an IP network allocates datagrams. These data buffers can be read and written over the network, and data can be freely transferred between buffers on different depots without passing through the client. In effect, IBP provisions the network with a distributed set of shared storage volumes, and allows clients to allocate from them freely.

In so far as IBP applies the IP paradigm to the sharing of storage resources across the network, it inherits both that paradigm's strengths and its weaknesses. There are two chief weaknesses. First, even more so than IP, IBP is susceptible to Denial of Use (DoU) attacks. Second, since IBP is intended to offer storage services even across the wide area network, it cannot offer anything close to the kind of strong storage service semantics that users are used to getting from processor attached storage.

Both of these problems are addressed through special characteristics of the way IBP allocates storage:

- *Allocations of storage in IBP can be time limited.* When the lease on an allocated data buffer 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 “allocation policy” 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.* IBP is designed to model storage accessed over the network, so 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 be necessary to assume that storage can be

permanently lost. Thus, *IBP is a “best effort” storage service*. IBP even supports the option of requesting “volatile” storage allocation semantics, allowing the allocation of unused storage that can be revoked at any time. In all cases such weak semantics mean that the level of service must be characterized statistically.

A more detailed account of the IBP API and a description of the status of the current software that implements the IBP client, servers, and protocol is available at <http://icl.cs.utk.edu/ibp>. The key point to note here, however, is that IBP’s limitations on the size and duration of allocation and its weak allocation semantics are precisely the reason that IBP cannot directly implement stronger, more reliable storage abstractions such as conventional files. On the other hand, the fact that the semantics of IBP remain so weak means that the underlying storage resources remain exposed, and this permits layers built on top of IBP to make extremely flexible use of them. As we shall see below, the *exNode* is designed to put this flexibility to good use.

#### IV. 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 [13]).
- *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) [9, 14].

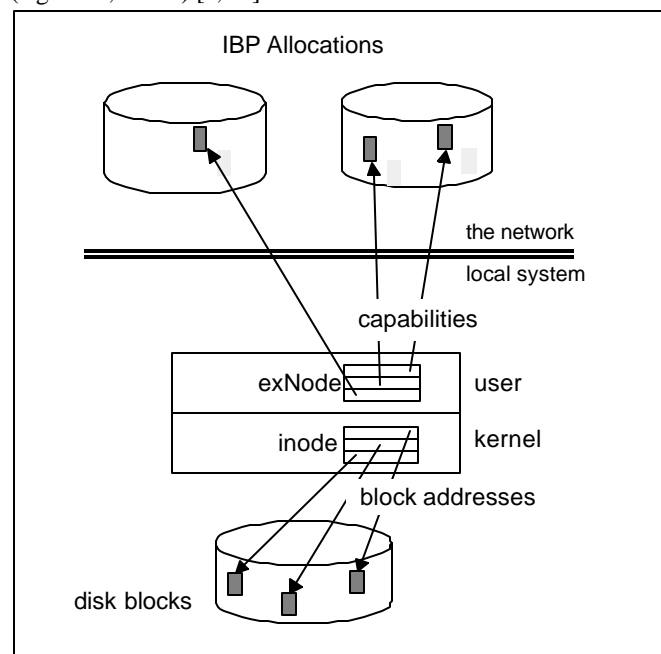


Fig. 2 *exNode* vs. *inode*

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 it. 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.

## V. THE EXNODE: MOBILE CONTROL STATE OF A NETWORK FILE

The exNode implements an abstract data structure that represents information known about the storage resources implementing a single file. The exNode is a set of declarations and assertions that together describe the state of the file. For purposes of illustration in this we introduce a small subset of the exNode specification and a minimal example of its application.

### A. A Simple exNode API

In this minimal formulation, the exNode is a set of *mappings*, each of which specifies that a particular portion of the file's byte extent during a certain period of time is mapped to a storage resource specifier that is given by a string (a URL or IBP capability).

A minimal exNode API must give us a means to create and destroy these sets of mappings, as well as a way of building them.

- Creation and destruction are implemented by simple constructor and destructor functions.

```
xnd_t n = xnd_create()
xnd_destroy(xnd_t n)
```

- An exNode is built by an operation that adds a mapping by specifying a data extent (start byte and length) a temporal extent (start time and duration) and a storage resource specifier.

```
xnd_addmap(xnd_t n,
           unsigned int data_start,
           unsigned int data_length,
           time_t time_start,
           time_t time_length,
           char *storage)
```

- The simplest possible useful query to an exNode simply finds one (of possibly many) storage resource descriptor and offset that holds the nth byte of the data extent at a specified time.

```
xnd_bytequery(xnd_t n,
              unsigned int byte_pos,
              time_t when,
              char **storage,
              unsigned int *offset);
```

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.

### B. XML Serialization of the exNode

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.

### C. Sample exNode Applications

- *IBP-Mail* [15] is a simple application that uses IBP to store mail attachments rather than include them in the SMTP payload using MIME encoding. *IBP-Mail* builds an exNode to represent the attached file and then sends the XML serialization of that file in the SMTP payload. The receiver can then rebuild an exNode data structure and use it to access the stored attachment.
- A *simple distributed file system* can be built by storing serialized exNodes in the host file system and using them like Unix soft links. Programs that would normally access a file instead find the exNode serialization, build an exNode data structure and use it to access the file. Caching can be implemented by creating a copy of accessed data on a nearby IBP depot and entering the additional mappings into the stored exNode.
- A *mobile agent* that uses IBP depots to store part of its state can carry that state between hosts in the form of a serialized exNode. If the hosts understand the exNode serialization, then they can perform file system tasks for

the agent while it is resident, returning the updated exNode to the agent when it migrates.

## VI. ACTIVE FILE MANAGEMENT USING THE EXNODE

. In conventional file systems, many users consider the mapping of files to storage resources as static or changing only in response to end-user operations, but in fact this is far from true:

- Even in a conventional disk-based file system, detection of impending failure of the physical medium can result in the movement of data and remapping of disk block addresses.
- Defragmentation and backups can be another examples of autonomous movement of data by the files system not driven by end-user operations.
- In a RAID storage system, partial or complete failure of a disk results in regeneration and remapping of data to restore fault tolerance.
- In network-based systems, scanning for viruses can be a necessary autonomous action resulting in deletion of files or movement to a quarantine area.

The exNode is most closely analogous to the state of a process when it is used to implement an autonomous activity that is not under the direct control of a client, and may be completely hidden. The following activities are examples of “file-process” activity.

### A. Active Probing to Maintain Fault Tolerance

The exNode can be used to express simple redundant storage of data or, with appropriate extension, to express storage aggregation functions such as redundant storage of error correcting codes. However, as with RAID systems, when failures are detected, data must be copied to reestablish redundancy. In the case of the network, which can experience partitions and other extreme events that cause depots to become unreachable, active probing may be required in order to ensure that data losses are detected in a timely manner. This can be accomplished by the host setting a timer and actively probing every depot for reachability. Because transient network partitions are common, this data must be correlated over time to deduce the likelihood of data loss or long-term unavailability. The frequency of probing may be adjusted to account for network conditions or the urgency of constant availability required by the application.

### B. Lease Renewal

In the case of IBP, storage allocations are time-limited, and so a persistent file must renew its leases regularly, and perhaps allocate and copy data to new storage resources when the old ones become unavailable. The exNode represents the time-dependent state of the file, and it is the basis for scheduling reallocation events.

The most straightforward strategy is for the system that hosts the exNode determines the allocation that expires first,

and set a timer a fixed amount before that expiration. When the timer is triggered, the host system attempts to renew the allocation lease, and if renewal fails it makes a new allocation and the copies the data. If all attempts at making a new allocation fail, the time is scheduled to timer again. Upon success, the timer is reset according to the next expiry time. This implements a simply lease renewal process driven by expiry events.

Note that in the simple case above, the time at which the renewal timer is set will be a function of a few parameters, including the expected time to renew or reallocate, and the maximum expected time that the host will be down. Larger values for these times will require more lead time before expirations.

Of course, the expected values of these parameters are not constant, but depend heavily on the state of available storage and the state of the host system. When available storage is scarce, more effort may be necessary to make the needed allocations and it may eventually consume a considerable amount of the host effort. When extended downtime is scheduled for the host (as when periodic maintenance is made or when a mobile system is disconnected from the network) the result is a need to make longer allocations and service them more regularly.

The easy answer to this variability in expected parameter values is assuming a worst-case value. The problem with this approach is that it can lead to extreme inefficiency when the expected behavior of the system is much better than the worst case. A preferable solution is to use estimates of the parameter values obtained, either empirically or through explicit user directives.

The state of the network is an aggregation of the behavior of many users, so it is impossible to obtain useful user directives. Thus, empirical estimates are the only option. If we look at the case of TCP/IP as an analogy in the world of synchronous point-to-point communication, it obtains estimates of the state of congestion on the network path between communicating pairs by detecting packet loss and interpreting it as congestion. This approach does not work in the case of storage, because there is no unique choice of depot. For this reason, exNode network monitoring is done through a global service provided by the the Logistical Backbone (L-Bone), an IBP directory project led by Jim Plank [5], using Rich Wolski's Network Weather Service [16] for statistical prediction.

The behavior of the host system can sometimes be predicted when the user is cooperative. An example of such cooperation is the controlled shutdown option provided by all operating systems that cache portions of the file control and data state in volatile main memory. The controlled shutdown provides warning for the file system to write all data out to non-volatile disk storage before the volatile storage becomes unavailable.

In the case of data stored in time-limited allocations, a shutdown longer in duration than the allocation lease will result in a loss of stored data. In cases where short-lived

allocations represent cached or temporary data, no action may be required. If, however, a system is taking advantage of short leases to avoid the effort or expense of obtaining long leases, it may be necessary to make longer allocations in order to survive a shutdown. In this case, it will be necessary to lengthen leases and reset timers well in advance of the shutdown, and so considerable discipline may be required on the part of the system administrator.

To account for these and other similar factors, the strategy for determining the duration of allocations and the setting of timers may be required to have dynamic components that make use of network monitoring and user directives. There is no magic in time-limited allocations; in order to obtain stability, they must be used in restricted ways or else discipline must be exercised in predicting and managing downtime.

### C. Defragmentation

In any system that responds to congestion by locally restricting the size of an allowable allocation, fragmentation can result. In this regard IBP is no exception: when depots become full, they limit the size of an allowable allocation, and clients will begin fragmenting the files they represent using the exNode (see Figure 3). For the case of network storage, fragmentation results in a loss of reliability, requiring increased forward error correction in the form of error coding or full duplication of stored data. This can put undesirable loads on the host system and actually increases the global demand for storage – at some point allocations simply fail.

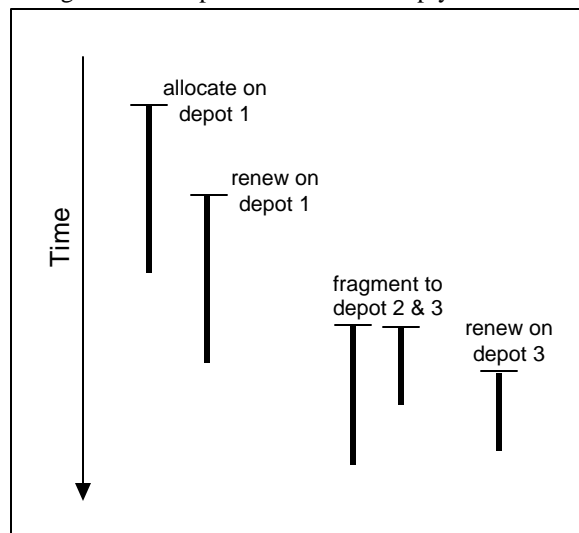


Fig. 3 Spatial and Temporal Fragmentation

Congestion can be caused by the underprovisioning of the network with storage resources, but it can also be caused by network partitioning that makes remote storage resources unavailable to the host system. Thus, storage congestion can be transient, but when it is relieved, the fragmentation that has been caused by it can persist. What is required is the merging of fragmented allocations. While it would be possible to attempt the wholesale defragmentation of an exNode, this may place a burden on the host system, and if attempted at a time

when congestion is not yet relieved may be fruitless. Instead, the host system may choose to attempt defragmentation through more aggressive lease renewal, combined with attempted merging of adjacent fragments. Over time, this will lead to a natural defragmentation up to the point where depots resist larger allocations.

#### D. Asynchronous Transfer Management

The current IBP client API implements a simple set of synchronous calls for all storage allocation, management and data transfer operations. However, large data transfers can take a very long time, sometimes longer than the exNode will reside at the client system at which it was initiated. In this case, the data transfer call must itself generate a capability to represent the state of the transfer in process. In order to find out the result of the data transfer operation and service the exNode accordingly, the receiver of the exNode must obtain the data transfer capability. Thus, data transfer capabilities and metadata describing the transfers they represent must be part of the exNode in transit. When the exNode is transferred between hosts before reaching its final destination (as when it is held by a mail spool server) that intermediate host can interpret and service the exNode (for instance representing a mail attachment).

A further complication arises when the intent of the host is not to initiate a single data transfer, but a set of dependent transfers, as when a mail message is routed to a set of receivers, making copies in a tree-structured manner. In this case, the sequence of operations and the dependences between them must be encoded in the exNode, and the processing of the exNode can involve issuing new data transfers as their dependences are resolved.

### VII. CONCLUSIONS

In designing network storage and computing systems we are attempting to work according to an architectural paradigm that requires us to *model* the underlying resources using a primitive abstraction and then *expose* that primitive abstraction to the network through a simple protocol, simple data representations and simple programming APIs. Accordingly, we have developed the Internet Backplane Protocol as our primitive storage service at the lowest globally accessible layer of the network storage stack.

Having modeled the underlying storage resource with an abstraction that is accurate in reflecting its weakness when accessed directly over the Internet, IBP does not provide the most commonly used and widely understood abstraction of storage, viz. of a reliable, persistent, unbounded files. In this paper we have motivated and taken a step towards the definition of the exNode, which is an abstraction designed to support the aggregation and management of IBP allocations in order to implement the file abstraction.

In implementing network files, there are many policy decisions to be made regarding the placement, duration and

degree of replication of data allocations. Following our layered architecture, the exNode is neutral to the policy used to make allocations – it is only used to manage them. Thus, the exNode abstraction can itself be used as a tool by a number of different file creation and maintenance services. The importance of the exNode is that it provides a framework within which a set of lower level services can be provided that is common across a broad class of still higher-level services.

If we think of our project as building a file system stack analogous to the network stack, the lowest layer, or access layer, is implemented by device drivers; the next layer is IBP, and the exNode provides a third layer. Higher layers will need to be created for allocation and management policy, and to provide a service at the user level analogous to that currently provided in distributed operating systems, but it will be built in a scalable manner that takes account of the weak properties of remote storage allocation in a scalable network.

### REFERENCES

- [1] R. Aiken, M. Carey, B. Carpenter, I. Foster, C. Lynch, J. Mambretti, R. Moore, J. Strasner, and B. Teitelbaum, "Network Policy and Services: A Report of a Workshop on Middleware," IETF, RFC 2768 February 2000. <http://www.ietf.org/rfc/rfc2768.txt>.
- [2] K. Geihs, "Middleware Challenges Ahead," *Computer*, vol. 34, no. 6, pp. 24-31, 2001.
- [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] D. P. Reed, J. H. Saltzer, and D. D. Clark, "Comment on Active Networking and End-to-End Arguments," *IEEE Network*, vol. 12, no. 3, pp. 69-71, 1998.
- [5] 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.
- [6] F. Baskett, J. H. Howard, and J. T. Montague, "Task Communications in DEMOS," presented at 6th ACM Symposium on Operating Systems Principles, West Lafayette, IN, November, 1977.
- [7] R. K. Khattar, M. S. Murphy, G. J. Tarella, and K. E. Nystrom, "Introduction to Storage Area Network, SAN," IBM International Technical Support Organization, Redbook SG245470, 1999.
- [8] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277-288, 1984.
- [9] J. H. Morris and *e. al.*, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 20, no. 3, pp. 184-201, 1986.
- [10] G. Gibson and R. V. Meter, "Network Attached Storage Architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37-45, 2000.
- [11] 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.
- [12] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," presented at CASCON'98, Toronto, Canada, 1998.
- [13] 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.

- [14] 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.
- [15] 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.
- [16] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computer Systems*, vol. 15, pp. 757-768, 1999.