The Architecture of BPFS: a Basic Parallel File System Version 1.0

Robert D. Russell

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The Architecture of BPFS: a Basic Parallel File System Version 1.0

Robert D. Russell

LIP, ENS Lyon

No 3460

Juillet 1998

THÈME 1

ISSN 0249-6399

RHÔNE-ALPES

The Architecture of BPFS: a Basic Parallel File System Version 1.0

Robert D. Russell *
LIP, ENS Lyon

Thème 1 — Réseaux et systèmes
Projet ReMaP

Rapport de recherche n° 3460 — juillet 1998 — 47 pages

Abstract: BPFS is a distributed, modular parallel file system designed to be used on networks of workstations. It is specified as a set of active components, functions utilized by the components, and protocols for communication between the components. The components can be implemented in many different ways, depending on the hardware and software support systems available, the performance desired, etc. A key idea is to be able to experiment with different implementations and configurations under different operating conditions to achieve a completely general, flexible system that is also capable of delivering good performance. In particular, it should be possible to implement this system on "commodity, off-the-shelf" (COTS) hardware and software. However, it should also be possible to implement specialized versions of some or all components to take advantage of unique hardware or software features.

BPFS is intended to support a wide range of possible applications, including real-time video on demand, medical and satellite image processing, out-of-core array manipulations, and general parallel computations that need a high performance file system. The parallel file system is intended to be "always available" for simultaneous use by any number of different applications. It is also capable of efficiently handling huge files containing many terabytes of data.

This report describes the architecture of BPFS in terms of the set of components, their organization, the functions they utilize, and the protocol specifications for communication between them.

Key-words: Parallel File Systems. Distributed File Systems. File Systems. System Architecture. Protocols.

(Résumé : tsvp)

* On leave during the 1997-98 academic year as an Associated Professor at the Laboratoire de l'Informatique du Parallélisme, École normale supérieure de Lyon. Permanent address: Department of Computer Science, Kingsbury Hall, University of New Hampshire, Durham, NH 03824-3591, USA. Email: rdr@unh.edu

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

L'architecture du système BPFS: un système basique de fichiers parallèle Version 1.0

Résumé : BPFS est un système de fichiers parallèle, modulaire et distribué, destiné à être utilisé sur les réseaux de postes de travail. Il est spécifié comme un ensemble de composants actifs, de fonctions utilisées par ces composants et de protocoles pour la communication inter-composants. Les composants peuvent être implémentés de façons diverses selon le matériel et le logiciel disponible, la performance souhaitée, etc. Une idée clé en est la possibilité d'expérimenter avec des implémentations et des configurations diverses dans des conditions d'utilisation differentes, afin de réaliser un système complètement général et flexible, qui est aussi capable d'atteindre bonnes performances. En particulier, il devrait être possible d'implémenter ce système avec du matériel et du logiciel "commodity, off-the-shelf" (COTS). Cependant, des versions spécialisées de certains (voire de tous) composants peuvent être implémentées afin d'exploiter les caractéristiques particulières de certains matériels ou logiciels.

Le but de BPFS est de soutenir toute une gamme d'applications, y compris la vidéo en temps réel sur demande, le traitement d'images médicales ou d'images prises par satellite, les manipulations de matrices en dehors de la mémoire et les calculs parallèles généraux qui nécessitent un système de fichiers de haute performance. Le système de fichiers parallèle est conçu de façon à être "toujours disponible" pour l'usage simultané par un grand nombre d'applications diverses. Il est aussi capable de traiter efficacement des fichiers énormes constitués d'un grand nombre de terabytes de données.

Ce rapport décrit l'architecture de BPFS en termes de l'ensemble des composants, leur organisation, les fonctions qu'ils utilisent et les spécifications de protocoles de communication.

Mots-clé : Système de fichiers parallèle. Système de fichiers distribué. Système de fichiers. Architecture de système. Protocoles.

1 Introduction

This report describes the architecture of BPFS — a Basic Parallel File System. It is designed to be one component of a more general system for multi-media parallel computation on groups of workstations interconnected by high-speed communication networks (i.e., Networks of Workstations (NOWs) and Clusters of Workstations (COWs)). The goal of BPFS is to be a general-purpose distributed block storage server. Therefore, it does not provide or utilize any mechanisms, such as synchronization, locks, barriers, etc., that could reasonably be expected to be provided by other components of a parallel computation system and that may be necessary at the application level to properly coordinate the parallel access to parallel files.

Because of its specification in terms of open protocols between network nodes, the design approach taken with BPFS is unlike that taken for other parallel file systems reported in the literature. This approach is very low level and is clearly intended for NOWs and COWs — it makes no sense whatsoever for "traditional" parallel machines such as the Intel Paragon, for example. The intention is to encourage experimentation with and evolution of these protocols, and to stress the "glue" between components rather than the components themselves, so that as newer technology, such as network attached disks, becomes available it can be utilized within the existing framework rather than requiring a redesign.

The use of the word "basic" in the title is intended to imply "fundamental", "no frills", and "low level". In particular, it is expected that programmers will not use this file system directly but rather through higher-level interfaces that are layered on top of it. The specifications in this document are in terms of components and protocols, not programming interfaces. The bulk of this report will be devoted to specifying the protocols and the actions they require from the components.

BPFS provides a tool by which files can be stored and accessed "in parallel" on workstation clusters. This use of the word "parallel" has several parts:

- The data in a single file is stored on several disks on different nodes using a technique known as "striping" or "declustering". All disks must therefore be accessed "in parallel" in order to obtain the complete file.
- Many processes will have simultaneous access the file system as a whole, as well as simultaneous access to individual files. Thus the active components of BPFS must operate "in parallel" on behalf of many user applications. This requires attention to the semantics of file access to guarantee consistency and integrity between file components while allowing files to be shared.
- A single application program will usually consist of several processes on separate network nodes all operating "in parallel". Thus BPFS must operate "in parallel" on behalf of a single user application.

The goal of all this parallel activity is to provide user applications with the ability to overlap computation with I/O, and to obtain higher I/O bandwidth and lower I/O latency than would be possible with a non-parallel file system.

The plan for the rest of this report is as follows: Section 2 gives an overview of the BPFS architecture, introducing the components and showing how they are related. Section 3 gives a brief review of the literature on parallel file systems, stressing those systems that most influenced the design of BPFS. Section 4 defines the general tasks associated with each of the four active components. Section 5 describes the functions which an implementation is required to supply in order to provide some of the flexibility and portability we would like to have for the active components. The longest section, Section 6, defines the protocols and explains the detailed actions of the active components as they relate to these protocols. This report concludes with a short section describing plans for future work. A brief overview of the first implementation is given in an Appendix.

2 General Overview

BPFS assumes a model of a parallel computing system that consists of a set of heterogeneous "logical nodes" capable of communicating with each other via a "communication network". There are three types of logical nodes: client nodes, manager nodes and server nodes.

A node may contain two types of "components":

- "active components" — the processes and/or threads that perform the operations of the parallel file system. There are four types of active components: clients, managers, servers and agents. These active components communicate with each other by using specific "protocols".
- "passive components" — the disk storage areas that contain the information in the parallel files. There are two types of passive components: data and metadata.

Figure 1 is a diagram showing the organization of logical nodes, the components on each, and their network connections.

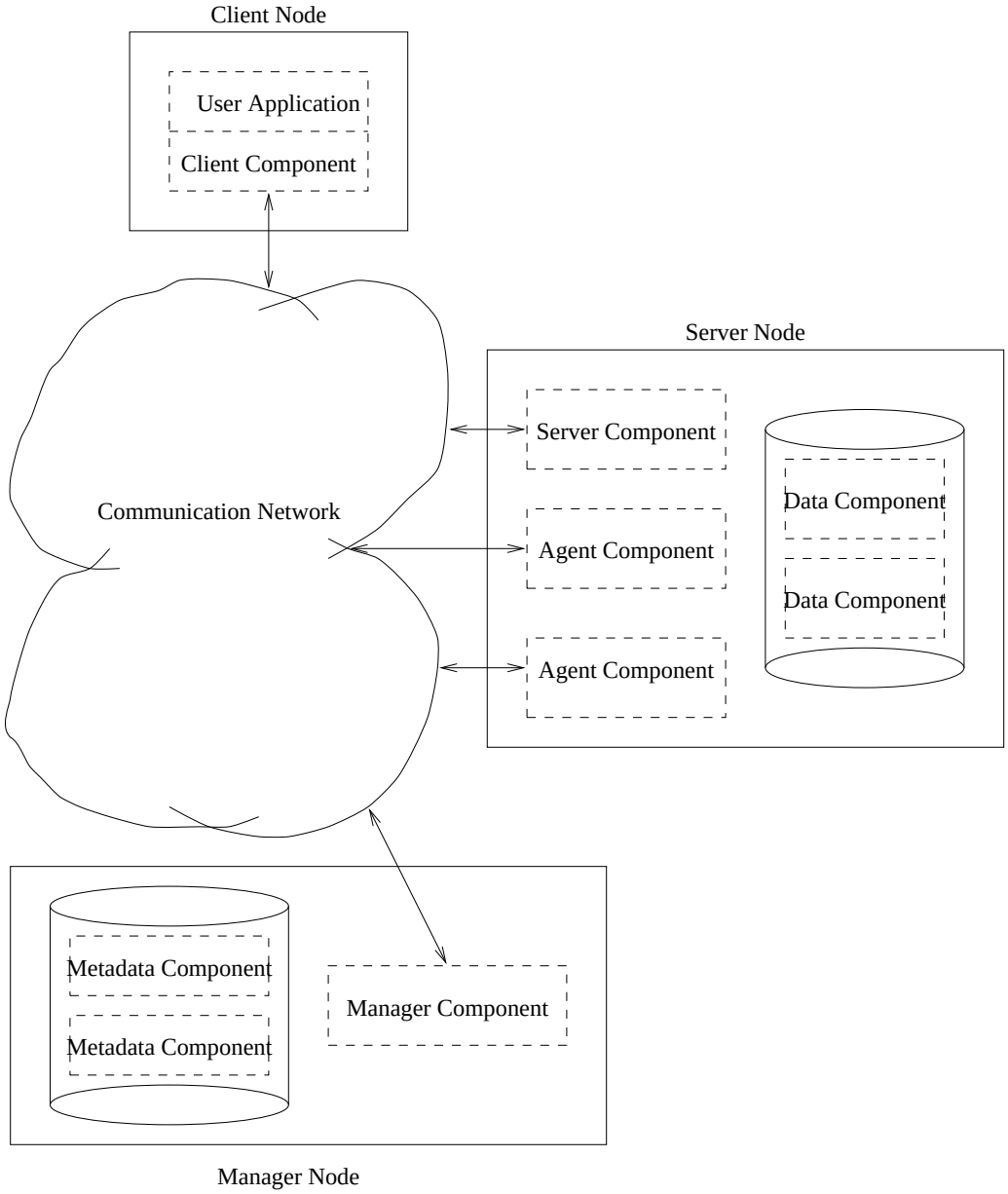

Figure 1: Overview of the logical nodes, components and communication network in BPFS. Solid rectangles indicate logical nodes, dashed rectangles indicate components, the irregular "cloud" indicates the communication network and arrows indicate network connections.

A single instance of the parallel file system is assumed to be "constantly available" to all applications simultaneously (i.e., it is not restricted to being run as part of a single application). It is assumed that logical nodes will dynamically join and depart from a running system.

It is further assumed that data is organized, stored and communicated in fixed-size blocks. The exact size of a block is determined by the implementation, but is expected to be at least the size of a physical disk block (usually 4096 bytes) and more likely larger. The block size must be the same for all applications using the same BPFS system. In this report, the symbol **BLOCKSIZE** is used to represent this implementation-defined value.

2.1 Logical Nodes

There are three types of logical nodes, although in any real system several different types of logical nodes may (and usually will) coexist on a single physical node. The types of logical nodes are:

1. Client nodes. A logical client node consists of a user application process (i.e., the end user of the parallel file system) linked with a client component.

2. Manager nodes. A logical manager node consists of a manager component, some amount of permanent storage for metadata components that are controlled by the manager, and some subset of the global file name space controlled by the manager.

3. Server nodes. A logical server node consists of a server component, a number of agent components, and some large amount of storage for data components that are controlled by the server.

2.2 Network

The network can be:

- A general purpose facility such as TCP over a Local Area Network (LAN), the global Internet, or a dedicated System Area Network (SAN) such as Myrinet.

- A special purpose facility such as BIP [16] over Myrinet, or SBP [17] over a dedicated Ethernet or ATM network.

- An integrated communication system such as Nexus [7].

Each active component in BPFS must have a unique "location" in this network. There are two aspects to a "location" — its internal representation and its external representation — and the contents of each depends on the implementation of the network layer. The internal representation is an opaque value occupying 8 bytes, which must be in network byte order, since locations are themselves sent over the network between potentially heterogeneous nodes. The external representation is an ASCII equivalent that can be represented in a C string and stored as part of the human-readable metadata of a file. Note that there is no automatic interoperability between different network representations, so if the network layer is changed, parallel files created with the previous network may no longer be accessible.

In TCP/IP, for example, a "location" consists of an IP address and port number. In the internal representation, the opaque value will contain the IP address as a 32-bit binary number and the port number as a 16-bit binary number, both in network byte order. In the external representation, the location will be the string representation of an IP address in "dotted-decimal" notation or an equivalent DNS name, followed by a colon followed by the port number as an integer. The port number can be omitted from the external representation, in which case a "well known port number" is used by default when the external representation is converted into internal form.

The implementation must define two functions to convert between the internal and external representations of a location (see Section 5.2).

2.3 Files

In the BPFS model, a "file" is a contiguous sequence of logical blocks numbered $0, 1, 2, \ldots, n$, that are stored in "permanent" storage under a single name. Each block contains the same fixed number of 8-bit bytes except possibly the last block in a file, which can be smaller. A "parallel file" consists of one "metadata" component stored on a manager node, and one or more "data" components each stored on a separate server node. Each data component consists of a contiguous sequence of relative blocks numbered $0, 1, 2, \ldots, k$. Different data components of the same file may contain different numbers of blocks.

A client maps a file onto a parallel file by a "mapping function", often called a "declustering function" or "striping function" in the literature. For each logical block in the file, this function determines which data component contains the block and the corresponding number of the relative block within that data component.

Metadata consists of the following information:

- The mapping function that defines how logical blocks in the file are mapped onto relative blocks in the data components.
- A parameter called the "stripe thickness" used by the mapping function.
- The list of server nodes on which the data components are stored.

The single metadata component for a parallel file is stored on a manager node. The data components are distributed on one or more server nodes. Depending on the implementation, the metadata component and each data component may be stored as normal files in the underlying host file system using a suitable naming convention.

The metadata component is stored in human-readable text format. The mapping function and the thickness parameter are represented by integer numbers. The list of server nodes is represented as a blank-separated list of server locations in their external representation. Each implementation must define the exact mechanism by which it loads with the client the executable code for user-defined mapping functions to correspond with the function numbers in metadata components.

2.4 Active Components

BPFS consists of the following active components:

1. Clients residing on client nodes, any number of clients per logical node.
2. Managers residing on manager nodes, one manager per logical node.
3. Servers residing on server nodes, one server per logical node.
4. Agents residing on server nodes, any number of agents per logical node. Each agent controls a single open file. All agents on the same server node are controlled by the server component on that node.

There are usually multiple instances of each of these components in a running system. Although there is only one manager per manager node, and one server per server node, multiple managers (servers) can run simultaneously on the same physical node provided they operate in non-overlapping domains (for example, in separate partitions on the node's disks).

The active components communicate between themselves by the use of special protocols. Figure 2 is a diagram showing these components, the protocols between them, and the disk I/O performed by each component.

2.5 Functions

BPFS contains the following functions by which the actions of the components can be tailored to particular application and system requirements:

1. Functions to supply a manager with a set of default metadata to use when creating new files.

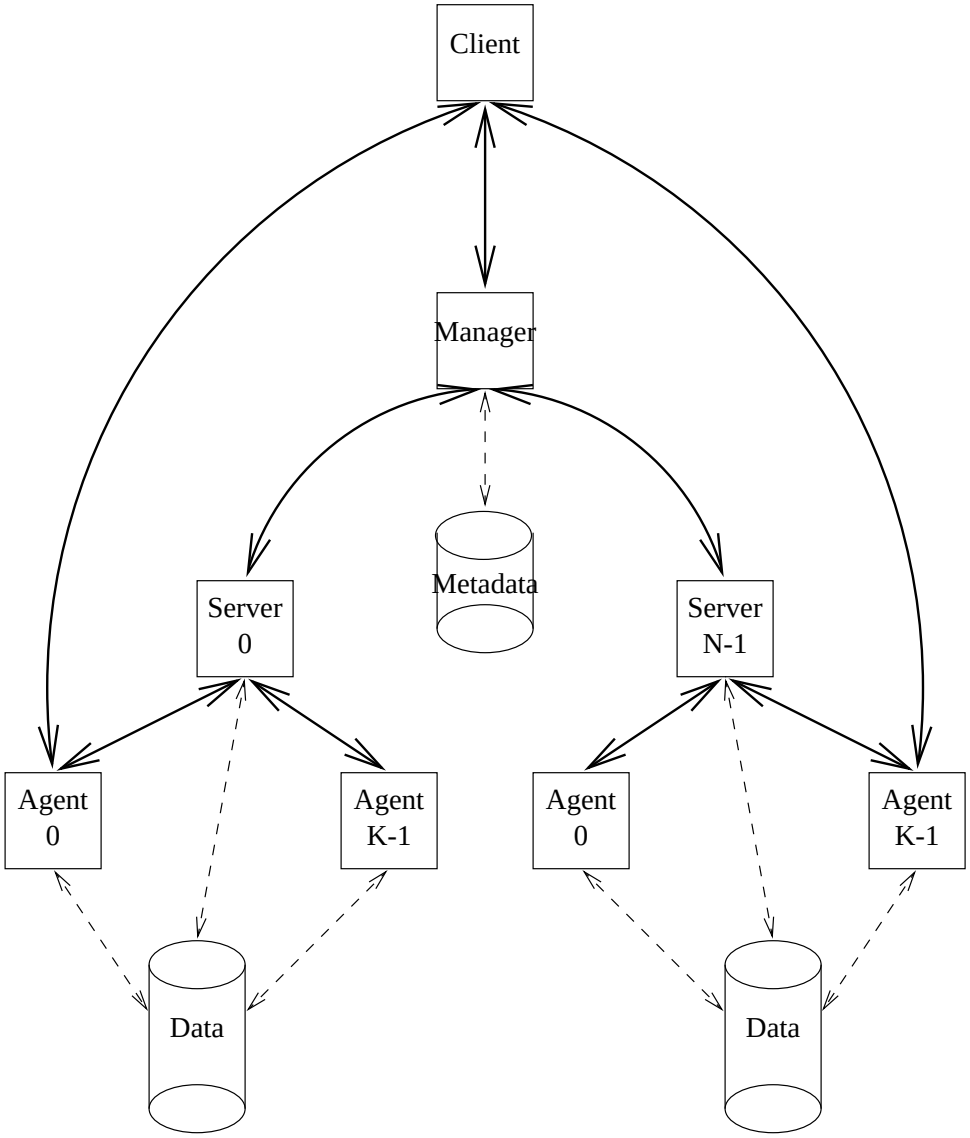

Figure 2: Overview of active components, protocols and file I/O in BPFS. Squares indicate components, solid arrows indicate protocols and dashed arrows indicate disk I/O.

2. Functions to convert between the internal and external representations of a network "location".
3. A function to map a file name onto a manager location.
4. A function to gather the access control information necessary for clients to access files.
5. For each mapping function, a set of striping functions to map logical block numbers onto data components and relative block numbers within a data component.
6. For each agent-side cache replacement policy, a set of functions to add, remove, and reorder blocks on the cache list.

An implementation will define default values for each of these functions, and will provide a mechanism for users and system administrators to dynamically specify alternative functions, subject to certain constraints.

2.6 Protocols

BPFS specifies protocols for each of the following component to component communications:

1. Client – manager
2. Client – agent
3. Manager – server
4. Server – agent

These protocols are designed to be efficient on both 32- and 64-bit architectures. Numeric fields are never bigger than 32 bits, and major data structures within messages are aligned on 8-byte boundaries and are multiples of 8 bytes in size. However, the limitation to 32-bit values does not mean that file sizes are limited to 2^{32} bytes. This is due to the fact that BPFS deals only with block numbers, not byte numbers. Therefore, a file is limited to $BLOCKSIZE \times 2^{32}$ bytes. If BLOCKSIZE is 4096 (2^{12}) bytes, the file size would be limited to 2^{44} bytes, which is more than 17 terabytes. Obviously larger block sizes would allow correspondingly larger file sizes. Of course, this assumes that the underlying file system is capable of storing and addressing such large files.

The bulk of this report will be devoted to specifying these protocols and the actions they require from the active components.

3 Background and Related Work

This work was inspired by the large number of parallel and distributed file systems that have been developed in the past few years. A number of these along with their key ideas are discussed here.

RAMA [12] (Rapid Access to Massive Archive) was designed by Miller and Katz at the University of California at Berkeley. Their key innovation was the use of a system-wide hashing function that pseudo-randomly distributed blocks in a parallel file across many disks. This function could be computed directly by a client application on any node in the system, thereby allowing each application direct access to any block without consulting a central node. This approach had two important consequences: it provided good performance across a wide variety of workloads without any effort on the part of the programmer to place data advantageously; and it scaled to a large number of nodes and disks. They provide simultations showing that their strategy of random block placement can perform within 10 to 15% of the optimum explicit placement, and is a factor of four or more better when the explicit placement is poor. In BPFS we have generalized their notion of a system-wide function for direct client-to-server block mapping to allow applications to provide arbitrary block mapping functions with each file.

PPFS [5] [9] (Portable Parallel File System) was developed by Reed and Chien at the University of Illinois at Urbana-Champaign. This is a portable user-level input/output library designed to permit rapid and flexible experimentation with policies for buffering, caching, data distribution and prefetching parallel files. They provided predefined policies for all necessary functions, but also allowed users to specify file layouts, access patterns, and prefetching patterns. In BPFS we have generalized this notion to allow many of the file system functions to be provided by the application programmer and/or the system implementor.

PPFS assumes three things for portability: an underlying UNIX-like file system, a parallel machine environment which supports message passing, and a C++ programming language environment running on the UNIX operating system. The corresponding requirements for the design of BPFS are more fundamental: any host file system, including raw disk access, any networking technology, and a standard C programming language environment running on the UNIX operating system. A key difference in design philosophy is that BPFS is intended to be a utility that is always up and running, and that simultaneously serves all applications for all users on the physical hardware. With PPFS, on the other hand, each user's parallel virtual machine gets its own copy of PPFS that is independent of other copies in other virtual machines on the same physical hardware.

The work by Gibson on SPSS [8] (Scotch Parallel Storage Systems) at Carnegie Mellon University incorporates into a running system earlier work by Patterson and Gibson [15] stressing the importance of

prefetching and caching on parallel file system performance. BPFS provides two mechanisms for prefetching: the ability to request groups of blocks at once, and the ability to request streams of blocks in which the agent "pumps" data to the client at a specified rate. Control of these mechanisms lies in the layers above BPFS itself, as does any client-side caching mechanism. Agent-side caching on a per-file basis is provided directly in BPFS, although it can be controlled by higher levels. BPFS also follows the SPSS philosophy of providing just a basic parallel storage component within a broader parallel programming system. In particular, neither BPFS nor SPSS provide any of the common synchronization or locking mechanisms that applications may need to properly coordinate parallel access to parallel files. These mechanisms would have to be provided by a "message passing" subsystem, such as MPI [6], that would be used from a layer on top of BPFS.

DPSS [10] (Distributed Parallel Storage System), developed by Tierney at the Lawrence Berkeley National Laboratory, is a collection of disk servers operating in parallel over a wide area network to provide logical block level access to large data sets. The implementation is based on user-level software running on UNIX workstations interconnected by an ATM Internet. It was built from commercial, off-the-shelf (COTS) components, which is also a major goal for BPFS.

xFS [1] was developed by Anderson at Berkeley. It was based on the use of a striped log-structured file system, in which new blocks are always written at the end of the available file storage space and old blocks are reclaimed later via a background collection procedure. The complexity of this system made it difficult to implement and debug, and it required some changes to the operating system kernel which becomes a serious impediment to portability. The design of BPFS attempts to avoid these complexities. It also avoids the cooperative caching designed into xFS, taking instead the same view as SPSS that such synchronization and process cooperation properly belong in higher-level layers built on top of the basic file system.

The work of Kotz at Dartmouth on the Galley file system [11] [14] also stresses the philosophy that a single strategy for various aspects of parallel file systems, such as caching, cannot suit all applications. He stresses the idea of building a simple, general purpose core storage system, and then layering a set of high-level libraries on top of it to provide many different application programmer interfaces (APIs). This is exactly the philosophy adopted by BPFS. The ViC* preprocessor [4] developed by Cormen at Dartmouth, the PASSION Runtime Library [3] (Parallel and Scalable Software for Input-Output) developed by Choudhary at Syracuse, and the Panda Library [2] developed by Seamons and Winslett at the University of Illinois at Urbana-Champaign, illustrate different high-level tools that are layered on top of a "commodity" parallel file system to provide a convenient API for manipulating large, "out of core" arrays.

Even the MPI-IO interface [13], which is considered "low level" compared to Panda and ViC*, is still considered "high level" by BPFS, and a companion report will describe our implementation of MPI-IO on top of BPFS using the ADIO (Abstract Device Interface for Implementing Portable Parallel-I/O Interfaces) developed by Thakur et al. [18]

4 Active Components

4.1 The Client

The **client** is that portion of a user process that deals with all access to parallel files by users. It may be accessed directly by a user application program, but more likely will be accessed indirectly through a standardized interface such as MPI-IO. If a user application consists of several processes, any communication between those processes in order to coordinate the use of a parallel file must occur either in the user program itself or in the interface between the user program and BPFS. BPFS itself assumes no communication or coordination between clients (i.e., it assumes that all clients are independent of each other).

The client software is usually implemented as a library package that handles all the communication with the other commponents of BPFS over the network. The first implementation uses TCP/IP, but versions are planned that would use other network communication facilities, such as BIP and SBP.

The functionality provided by a client consists of two types of functions:

1. Functions necessary to manage a file, such as deleting it, renaming it, and obtaining information about it (i.e., its size, its access permissions, its locations, etc.).

2. Functions necessary to read and write data from/to the file.

As shown in Figure 2, clients never deal directly with storage of the metadata or data — they always go through either a manager or an agent. All management functions require communication between the client and the file's manager. All read and write operations require the client to first communicate with the file's manager to "open" the file and set up a connection between the client and each of the file's agents. Once these connections are established, all data flows directly between the client and the agents with no further intervention by either the manager or the server until the client is finished with the file and wishes to "close" it, at which time the client must again contact the manager.

To perform a read or write operation, the user software provides the number of the affected block to the client software. The client software is responsible for applying the file's mapping function to this logical block number to obtain the connection to the appropriate agent and the relative block number on that agent's data component. The client will then send a request for that relative block directly to that agent, and will receive a response directly from that agent. Note that data is always transfered in units of fixed-size blocks — any mapping of arbitrary byte sequences onto blocks must be done by the interface between the client and the user software. For example, the MPI-IO interface has been implemented on top of a client to provide the file "view" facility as defined in the MPI-IO standard.

Each file indicates its own mapping function which is supplied and utilized by the client software in user space. This allows an application great flexibility to provide functions suited to their particular needs. For example, mapping functions can be provided that utilize different data layouts for different parts of a file, or that interact dynamically with other parts of the application program to determine the best data layout, etc.

4.2 The Manager

Each **manager** is an independent daemon process running on a manager node whose task is to create, delete and maintain metadata stored on that node about a set of parallel files. Each file has exactly one manager and all clients wishing to access that file must first go through that manager. There can be multiple simultaneous managers, each managing a subset of the total universe of parallel files. The mechanism by which a file is mapped onto a manager must be known to all clients, but is itself a lower-level function in BPFS that can be different from implementation to implementation (see Section 5.3).

The manager is responsible for coordinating the actions of the set of servers on which the data components of a parallel file reside. The manager maintains the integrity and consistency of the file during system operation. For example, most file operations must be performed on all of the data components simultaneously, and many of them must be done in an atomic fashion with respect to other operations on the same file. If any server is unable to perform an operation for any reason, then none of the other servers must be allowed to perform it either. The manager is responsible for coordinating and ensuring this consistency.

By "off-loading" these functions onto a separate management process, BPFS clients can potentially increase their gain from parallelism, since they can overlap their computation with the tasks performed by the manager.

The client interacts with a manager when performing "management" functions on the file, and when a file is opened or closed. Management functions effect only the metadata of the file, and make no access to the data contained in the file (unless it is to delete the file entirely). The types of management functions available are:

1. Obtain status information about a file (its size, its location(s), its mapping function, its access permissions, etc.).
2. Delete the file.
3. Rename the file.
4. Link the file to an additional name.

In order to access the data in a file, the client must first send an "open" request to the manager of the file. The manager obtains the metadata for an existing file, or creates it for a new file, and verifies that the access requested by this client is legitimate. This requires that the manager contact each of the servers on

which the file data itself resides. The manager must coordinate the setup of these servers and obtain their agreement on the validity of the client's request (for example, if the client wants to write to the file does he have permission on each server?). Once this is done, the manager returns the necessary information to the client. The manager does not participate in any way during the data transfer to/from the file, which occurs directly between the client and the agent. In fact, the manager retains no knowledge that the file is open — that is the job of the servers. When the client wishes to close the file it must recontact the manager in order to coordinate the closing among all the servers.

4.3 The Server

Each **server** is an independent daemon process running on a server node whose task is to create and maintain the data components stored on the disks of that node for a set of parallel files. In the first implementation, the server process utilizes the underlying local UNIX file system on the server node to store data. However, the basic design makes it possible to utilize other existing file systems, or to dedicate a disk to a server that utilizes "raw" access to manage disk blocks directly.

Upon receiving a request from a manager, a server will open a data component and verify that the request is legitimate. If necessary, it will then set up a new agent, or contact an existing agent, to control all data access to/from the file. It is the server's task to enforce consistent access semantics to a file by all clients. These semantics are:

1. Shared read-only: One or more clients may access the file simultaneously, but only to read data.
2. Exclusive write-only: Exactly one client may access the file at a time to write data.
3. Shared read-write: One or more clients may access the file simultaneously, and any client may either read or write data.

4.4 The Agent

The **agent** is a process or thread spawned by the server, one agent for each unique active data component on the server. The task of the agent is to read and write all data from/to the data component on behalf of all the clients. If several clients are simultaneously accessing the same file, they will all be dealing with the same agent on each server node, although over different connections.

When a manager determines from its communication with the servers that all data components of a client's open request can in fact participate consistently in the request, it will return to the client a table of locations for each component's agent. The client will then use these locations to establish a direct network connection to each agent, and all file data will flow over the network directly and independently between the client and each agent.

The agent is responsible for enforcing the consistency semantics of this data, and for maintaining a cache of data blocks as may be appropriate for the access patterns of the clients. It is also responsible for "pushing" data to clients at regular time intervals during data streaming operations.

In the initial implementation, each agent is a separate child process spawned by the server. However, future implementations could use threads to implement agents.

5 Functions

The BPFS functions are user-supplied or implementation-supplied functions that allow BPFS to adapt to different environments and applications. These functions can be grouped into two general categories: those defined once per implementation by the BPFS implementor, and those defined once per file by the BPFS user. Each implementation will define a mechanism by which these functions can be supplied by users and system administrators. If possible, this mechanism should allow dynamic modification of a running parallel file system. The following table categorizes these functions and also indicates which software component invokes them.

Function	Defined by	Called by
default metadata	implementation	manager
"location" conversion	implementation	manager
manager mapping	implementation	client
access control info gathering	implementation	client
file striping	user	client
agent caching	implementation	agent

5.1 Default Metadata Functions

These functions are invoked by a manager in order to obtain default metadata for use in the creation of new files. There are two functions, one to define the default list of servers known to the manager, and one to define the default striping function to map files onto those servers. Both of these functions are fixed for a given implementation.

```
extern int getdefaultserverlist( unsigned int listlen,
        char *list );

extern void getdefaultstriping( unsigned int *default_mapping,
        unsigned int *default_thickness );
```

5.1.1 getdefaultserverlist

The default server list function is called once by the manager when it starts up in order to obtain a default list of servers onto which it can store data components. For example, this information might be obtained from the manager's environment variables or configuration files in a system-dependent manner.

Input parameters:

- **listlen** — an unsigned int containing the number of bytes in the output parameter **list** allocated by the caller for storing the result. This value should be positive.

Output parameters:

- **list** — a null-terminated C string containing the list of comma- or blank-separated server locations. Each item in the string should have the implementation-defined external representation of a network location. For a TCP/IP network, this form will be the DNS name or the dotted-decimal IP address, followed by an optional colon and port number. The items in this list should be convertable to their equivalent internal representations by the implementation-defined function "q_strtolocation" (see Section 5.2). The storage for this string must be allocated by the caller, and must contain **listlen** bytes.

Function result:

- the number of characters stored in string **list** by this function, excluding the null terminator. If this value is positive, it is the same as strlen(**list**) and will never be larger than **listlen** − 1. If this value is zero, no default server list was obtained, and **list** will be the empty string. If this value is negative, it means that **listlen** was not big enough to hold the entire list, and its absolute value is the number of additional bytes needed to store the entire list as a null-terminated string. In this case, the first **listlen** − 1 bytes of the list will be stored in **list**, followed by the null-terminator.

5.1.2 getdefaultstriping

The default striping function is called by the manager in order to obtain the default function and its parameter to use to stripe data across the servers.

Input parameters:
None.

Output parameters:

- **default_mapping** — an unsigned integer value representing the mapping function to be used as the default striping function. A value of 1 means "regular striping". A value of 2 means "pseudo-random block placement". Other values indicate application-defined functions.
- **default_thickness** — an unsigned integer value representing the default "thickness" used by the default striping function. This is the number of consecutive logical blocks that are mapped into consecutive relative blocks on one server in each stripe of the file.

Function result:
None.

5.2 Location Conversion Functions

These functions convert between the internal and external representations of an implementation-defined network "location".

```
extern int q_strtolocation( const char *name,
        const char *default_name,
        struct q_location *location );

extern int q_locationtostr( struct q_location *location,
        unsigned int namelen,
        char *name );
```

5.2.1 q_strtolocation

This function converts the external representation of a location given in the **name** parameter into an equivalent internal representation stored in **location**. The additional input parameter, **default_name**, is used to supply any components missing from **name** that are necessary for the formation of **location**. For example, in TCP/IP, a "well known port number" supplied in the **default_name** parameter would be used if there were no explicit port number specified in **name**.

Input parameters:

- **name** — a null-terminated C string containing the implementation-defined external representation of a location.
- **default_name** — a null-terminated C string containing the implementation-defined external representation of a default location. Components missing in **name** will be taken from **default_name**.

Output parameters:

- **location** — the opaque 8-byte structure containing the implementation-defined internal representation of a location.

Function result:

- 0 if this function was able to perform the conversion, -1 if not, in which case the value in the output parameter is undefined.

5.2.2 q_locationtostr

This function converts the internal representation of a location given in the **location** parameter into an equivalent external representation that it stores in **name**. The additional input parameter, **namelen**, gives the number of bytes in **name** available for storing the result. The return value is the length of the string actually stored in **name**.

Input parameters:

- **location** — the opaque 8-byte structure containing the implementation-defined internal representation of a location.
- **namelen** — an unsigned int containing the number of bytes in the output parameter **name** allocated by the caller for storing the result. This value should be positive.

Output parameters:

- **name** — a null-terminated C string containing the implementation-defined external representation of the location given by **location**. The storage for this string must be allocated by the caller, and must contain **namelen** bytes.

Function result:

- the number of characters stored in string **name** by this function, excluding the null terminator. If this value is positive, it is the same as strlen(**name**) and will never be larger than **namelen** − 1. If this value is negative, it means that **namelen** was not big enough to hold the entire name, and its absolute value is the number of additional bytes needed to store the entire name as a null-terminated string. In this case, the first **namelen** − 1 bytes of the name will be stored in **name**, followed by the null-terminator.

5.3 Manager Mapping Function

The manager mapping function is used by client software to find the location of a manager for a file. The internal representation of a "location" is an opaque 8-byte value defined by the underlying network layer (see Section 2.2). This location can be used by the client software to establish a connection to the manager in order to communicate using the client-manager protocol. This function is fixed for a given implementation.

```
extern int getmanagerlocation( const char *filename,
        struct q_location *man_addr );
```

5.3.1 getmanagerlocation

This function is given as input a file name in the form of a null-terminated C string and computes as output the location of the manager that controls the metadata for the named file.

Input parameters:

- **filename** — the null-terminated C string containing the name of the file to be managed. The value of the file name is supplied by the client software wishing to utilize the file. If this parameter is NULL, the function should determine whether or not there will be only one manager for all files, regardless of their names. If so, it should return the internal representation of the location of that manager. If not, it should return an error indication. This enables the client software to connect once to a manager and to leave that connection open.

Output parameters:

- **man_addr** — the location of the manager.

Function result:

- 0 if this function was able to determine a manager location, -1 if not, in which case the value in the output parameter is undefined.

5.4 Access Control Information Gathering Function

The access control information gathering function enables an implementation to gather the necessary information about a user in order to establish and verify access rights to a file. This function is fixed for a given implementation.

```
extern unsigned long gatheruserinfo( char *here );
```

5.4.1 gatheruserinfo

This function is executed by the client software to gather whatever identification information about the running process must be sent to the manager in order to establish or verify the access rights this user has for parallel files.

Input parameters:

- **here** — the 8-byte aligned address of a storage area into which this function should store, in network byte order, the information it gathers. This storage area will, in fact, be in a buffer where the client software is building a message to send to the manager in order to open a file.

Output parameters:
None.
Function result:

- the number of bytes of data stored by this function in the area pointed to by **here**. This value is used by the client software when computing the length of the message to send to the manager.

5.5 File Striping Functions

The file striping functions enable a user to specify how logical blocks in a parallel file are mapped onto data components and relative block numbers within a data component. Each file must specify as part of its metadata a mapping function number and a thickness parameter. The mapping function number is used by the client to select the proper set of striping functions that apply to the file. The number itself is an arbitrary label used for purposes of identification only, and in no way implies any sort of coding or hashing in the functions themselves.

Each set must contain the following three functions, which are all executed by the client software:

```
extern void from_actual( unsigned int mapfun,
        unsigned int thickness, unsigned int nservers,
        unsigned int actual, unsigned int *serverno,
        unsigned int *blockno );

extern unsigned int to_actual( unsigned int mapfun,
        unsigned int thickness, unsigned int nservers,
        unsigned int serverno, unsigned int blockno );

extern unsigned int predict_stride( unsigned int mapfun,
        unsigned int thickness, unsigned int nservers,
        unsigned int startblock, unsigned int stride,
        unsigned int totalblocks, unsigned int this_stride[],
        unsigned int first[], unsigned int numblocks[],
        unsigned int order[] );
```

In these functions a server is refered to by its "ordinal number". If there are N servers specified in the metadata for this file, their ordinal numbers are given by the numbers $0 \ldots N-1$ according to the position of the server in the metadata list of servers.

5.5.1 from_actual

This function is given as input a logical block number and computes as output the number of the server on which this block is located, and its relative block number on that server.

Input parameters:

- **mapfun** — the number of the mapping function. Since each mapping function defines its own version of **from_actual**, this parameter may not be used. However, it's presence allows several mapping functions to share a single version of **from_actual** if that is convenient for the implementor. This value is obtained from the metadata of the file.
- **thickness** — the stripe thickness, which is the number of consecutive logical blocks that are mapped into consecutive relative blocks in each stripe. This value is obtained from the metadata of the file.
- **nservers** — the stripe width, which is the number of servers that are participating in the storage of this file. This value is obtained from the metadata of the file.
- **actual** — the logical block number in the file. This value is supplied by the client software that wants to do the mapping.

Output parameters:

- **serverno** — the ordinal number of the server on which the **actual** block resides.
- **blockno** — the relative block number on this server where the **actual** block resides.

Function result:
None.

5.5.2 to_actual

This function is given as input a server number and a relative block number, and computes as output the corresponding logical block number in the file. It is the inverse function of **from_actual**.

Input parameters:

- **mapfun** — the number of the mapping function. Since each mapping function defines its own version of **to_actual**, this parameter may not be used. However, it's presence allows several mapping functions to share a single version of **to_actual** if that is convenient for the implementor. This value is obtained from the metadata of the file.
- **thickness** — the stripe thickness, which is the number of consecutive logical blocks that are mapped into consecutive relative blocks in each stripe. This value is obtained from the metadata of the file.
- **nservers** — the stripe width, which is the number of servers that are participating in the storage of this file. This value is obtained from the metadata of the file.
- **serverno** — the ordinal number of a server. This value is supplied by the client software that wants to do the mapping and must be less than the value of **nservers**.
- **blockno** — a relative block number on this server. This value is supplied by the client software that wants to do the mapping.

Output parameters:
None.

Function result:

- the corresponding logical block number in the file.

5.5.3 predict_stride

This function determines if a request to read a set of logical blocks separated by a given "stride" value can be mapped into a set of requests, one for each server, in which there is a single "relative stride" value between relative blocks on that server (each server can have a different "relative stride" value).

Input parameters:

- **mapfun** — the number of the mapping function. Since each mapping function defines its own version of **predict_stride**, this parameter may not be used. However, it's presence allows several mapping functions to share a single version of **predict_stride** if that is convenient for the implementor. This value is obtained from the metadata of the file.
- **thickness** — the stripe thickness, which is the number of consecutive logical blocks that are mapped into consecutive relative blocks in each stripe. This value is obtained from the metadata of the file.
- **nservers** — the stripe width, which is the number of servers that are participating in the storage of this file. This value is obtained from the metadata of the file.
- **startblock** — the first logical block in the sequence. This value is supplied by the client software that wants to perform the request.
- **totalblocks** — the total number of logical blocks in the sequence, including the first. This value is supplied by the client software that wants to perform the request. A value of zero means "the rest of the file".

Output parameters:

- **this_stride** — an array with one entry for each server that should be set to the "relative stride" value for this server as determined by this function. The array is indexed by the server's ordinal number.
- **first** — an array with one entry for each server that should be set to the number of the first relative block on this server as determined by this function. The array is indexed by the server's ordinal number.
- **numblocks** — an array with one entry for each server that should be set to the number of relative blocks on this server as determined by this function. The array is indexed by the server's ordinal number. An entry with a zero value means that the corresponding server does not participate in this request. In the case when the value of input parameter **totalblocks** is 0, indicating "the rest of the file", then the non-zero values in this array correspondingly indicate "the rest of this data component" and should not be taken to mean "exactly this many blocks".
- **order** — an array with one entry for each server that should be set to the order of first use of blocks from this server in the request as determined by this function. The array is indexed by the server's ordinal number. The values in this array are in the range $0 \ldots K-1$, where K is the number of servers participating in this request as determined by the function result. The only items in this array containing valid values are those with the same subscripts (i.e., server numbers) as non-zero items in the **numblocks** array.

Function result:

- the number of servers participating in this request as determined by this function. If this value is zero, the request cannot be satisfied and the values in all the output parameters are undefined. This number should be the number of non-zero entries returned in the array **numblocks**.

5.6 Agent Cache Replacement Functions

Each agent may maintain a cache of blocks for the file it is serving. The size of this cache and the policies it follows can be controlled from the clients through the use of the SETCACHE operation. One of the parameters to the SETCACHE is a "cache replacement number". Each such number refers to a set of three functions used by the agent to enforce the corresponding policy. The system administrator can define new cache replacement policies by defining a set of functions that implement the new policies. The implementation will define the mechanism by which a set is associated with a number in the agent software.

Each set must contain the following three functions, which are all executed by the agent software:

```
extern void addition_function( struct buffer_record *new_buf,
        unsigned int policy, unsigned int n_in_cache,
        struct buffer_record **head,
        struct buffer_record **tail );

extern struct buffer_record *removal_function(
        unsigned int policy, unsigned int n_in_cache,
        struct buffer_record **head,
        struct buffer_record **tail );

extern void promotion_function( struct buffer_record *buf,
        unsigned int policy, unsigned int n_in_cache,
        struct buffer_record **head,
        struct buffer_record **tail );
```

The agent software maintains a doubly-linked list of buffers that constitutes its cache. Each implementation can define its own buffer structure which has the type "struct buffer_record" in the declarations above. In all of these functions, the first buffer in the cache list is pointed to by the parameter **head**, the last buffer in the list is pointed to by the parameter **tail**, the total number of buffers in the list is given by the parameter **n_in_cache**, and the replacement policy number is given by the parameter **policy**. The purpose of these functions is to maintain this list in the order required by the indicated policy. In general, each policy will be implemented by a different set of functions, so that the **policy** parameter will not be used. However, its presence allows a single function to implement several different policies at the convenience of the implementor.

Every implementation will supply a set of functions for three standard replacement policies: FIFO (**policy** = 1), LRU (**policy** = 2), and RANDOM (**policy** = 3). Note that these functions manipulate only the links which thread buffers on the cache list — they do not manipulate the contents of the buffers nor the hash table in any way. They specifically do not read or write buffers to disk, nor do they worry about whether buffers are "dirty" or not.

5.6.1 addition_function

This function is given as input a new buffer which it should add to the cache list at whatever position is appropriate for the indicated policy. For all three standard policies, the new buffer is added to the end of the list. This function must set the forward and backward links of the new buffer, must modify the links of its neighbors in the list, and may modify the values of the head and tail of the list, if appropriate.

Input parameters:

- **new_buf** — a pointer to the new buffer which is to be added to the cache list.
- **policy** — the number of the cache replacement policy.
- **n_in_cache** — the number of buffers in the cache before this new buffer is added (i.e., not including **new_buf**).

Input/Output parameters:

- **head** — a pointer to the first buffer in the cache list. The value pointed to will change if the new buffer is added at the front of the list.
- **tail** — a pointer to the last buffer in the cache list. The value pointed to will change if the new buffer is added at the end of the list.

Function result:
None.

5.6.2 removal_function

This function returns a buffer which it has removed from the cache list at whatever position is appropriate for the indicated policy. For the FIFO and LRU policies, this buffer is always removed from the front of the list. For the RANDOM policy, the buffer is removed from a random position in the list. This function must modify the forward and backward links of the buffer's neighbors in the list, and may modify the values of the head and tail of the list, as appropriate.

Input parameters:

- **policy** — the number of the cache replacement policy.
- **n_in_cache** — the number of buffers in the cache before this buffer is removed (i.e., including the buffer returned by this function).

Input/Output parameters:

- **head** — a pointer to the first buffer in the cache list. The value pointed to will change if the new buffer is added at the front of the list.
- **tail** — a pointer to the last buffer in the cache list. The value pointed to will change if the new buffer is added at the end of the list.

Function result:

- **new_buf** — a pointer to the buffer which has been removed from the cache list, or NULL if the list was already empty.

5.6.3 promotion_function

This function is called whenever the agent references a buffer that is already in the cache. It given as input a pointer to that buffer, which is already in the cache list and which should be moved within the list to whatever position is appropriate for the indicated policy. For the FIFO and RANDOM policies, the buffer is left where it is. For the LRU policy, the buffer is moved to the end of the list. This function must modify the forward and backward links of the buffer and of its neighbors in the list, and may modify the values of the head and tail of the list, as appropriate.

Input parameters:

- **buf** — a pointer to a buffer which is already in the cache list and which is to be moved within the list.
- **policy** — the number of the cache replacement policy.
- **n_in_cache** — the number of buffers in the cache (i.e., including **buf**).

Input/Output parameters:

- **head** — a pointer to the first buffer in the cache list. The value pointed to will change if the buffer is moved to the front of the list and it was not already there.
- **tail** — a pointer to the last buffer in the cache list. The value pointed to will change if the buffer is moved to the end of the list and it was not already there.

Function result:
None.

6 Protocols

This section describes the four protocols defined in BPFS. For simplicity, all four protocols use the same message format consisting of a fixed-size header part followed by a variable length data part. In order to simplify the buffering and caching schemes, the length of any data part must be no greater than the system-wide BLOCKSIZE. Since BLOCKSIZE is expected to be at least 4096 bytes, this restriction does not appear to impose a severe limitation.

6.1 The Message Header

The **message header** always contains 24 bytes broken into 7 fields as follows (the "normal" function of each field is indicated – the exact use of a field depends on the particular protocol):

- 4 bytes — the **total length** (in bytes) of the entire message (≥ 24)
- 2 bytes — the **operation code** of the message.
- 2 bytes — an **open file identification** for the file to which this message refers.
- 4 bytes — a **message identification code** for this message.
- 4 bytes — the **block number**.
- 4 bytes — the **block length**.
- 4 bytes — **more blocks**, an indication of the number of blocks in an operation.

All 7 fields contain unsigned integral values. When sent or received over a network, the value in each field of the message header must be in "network byte order". The function **htons** (**htonl**) must be used to transform 2 (4) byte integer values from the host byte order into the network byte order before sending. The function **ntohs** (**ntohl**) must be used to transform 2 (4) byte integer values from the network byte order into the host byte order upon reception.

In most cases a communication is initiated by one "request" message sent from component "A" to component "B", and terminates by one "reply" message from "B" back to "A". One of the protocol conventions is that a reply message always has an operation code value of zero (0) if the request was successfully satisfied, and non-zero if there was an error. The identification code field is used to match a reply with a request if necessary.

6.2 Message Operation Codes

There are currently 15 defined operation codes, although not all codes are used by all four protocols. The following table indicates the possibilities and their general use:

- ABORT – abort a previous request.
- CLOSEFILE – close a previously opened parallel file.
- CONFIRM – confirm a previous request.
- DELETEFILE – delete a parallel file.
- ERASEFILE – erase a parallel file or fragments of a parallel file.
- GETSTATUS – get status information about a parallel file.
- LINKFILE – link an existing parallel file to an additional name.
- MOVEFILE – change the name of an existing parallel file.
- RDWRBLOCK – open a parallel file for reading and writing simultaneously.
- READBLOCK – open a parallel file for reading, or read a block of data.
- SETCACHE – get or set agent cache parameters for an open file.
- STARTSTREAM – start streaming a parallel file.
- STOPSTREAM – stop streaming a parallel file.
- SYNCFILE – force onto disk all data previously written to a parallel file.
- WRITEBLOCK – open a parallel file for writing, or write a block of data.

6.3 The Protocol between Client and Manager

The connection between the client and the manager is established when the client does a connect to the "location" returned by the implementation-defined "getmanagerlocation" function (see Section 5.3). The input parameter to this function is the name of the file that the client wishes to process. This connection can be kept open for as long as the client desires. Either it can be closed at the end of each operation and reopened at the start of the next (even if on the same file), or it can be opened once and kept open until the client terminates.

The manager is "stateless" because it keeps no record of past operations on behalf of clients, other than the "permanent" metadata it stores about files. In particular, the manager does not keep a record of which files are open — this is the task of the servers.

There are nine different operations in the protocol between client and manager. These can be grouped into two general classes: five self-contained operations to manage a file (GETSTATUS, DELETEFILE, ERASEFILE, LINKFILE, MOVEFILE), and four operations to open or close the file for subsequent data transfer (READBLOCK, WRITEBLOCK, RDWRBLOCK, CLOSEFILE). The five management operations are discussed first, since they are simpler. Once a file has been opened successfully by a client, the only one of the five management operations allowed on that file by any client is GETSTATUS. In other words, once a file is "in use" by some client, no client is allowed to change or remove any of the metadata for that file. This restriction is enforced by the servers, since the manager is stateless and therefore does not know if a file is "in use".

All nine commands in this protocol start with a request from a client to a manager and finish when the client receives a reply back from the manager. The client creates a unique identification code for each operation that is sent in the request and returned in the reply so that the reply can be matched with the request.

For the five management commands, a request message consists of the 24-byte header followed by a variable length data part. This data part contains two null-terminated C strings, one immediately following the other. The first string is the "old" name of the existing file to which this command refers. The second string is the "new" name required only by the LINKFILE and MOVEFILE commands. This string will be empty (i.e., it will consist of just the null terminator) for the GETSTATUS, DELETE, and ERASE commands.

The general format of the request header for the five management commands is:

Field	Value	Meaning
total length	$24 + n + 2$	$n = strlen(old) + strlen(new)$
op code	$\neq 0$	one of 5 management op codes
file id	0	unused
id	identification	to match reply with request
block number	0	unused
block length	BLOCKSIZE	number of bytes per block in file
more blocks	0	unused

The manager will process this request and will always send back a reply that indicates either "success" or "failure" of the operation. The general format of a reply indicating "failure" consists of a 24-byte header followed by the text of an error message describing the failure. The general format of the "failure" reply header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	requested op was not successful
file id	0	unused
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

The data portion of a "failure" reply contains the text of an error message in the form of a null-terminated C string. In case of multiple errors, the manager arbitrarily chooses one to report to the client.

The format of a reply indicating "success" depends on the type of request to which it is a reply. For the GETSTATUS command it consists of a 24-byte header followed by the status information returned by each server as explained in Section 6.3.1 below. For the four commands DELETEFILE, ERASEFILE, LINKFILE, and MOVEFILE, it consists of just a 24-byte header. The general format of the "success" reply header for these four commands is:

Field	Value	Meaning
total length	24	length of header
op code	0	requested op was successful
file id	0	unused
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

6.3.1 GETSTATUS

This function allows a client to retrieve status information about a parallel file. This information includes:

- The stripe thickness used to map blocks onto servers.
- The number of servers on which data components of this file reside.
- For each data component, its location, file protection codes, and size.

In a GETSTATUS request the "new" name in the variable length data part is empty.

Upon receipt of a GETSTATUS request, the manager will read the metadata for the file named in the request, will forward the request to all the servers mentioned in the metadata, and will wait for their replies. Each server is expected to determine if its data component exists, and to reply with its current size and protection codes on that server. If all the servers reply with this information, the manager will format and send a "success" reply back to the client. If any of the servers is unable to obtain status information about its data component, the manager will send a "failure" reply back to the client.

For the GETSTATUS request, the "success" reply header is followed by an array of 24-byte items, one for each of the N servers. The format of the GETSTATUS "success" reply header is:

Field	Value	Meaning
total length	$24 + N \times 24$	length of header plus array
op code	0	requested op was successful
file id	0	unused
id	id of request	to match reply with request
block number	N	the number of servers used by the file
block length	≥ 1	the stripe thickness of the file
more blocks	≥ 1	the mapping function used by the file

Each 24-byte item in the array contains the following structures:

- 8 bytes — the location of this server.
- 2 bytes — the protection codes on this server.
- 2 bytes — unused.
- 4 bytes — the number of full blocks on this server.
- 4 bytes — the number of extra bytes in any partial last block.
- 4 bytes — unused.

The internal representation of a server "location" is an opaque value defined by the implementation of the underlying network layer (see Section 2.2). All the other items are unsigned integral values in network byte order. The size of the data component on each server is expressed as the number of full blocks plus the number of extra bytes in any partial last block. If a file is striped in a regular fashion over N servers numbered $0 \ldots N-1$, the total size (in bytes) of the parallel file is given as:

$$\sum_{i=0}^{N-1} (full\ blocks_i \times BLOCKSIZE + extra\ bytes_i)$$

Note that if the file uses an irregular mapping function, such as the pseudo-random mapping, then there may well be unused blocks included in the size of each data component, and the sum given above will be larger than the actual amount of data stored in the parallel file.

6.3.2 DELETEFILE

This function exists to delete all the metadata and data components of a parallel file. The difference between this operation and ERASEFILE is that DELETEFILE succeeds only if the parallel file is correctly constructed and would be accessible (i.e., all the servers are currently up and running). If any error is detected, such as missing metadata or an undeletable data component, nothing is deleted and the error is reported back to the client.

ERASEFILE always erases whatever components of a parallel it can find. For example, if one of the data components indicated in the metadata is missing or its server is inaccessible, the rest of the data components and the metadata are nonetheless deleted. If the metadata itself is missing, the manager will contact all the servers it knows about to erase any lingering data components. ERASEFILE is useful for cleaning up when bugs or system crashes lead to incorrectly constructed parallel files.

In a DELETEFILE request the "new" name in the variable length data part is empty.

Upon receipt of a DELETEFILE request, the manager will read the metadata for the file named in the request, will forward the request to all the servers mentioned in the metadata, and will then wait for their replies. If all the servers reply that they would be able to delete their data component of this file (or that it does not exist), the manager deletes the metadata and sends a confirm message to all of the servers and awaits their replies to that. It is only upon receipt of this confirm from the manager that each server will actually delete the file and send another (final) reply back to the manager indicating the delete has been performed. Once this final reply has been received from all the servers the manager will format and send a "success" reply back to the client.

If any of the servers determines that it is unable to delete its data component, the manager will send an abort message to all the other servers that indicated they would be able to perform the delete (so that none of them will actually delete an existing data component). The manager will then send a "failure" reply back to the client. In this case neither the metadata nor any of the data components are deleted.

6.3.3 ERASEFILE

This function exists to erase (i.e. delete) all the metadata and data components of a parallel file. The difference between this operation and DELETEFILE is that ERASEFILE will erase components even if the parallel file is not correctly constructed and would not be usable by other commands (for example, because one of the servers is down).

ERASEFILE always erases whatever components of a parallel it can find. For example, if one of the data components indicated in the metadata is missing or its server is inaccessible, the rest of the data components and the metadata are nonetheless deleted. If the metadata itself is missing, the manager will contact all the servers it knows about to erase any lingering data components. ERASEFILE is useful for cleaning up when bugs or system crashes lead to incorrectly constructed parallel files.

In an ERASEFILE request the "new" name in the variable length data part is empty.

Upon receipt of an ERASEFILE request, the manager will read the metadata for the file named in the request, will forward the request to all the servers mentioned in the metadata, and will then wait for their replies. If the metadata does not exist for this file, the manager will forward the request to all the servers to which it is connected. If all the servers reply that they succeeded in erasing their data component of this file, or that it does not exist, the manager will format and send a "success" reply back to the client. If any of the servers is unable to erase its data component, or if the manager is unable to erase the metadata component (in both cases this could be caused by the client's lack of permission to delete the corresponding file), the manager will send a "failure" reply back to the client. However, unlike DELETEFILE, in the failure case the metadata and as many of the data components as possible are erased by ERASEFILE.

6.3.4 LINKFILE

This function exists to link an existing parallel file to a new name. It is an error if any metadata or data components for the new name exist prior to this operation. (In UNIX parlance this is a "link" to the file. Most people would call it a "synonym" or "alias".)

In a LINKFILE request both the "file" name and "new" name strings in the variable length data part must each contain at least 1 character.

Upon receipt of a LINKFILE request, the manager will read the metadata for the file named in the request, and will attempt to link that metadata component to the new name. If the metadata does not exist for this file, or if the manager cannot link the existing metadata component to the new name, the manager will immediately return a "failure" reply to the client without contacting the servers. Otherwise, the manager will forward the request to all the servers mentioned in the metadata, and will then wait for their replies. If all the servers reply that they succeeded in linking their data component of this existing file to the new name, the manager will send a confirm message to all of the servers and will then await their replies to that. When this final reply has been received from all the servers, the manager will format and send a "success" reply back to the client.

If any of the servers is unable to link its data component to the new name, the manager will send an abort message to all of the other servers that indicated success and will send a "failure" reply back to the client. Upon receiving an abort message a server will unlink (delete) the link it just created to the new name. A server would not be able to create a link, for example, if either there is no data component with the indicated file name, or a component with the new name already exists.

6.3.5 MOVEFILE

This function exists to change the name of an existing parallel file. It is an error if any metadata or data components for the new name exist prior to this operation. (In UNIX parlance this is a "move" of the file. Most people would call it a "rename".)

In a MOVEFILE request both the "file" name and "new" name strings in the variable length data part must each contain at least 1 character.

Upon receipt of a MOVEFILE request, the manager will read the metadata for the file named in the request, and will attempt to link that metadata component to the new name. If the metadata does not exist for this file, or if the manager cannot link the existing metadata component to the new name, the manager will immediately return a "failure" reply to the client without contacting the servers. Otherwise, the manager will forward the request to all the servers mentioned in the metadata, and will wait for their replies. If all the servers reply that they succeeded in linking their data component of this existing file to the new name, the manager will send a confirm message to all of the servers and will then await their replies to that. It is only upon receipt of this confirm message that each server will delete the old name and will then reply to the manager once again. When this final reply has been received from all the servers, the manager will format and send a "success" reply back to the client.

If any of the servers is unable to link its data component to the new name, the manager will send an abort message to all of the servers that indicated success and will send a "failure" reply back to the client. Upon receiving an abort message a server will unlink (delete) the link it just created to the new name, but will leave the existing data component unchanged. A server would not be able to create a link to the new name, for example, if either there is no data component with the indicated file name, or a component with the new name already exists.

6.3.6 READBLOCK, WRITEBLOCK, RDWRBLOCK

In the protocol between the client and the manager, these three operations are sent by the client in order to open a parallel file for reading, for writing, and for updating (i.e., reading and writing simultaneously) respectively.

A file opened with READBLOCK becomes "shared read only" — it can also be opened simultaneously by the same or other clients using READBLOCK, but it cannot be opened for writing or updating, and any attempt to do so will be refused by the servers.

A file opened by WRITEBLOCK becomes "exclusive write only". As long as it remains open, all other attempts to open it in any mode are refused by the servers.

A file opened by RDWRBLOCK becomes "shared read and write" (i.e., shared updating). As long as it remains open, other attempts to open it must also be with RDWRBLOCK or they will be refused by the servers.

These restrictions are summarized in the following table.

	New Attempt to Open as		
Now Open as	READBLOCK	WRITEBLOCK	RDWRBLOCK
READBLOCK	ok	refused	refused
WRITEBLOCK	refused	refused	refused
RDWRBLOCK	refused	refused	ok

The general format of the request header for an open is:

Field	Value	Meaning
total length	$24 + n + K$	$n = (strlen(name) + 2 + 7)\& \sim 7$
op code	$\neq 0$	one of 3 open op codes
file id	≥ 0	protection codes
id	identification	to match reply with request
block number	≥ 0	attributes
block length	BLOCKSIZE	number of bytes per block in file
more blocks	n	offset to metadata parameters

In this header, the attributes value stored in the "block number" field is treated as a sequence of four independent bytes, and hence is not converted to/from network byte order when sending/receiving (because independent bytes are always in network order). At present only 2 of these bytes are actually used:

- Byte 0 is the "truncate" attribute which can have 2 values:

  0 indicating "do not truncate" an existing file on open.

  1 indicating "truncate" an existing file on open.

  This attribute is utilized only when opening with WRITEBLOCK and is ignored by the other opens. Truncation means that all data in an existing file is erased at the time of the open, so that the file size becomes 0.

- Byte 1 is the "create" attribute which can have 3 values:

  0 indicating "never create" the file if it does not already exist.

  1 indicating "always create" the file if it does not already exist.

  2 indicating "create as needed", which creates the file if and only if it does not already exist.

  This attribute is ignored by READBLOCK, since a file to be read must always exist. The servers will report errors if this attribute is "never create" and the file does not already exist, or if this attribute is "always create" and the file does already exist.

The variable length data part of an open request message contains the file name (which must contain at least 1 character) as a null-terminated C string, followed by an empty null-terminated C string, followed by a data structure containing metadata parameters, followed by a structure containing implementation-defined information to identify the current user. Each of these structures is aligned on the first 8-byte boundary after the previous information in the message. The byte offset to the start of the first structure within the data part is indicated by n in the "more blocks" field of the header shown above. The total length of both structures (indicated by K included in the "total length" field of the header shown above) is $K = ((12 + strlen(list) + 1 + 7)\& \sim 7) + M$, where M is the length of the second structure and *list* is a field in the first structure, as described below.

The first structure contains metadata parameters that will actually be used by the manager if and only if a new file is created by this open. However, this information must always be sent as part of every open request. The fields in this structure are described next:

- a 4-byte unsigned integer giving the number of the mapping function to use when declustering logical blocks onto the servers. This value is 0 to indicate the default mapping (1). Functions defined so far are:

  1. regular striping with each stripe having a "thickness" given by the next value. The "width" of the stripe (i.e., the number of servers participating in storing the data) is given by the value following that.
  2. pseudo-random placement with a "thickness" given by the next value and a "width" given by the value following that.

- a 4-byte unsigned integer giving the "thickness" parameter to the mapping function. This is the number of consecutive blocks per server in each stripe across the disks. This value is 0 to indicate the default thickness (1).

- a 4-byte unsigned integer giving the "stripe width", which is the number of servers on which data components of this file are stored. This value is 0 to indicate the default width, which means all the servers in the following list if that list is not empty, otherwise it is all the servers currently available to the manager.

- a null-terminated C string containing a comma- or blank- separated list of the names of servers to be used to store data components of this file. If this list is empty, the data is striped across "stripe width" servers selected from the list of all servers currently available to the manager. In this case it is an error if "stripe width" is greater than the number of currently available servers. If this list is not empty, the data is striped across the first "stripe width" servers in this list. In this case it is an error if "stripe width" is greater than the number of servers in this list. The form of a server "name" depends on the underlying network layer used by BPFS. When the network layer is TCP/IP, a "name" is either the DNS name or the IP address in dotted decimal notation, followed by an optional port number that is separated from the name by a colon (for example, "magenbitter:3775" or "140.77.11.38:3775").

The second structure is an M-byte block of data containing user identification information that will allow the manager to establish (for new files) or verify (for existing files) the access rights of this user for this file. This data is gathered and put into the message by the implementation-defined function "gatheruserinfo", which returns M, the total number of bytes in this block of information. Note that this structure always begins on the first 8-byte boundary following the null-terminator that ends the list in the first structure.

When the manager receives an open request, it will attempt to read the metadata for an existing file, or create the metadata for an new file using the metadata parameters supplied in the request. It then forwards the request to all the servers and waits for their replies. Each server must attempt to open the file "non destructively", which means that no data should be lost as a consequence of this open. This implies that a new file can be created, but an existing file should not yet be truncated by this open.

If all the servers reply that they were able to open the file in the correct mode (except for truncation), the manager will send back a confirm message if the "truncate" attribute was set and the operation was WRITEBLOCK. It is only upon receipt of this confirm that an existing file is truncated by the server and another reply is sent back to the manager. If a truncation is not needed, or when all the additional replies after a truncation have been received by the manager, the manager will send a "success" reply back to the client.

If any server replies that it was unable to open the file, the manager will send an abort message to all the other servers that were able to open the file, will delete the metadata component if it was newly created by this request, and will then send a "failure" reply back to the client. Upon receipt of an abort in this case, the server will simply close the file, deleting it if it was newly created. Note that since the first open did not truncate an existing file, no data will be lost should it have to be closed as a result of an abort. In this way file consistency is maintained across all servers containing data components of a single parallel file.

The format of the "failure" reply to an open is the same as that returned by any of the five management operations explained previously on page 22: a 24-byte header followed by the text of an error message. The format of the "success" reply to an open is the same as that returned by the GETSTATUS operation explained previously in Section 6.3.1 on page 23: a 24-byte header followed by an array of 24-byte items, one

for each server participating in storing this file. However, the contents of 24-byte array items are different, as shown in the following:

- 8 bytes — the location of agent handling this file on this server.
- 2 bytes — unused.
- 2 bytes — open file id for this file on this server.
- 4 bytes — unused.
- 4 bytes — unused.
- 4 bytes — unused.

Note that the "location" field is now the internal representation of the location of the agent that is handling this file on behalf of the server, not the location of the server as it was in the GETSTATUS reply. The client must use each of these agent "location" values to establish direct connections to the agents for this file running on each server node. All data will then flow over these direct connections between client and agent. The "open file id" must be copied into the header of all messages between client and agent in order to identify (redundantly) the file to which the message refers.

6.3.7 CLOSEFILE

In order to close a previously opened parallel file, the client must first close its connections to all the file's agents, and then send a CLOSEFILE request to the manager. This request will be forwarded to all the servers by the manager. Upon receipt of a reply from all the servers, the manager will send a reply back to the client.

The format of a CLOSEFILE request consists of the usual 24-byte header followed by a variable length data part. The request header has the same format as that for the other requests from a client to a manager, except that the "block number" and "more blocks" fields now contain useful information, as shown next:

Field	Value	Meaning
total length	$24 + n + 24 \times N$	$n = (strlen(name) + 2 + 7)\& \sim 7$
op code	$CLOSEFILE$	close an open file
file id	0	unused
id	identification	to match reply with request
block number	N	number of servers in use
block length	BLOCKSIZE	number of bytes per block in file
more blocks	n	offset to array

The variable length part of a close request consists of two null-terminated C strings (the first containing the file name, the second is empty), followed by an array of 24-byte items, one for each of the N servers. This array starts on the next 8-byte boundary after the null character that terminates the second (empty) string. The byte offset to the start of this array within the data part is indicated by n in the "more blocks" field of the header shown above.

The items in this array have the same format and value as the items obtained earlier by this client in the "success" reply to the request that opened this file. Only the "location" and "open file id" fields contain useful information, the other fields should contain zeroes.

For both the "success" and "failure" cases, the reply to a CLOSEFILE sent by a manager back to a client has a format identical to the format of a reply to any of the four management commands DELETEFILE, ERASEFILE, LINKFILE, and MOVEFILE, as discussed previously on page 22.

6.4 The Protocol between Client and Agent

This protocol begins when the client connects to an agent using the "location" information returned to the client by the manager in a "success" reply to an open request. That reply contains information for a separate connection between the client and each agent. The protocol ends when the client closes this connection prior to sending a CLOSEFILE request to the manager. All data flow between a file and a client is accomplished by this protocol. All the operations in this protocol identify the file to the agent by using an "open file id" that was returned to the client by the manager in the "success" reply to the open.

There are six operations in this protocol:

1. READBLOCK — to request a set of blocks to be read from the disk by the agent and sent to the client.
2. WRITEBLOCK — to send one block from the client to the agent for writing onto the disk.
3. STARTSTREAM — to initiate a "stream" whereby the agent will read blocks from the disk and send them to the client without specific READBLOCK requests by the client.
4. STOPSTREAM — to stop a "stream".
5. SYNCFILE — to force onto the disk of the agent all blocks previously sent by WRITEBLOCK requests.
6. SETCACHE — to obtain or change parameters effecting the agent's use of a buffer cache on an open file.

6.4.1 READBLOCK

This operation is sent from the client to the agent in order to request that a set of blocks be read from the disk by the agent and sent back to the client. It begins with a request from the client, and ends when the last block requested is sent by the agent. It is acceptable to an agent only if the file was previously opened successfully in READBLOCK or RDWRBLOCK mode.

A READBLOCK request message consists of just the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	READBLOCK	read N blocks
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	≥ 0	relative number of block
block length	BLOCKSIZE	number of bytes per block in file
more blocks	$N (\geq 0)$	total number of blocks to read

With this message the client is requesting the agent to read and send back N blocks, where N is the value in the "more blocks" field. Even if this value is zero, at least one block will be sent back. Each block will be sent in a separate reply message, all of which will contain identical copies of the "file id" and "id" fields from the request. The "file id" field contains the open file identification code returned to the client by the manager in the "success" reply to the open request that established this connection. The "id" field is an arbitrary value defined by the client to match replies with the request. The first block is read from disk using the value in the "block number" field as the relative block number for this agent. Successive blocks are taken from successive relative blocks. The agent will not accept any more input requests from the client until the last of these successive blocks has been sent back to the client.

Each reply consists of a 24-byte header followed by a variable length data part.

For a "failure" reply, the variable length part contains the error message as a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	READBLOCK was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	≥ 0	relative number of block in error
block length	0	no data read
more blocks	0	no more blocks follow

For a "success" reply, the variable length part contains the actual bytes of user data, which can never be more than BLOCKSIZE bytes, and will usually be exactly BLOCKSIZE bytes, except possibly for the last block in the file, which may be shorter. If an end of file is read, the variable length part will be empty. The length of this variable length part is returned in the "block length" field of the reply header (so that a value of 0 means end of file). The "more bytes" field in the reply indicates the number of blocks that are expected to follow this block as part of the same request. Therefore, a "success" reply containing the last block in a sequence (including a sequence cut short by reading an end of file) will always have a zero in this field. The format of the "success" header is:

Field	Value	Meaning
total length	$24 + N$	N = bytes of data read successfully
op code	0	READBLOCK was successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	≥ 0	relative number of block read
block length	$N(\geq 0)$	bytes of data read successfully
more blocks	≥ 0	number of blocks to follow

After a "success" reply indicating end of file, or after any "failure" reply, the agent will stop sending replies to this READBLOCK request, regardless of how many blocks were left unread, and begin accepting new requests again from this client.

6.4.2 WRITEBLOCK

This operation is sent from the client to the agent in order to request that a block of data contained in the request be written to the disk by the agent. It is acceptable to an agent only if the file was previously opened successfully in WRITEBLOCK or RDWRBLOCK mode.

A WRITEBLOCK request message consists of the 24-byte header followed by a variable length part containing the data bytes to be written to disk by the agent. The length of this variable length part must be greater than zero and not greater than BLOCKSIZE. Except for the last block in a file, it should always be exactly BLOCKSIZE. The header format for a WRITEBLOCK request is as follows:

Field	Value	Meaning
total length	$24 + N$	N = number of bytes of data
op code	WRITEBLOCK	write one block of N bytes
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	≥ 0	relative number of block
block length	$N(> 0)$	number of bytes of data
more blocks	0 or 1	0 for reply, 1 for none

The "more blocks" field contains 0 if the client wishes to receive a reply from the agent indicating the success or failure of the write operation. If "more blocks" is 1, the agent does not send a reply to this request whether it succeeds or fails. The "file id" field contains the open file identification code returned to the client

by the manager in the "success" reply to the open request that established this connection. The "id" field is an arbitrary value defined by the client to match the reply with the request. The block is written to disk using the value in the "block number" field as the relative block number for this agent.

If a reply is indicated, it consists of a 24-byte header followed by a variable length data part.

For a "failure" reply, the variable length part contains the error message as a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	WRITEBLOCK was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	≥ 0	relative number of block in error
block length	0	no data written
more blocks	0	reply requested

For a "success" reply, the variable length part is empty. The format of the "success" header is:

Field	Value	Meaning
total length	24	length of header
op code	0	WRITEBLOCK was successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	≥ 0	relative number of block written
block length	$N(> 0)$	bytes of data written successfully
more blocks	0	reply requested

6.4.3 STARTSTREAM

This operation is sent from the client to the agent in order to put the agent into "stream" mode. In this mode, the agent reads blocks from the disk and sends them to the client without receiving explicit READBLOCK requests from the client. In stream mode an agent continues to send blocks to the client until either the end of file is read, an error occurs, the total number of blocks specified in the STARTSTREAM are sent, or the agent receives a STOPSTREAM request. A client can have no more than one stream per file active at any time, and while a stream is active on a file no other read operations can be requested for that file. The STARTSTREAM operation begins with a request from the client, and ends when a reply is sent back by the agent. It is acceptable to an agent only if the file was previously opened successfully in READBLOCK mode.

The STARTSTREAM request message from the client to the agent consists of the 24-byte header followed by a 24-byte data part. The header for a STARTSTREAM request is in the following format:

Field	Value	Meaning
total length	48	fixed length of header and data
op code	STARTSTREAM	start a stream
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	≥ 0	first relative block in stream
block length	BLOCKSIZE	number of bytes per block in file
more blocks	≥ 0	number of blocks in stream

The data part of a STARTSTREAM request contains 24 bytes of information in the following order:

1. a 4-byte unsigned integer giving the "stride" value to be used to generate successive relative block numbers in the stream. If this value is 0 the agent will use a stride of 1.

2. a 4-byte unsigned integer giving the numerator of the "rate fraction".

3. a 4-byte unsigned integer giving the denominator of the "rate fraction".

4. a 4-byte unsigned integer giving the relative order of this server in this stream.

5. a 4-byte unsigned integer giving the total number of servers participating in this stream.

6. a 4-byte unsigned integer giving the id to be used by the agent when sending blocks back to the client as part of this stream.

Upon receiving a STARTSTREAM request, the agent will take appropriate action to set up a stream back to the client. The first block to be read from disk as part of the stream is that indicated by the relative block number in the "block number" field of the STARTSTREAM request header. The relative block number of each successive block in the stream is computed by adding the "stride" value to the block number of the previous block. The "more blocks" field in the header indicates the total number of blocks to send. However, if this field is 0, then the stream continues to the end of file. The rate at which blocks are sent is determined from the fraction obtained by dividing the "numerator" field by the "denominator" field. This rate gives the number of blocks per second to be sent by the agent. If either the "numerator" or "denominator" fields are zero, blocks will be send "as fast as possible". Note that in order to express a rate in terms of bytes per second, the "numerator" field should contain this rate and the "denominator" field should contain BLOCKSIZE.

The "relative order" and "total number of servers" fields in the data part are used to compute the delay for the first block in the stream as a fraction of the delay interval between successive blocks determined from the rate. The "relative order" is a value in the range $0 \ldots N-1$, where N is the "total number of servers". If either the "server order" or the "total number of servers" fields is zero, the first block will be sent out with no delay. These two fields are ignored if blocks are being sent "as fast as possible". The accuracy of all timings will clearly depend on the granularity of time resolution available to the agent and the ability of the implementation to meet real-time deadlines.

Blocks sent back to the client by the agent contain in the "id" field of their header the identification code sent by the client to the agent as the last field of the data part of the STARTSTREAM message. This enables the client to establish whatever identification is appropriate to recognize the data blocks in this stream. The "file id" field will contain a copy of the "file id" field from the STARTSTREAM request header. The rest of the fields in the data block headers of a stream will be set identically to the fields in the data block headers sent as replies to a READBLOCK request.

If the stream is successfully set up, the agent sends back a "success" reply consisting of just the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	0	STARTSTREAM was successful
file id	≥ 0	same as in request
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

The first data block from this stream will follow this "success" reply, possibly delayed by an appropriate amount of time as determined by the rate information of the stream.

If the agent encounters an error when attempting to set up a stream, it sends back a "failure" reply consisting of a 24-byte header followed by a variable length data part containing the text of an error message as a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	STARTSTREAM was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

Clearly no data blocks from this stream will follow a "failure" reply.

6.4.4 STOPSTREAM

This operation is sent from the client to the agent in order to stop any previously started stream on this file. This operation begins with a request from the client, and ends when a reply is sent back by the agent. Such a request is acceptable to an agent only if the file was previously opened successfully in READBLOCK mode. To be successful, it does not matter whether there is currently a stream in progress between this agent and this client or not, but if there is, it will be stopped. The format of the STOPSTREAM request is a message containing only the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	STOPSTREAM	stop a stream
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

If the STOPSTREAM is successful, the agent sends back a "success" reply consisting of just the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	0	STOPSTREAM was successful
file id	≥ 0	same as in request
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

After receiving this "success" reply, a client can be sure that it will not receive any additional data blocks from this stream.

If the agent encounters an error when attempting to perform a STOPSTREAM, it sends back a "failure" reply consisting of a 24-byte header followed by a variable length data part containing the text of an error message as a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	STOPSTREAM was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

6.4.5 SYNCFILE

This operation is sent from the client to the agent in order to force onto disk all blocks previously written to the indicated file. This will flush the cache for this file in the agent, and will also cause the agent to perform whatever operations are necessary on the server node to ensure that previously written blocks are in fact on permanent disk storage. In POSIX this is refered to as "synchronized I/O data integrity completion".

This operation begins with a request from the client, and ends when a reply is sent back by the agent. It is acceptable to an agent only if the file was previously opened successfully in WRITEBLOCK or RDWR-BLOCK mode.

The SYNCFILE request message from the client to the agent consists of just the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	SYNCFILE	force written blocks to disk
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

Upon receiving a SYNCFILE request, the agent will take appropriate action to flush its buffer cache (if any) for this file, and then force all previously written data blocks for this file onto the permanent disk storage. If this is successful, the agent sends back a "success" reply consisting of just the 24-byte header in the following format:

Field	Value	Meaning
total length	24	number of bytes in header
op code	0	SYNCFILE was successful
file id	≥ 0	same as in request
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

If the agent encounters an error when attempting to perform a SYNCFILE, it sends back a "failure" reply consisting of a 24-byte header followed by a variable length data part containing the text of an error message as a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	SYNCFILE was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

6.4.6 SETCACHE

This operation is sent from the client to the agent in order to obtain or change the parameters used by the agent to perform buffer caching on an open file.

This operation begins with a request from the client, and ends when a reply is sent back by the agent. It is acceptable to an agent only if the file was previously opened successfully in any mode.

The SETCACHE request message consists of the 24-byte header followed by a 24-byte data part. The header format is:

Field	Value	Meaning
total length	48	length of header and data part
op code	SETCACHE	set agent cache parameters
file id	≥ 0	open file id of this file
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

The data part is an array containing 6 4-byte unsigned integer values in the following order:

- an option to determine the action taken by the agent in response to this request. It can have the following values:

  0 turns agent caching off. In this case the rest of the items in this array are ignored by the agent.

  1 turns agent caching on.

  2 leaves agent caching unchanged. In this case the rest of the items in this array are ignored by the agent.

- a value indicating the "cache replacement strategy". It can have the following values:

  0 leave unchanged or use the implementation-defined default replacement strategy.

  1 use a "First-In, First-Out" (FIFO) replacement strategy.

  2 use a "Least Recently Used" (LRU) replacement strategy.

  3 use a "RANDOM" replacement strategy.

- a value indicating the "cache write-through policy". It can have the following values:

  0 leave unchanged or use the implementation-defined default write-through policy.

  1 use the policy that every write into the cache is also immediately written to disk ("write through").

  2 use the policy that a buffer is written to disk only when it is about to be replaced in the cache ("write behind").

- a value indicating the number of hash bins to use for the cache. A value of 0 means to leave this value unchanged or use the implementation-defined default.

- a value indicating the maximum number of buffers to keep in the cache. A value of 0 means to leave this value unchanged or use the implementation-defined default.

- currently unused.

Upon receiving a SETCACHE request, the agent will take the action indicated by the option value in the first item of the array. If the option is to leave caching unchanged, the agent will simply send back a "success" reply indicating the current status of the cache.

If the option is to turn caching off, any dirty buffers currently in the cache will be flushed to disk before the agent replies to this command.

If the option is to turn caching on when it is off, then any items in the array containing a value of zero are interpreted to mean "use the implementation-defined default".

If the option is to turn caching on and it is already on, then any items in the array containing a value of zero are interpreted to mean "leave the current value unchanged". Each value changed will require appropriate action by the agent before it sends a reply back to the client. If the "write through" policy changes from "write behind" to "write through", the agent must flush the cache. If the number of hash bins changes, all the buffers in the cache must be re-hashed. If the maximum number of buffers to keep in the cache decreases, the agent may have to remove some buffers from the cache to get below the new maximum.

Note that an agent services all clients accessing the same file, so it is possible that different clients will have different caching needs. BPFS offers no solution to this problem, since each client is free to impose its own caching parameters to the agent and the agent will utilize the "most recent" values sent to it. If this is a problem, a higher-level layer above BPFS can be implemented to coordinate the actions of the clients, perhaps scheduling them in a sequence such that only clients with similar caching requirements will run in parallel.

If the agent detects any error it will send back a "failure" reply consisting of a 24-byte header followed by the text of an error message in the form of a null-terminated C string. The format of the "failure" header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	SETCACHE was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

A "success" reply consists of a 24-byte header followed by a data part containing an array of 6 4-byte unsigned integer values that represent the current status of the agent's cache. The format of the "success" header is:

Field	Value	Meaning
total length	48	length of header and status data
op code	0	SETCACHE was successful
file id	≥ 0	same as in request
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

The 6 unsigned integer items in the data part following the header are as follows:

- the current activity state of the agent's cache: 0 means caching is off (in which case all following values in this array will be 0); 1 means caching is on.
- the current buffer replacement strategy in use: 1 means FIFO, 2 means LRU, 3 means RANDOM.
- the current write-through policy in use: 1 means write-through, 2 means write-behind.
- the current number of hash bins in the hash table.
- the maximum number of buffers that can be kept in the cache.
- the number of buffers currently in the cache.

6.5 The Protocol between Manager and Server

This protocol is the means by which a manager communicates with the data servers. There are two basic types of message exchange expressed in this protocol:

1. the manager forwards client requests to servers and gets back the servers' responses.

2. the manager sends a "confirm" or "abort" message to the server and gets back the servers' responses.

When the manager starts up, it must establish communication with a set of data servers. This is done by invoking the implementation-defined function "getdefaultservers", which returns the number of servers, a list of their locations, and a list of their names (to be used in metadata). This set becomes the "default" set of data servers that the manager will use when creating new files for which the client has not specified an explicit list of servers. There is a separate connection between the manager and each server. The manager may keep each connection open "forever", or may close and reopen connections as necessary.

6.5.1 Forwarded Operations

The nine possible operations sent from the client to the manager are the same nine operations in this protocol that the manager will forward to the servers. In most cases the manager makes only two changes to the message received from a client before forwarding it to each server.

1. The manager replaces the "identification" field in the header with its own unique identification, so that it can match the replies from the servers with this request.

2. The manager stores in the "block length" field of the header the ordinal number of this server in this file as determined by the list of servers in the metadata for this file. For example, if the metadata list for the requested file contains 4 servers A, B, C, and D, then the "block length" field in the header sent to server A will be 0, that in the header sent to B will be 1, that to C will be 2, and that to D will be 3.

The only other change the manager makes to a request received from a client before forwarding it to each server is that for the three open operations READBLOCK, WRITEBLOCK, and RDWRBLOCK, if the create attribute in the "block number" field in the request was 2, indicating "create as needed", then the manager will change this attribute to 0 if the metadata component already existed, and to 1 if it had to create the metadata component. This ensures that the creation of the data components by servers will be consistent with the creation of the metadata component by the manager.

Note that for the open and close operations, the messages sent by the manager to the servers contain the full variable-length data part sent to the manager by the client. If necessary to conform to implementation-defined naming conventions for the metadata and data components, the file name strings in this data part can be modified appropriately by the manager before forwarding to the client.

Upon receipt of a request for one of these nine operations forwarded from the client, the server will attempt to process it. If the server detects any error during this attempt, it will send a "failure" reply message back to the manager, otherwise it will send back a "success" reply.

The format of the "failure" reply is nearly identical to the format of a "failure" reply sent by the manager to a client. It consists of a 24-byte header followed by the text of an error message describing the failure. The general format of the "failure" reply header is:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	requested op was not successful
file id	0	reply to manager's request
id	id of request	to match reply with request
block number	0	unused
block length	≥ 0	ordinal number of this server in file
more blocks	0	unused

The data portion of a "failure" reply contains the text of an error message in the form of a null-terminated C string.

Likewise, the format of a "success" reply sent by a server back to a manager is similar to the "success" reply sent by the manager back to the client. For the four operations GETSTATUS, READBLOCK, WRITEBLOCK, and RDWRBLOCK, it consists of a 24-byte header followed by 24 bytes giving the status of the data component on this server. The general format of the "success" reply header is:

Field	Value	Meaning
total length	48	length of header and status info
op code	0	requested op was successful
file id	0	reply to manager's request
id	id of request	to match reply with request
block number	0	unused
block length	≥ 0	ordinal number of this server in file
more blocks	0	unused

The data part of the "success" reply contains the following 24 bytes of status information:

- 8 bytes — the location of the agent on this server.
- 2 bytes — the protection codes on this server.
- 2 bytes — an open file id on this server
- 4 bytes — the number of full blocks on this server.
- 4 bytes — the number of bytes in any partial last block.
- 4 bytes — currently unused

For the five operations CLOSEFILE, DELETEFILE, ERASEFILE, LINKFILE, and MOVEFILE, the "success" reply consists of just a 24-byte header in the following format:

Field	Value	Meaning
total length	24	length of header
op code	0	requested op was successful
file id	0	reply to manager's request
id	id of request	to match reply with request
block number	0	unused
block length	≥ 0	ordinal number of this server in file
more blocks	0	unused

6.5.2 Confirming or Aborting Server Actions

In order to ensure that all servers perform the same actions in a consistent manner, many operations require two exchanges between the manager and each server. The first exchange, described above, involves forwarding the client's request to each server and getting back a reply from each server. The second exchange, to be described next, involves sending to each server a CONFIRM request and getting back a reply to that, or sending to some of the servers an ABORT request, which requires no reply.

Between the time a server receives a request and the time it receives the relevant CONFIRM or ABORT, the file effected is blocked to all other operations except GETSTATUS.

If in the first exchange any server detects an error that prevents it from carrying out the requested operation, it will send a "failure" reply to the manager. This server is now finished with the operation and has made no change to any files or agents as a result of the failed operation. After all replies have been

received, the manager determines whether any of them were failures. If there was at least one failure, the manager may send an ABORT request to each of the other servers (if any) that sent back a "success" reply. It also sends a "failure" reply back to the client without waiting for a reply to the ABORT from the servers.

If all servers sent back "success" replies in the first exchange, the manager may send a CONFIRM request to each server. It then waits to receive a reply from each server before sending a "success" reply back to the client.

The use of the second exchange depends on the operation being performed and whether it was a success or failure. The following table illustrates the possibilities:

Operation	CONFIRM sent on success	ABORT sent on failure
CLOSEFILE	no	no
ERASEFILE	no	no
GETSTATUS	no	no
RDWRBLOCK	no	yes
READBLOCK	no	yes
WRITEBLOCK	if truncate is 1	yes
DELETEFILE	yes	yes
LINKFILE	yes	yes
MOVEFILE	yes	yes

The steps taken by the server in each of these cases is as follows:

- CLOSEFILE When the server receives the original CLOSEFILE request in the first exchange, it closes the file if at all possible. The only reason it might not be able to do so is that it did not have the file open in the first place (and so it is effectively already closed). It is assumed that if a server was able to open a file successfully, it must be able to close that file. Whether or not the close succeeds, this server should no longer have it open. Therefore, neither a CONFIRM nor an ABORT is necessary.

- ERASEFILE When the server receives the original ERASEFILE request in the first exchange, it deletes the file if at all possible. The only reason it might not be able to do so is that it did not have permission. Whether or not the erase succeeds, the server should have no access to it. By definition, ERASEFILE (unlike DELETEFILE) erases as much as possible without the need for everything to be erased. Therefore, the actions of one server do not need to be coordinated with the actions of any other server for this file, and neither a CONFIRM nor an ABORT is necessary.

- GETSTATUS When the server receives the original GETSTATUS request in the first exchange, it obtains the information about the file if at all possible. The only reason it might not be able to do so is that the file does not exist. This is a completely passive command as far as the state of the file is concerned, and therefore, whether or not the GETSTATUS succeeds, the actions of one server do not need to be coordinated with the actions of any other server for this file. Therefore, neither a CONFIRM nor an ABORT is necessary.

- RDWRBLOCK When the server receives the original RDWRBLOCK request in the first exchange, it opens the file for reading and writing if at all possible. If it succeeds, there is no need for a CONFIRM because there is nothing further for the server to do in this case — the file is already open. However, if some other server failed, then this server must be made to close the file, and therefore an ABORT is necessary.

- READBLOCK The situation is identical to that of RDWRBLOCK, except that the file is opened only for reading.

- WRITEBLOCK When the server receives the original WRITEBLOCK request in the first exchange, it opens the file for writing without truncation if at all possible. If it succeeds, and if the user requested truncation, there is a need for a CONFIRM because the truncation cannot be performed unless all the

servers have been able to open the file for writing successfully. If some other server failed, the manager will send an ABORT to each successful server so that it will close the file without truncation.

- DELETEFILE When the server receives the original DELETEFILE request in the first exchange, it must determine whether or not the file can be deleted without actually performing the delete. In UNIX, for example, this can be done by attempting to link the file to some other non-existant name in the same directory — if this succeeds, then the user has access rights which allow him to change the directory containing this file and a delete is possible.

  In its reply to the DELETEFILE, the server reports to the manager whether or not it can delete the file. If it cannot, this is a "failure" reply and the server is finished with this operation. If it can, this is a "success" reply and the server must away either a CONFIRM from the manager, indicating to go ahead and actually delete the file, or an ABORT from the manager, indicating to leave the file intact.

- LINKFILE When the server receives the original LINKFILE request in the first exchange, it creates the link to the new name if at all possible and reports back its success or failure to the manager. If any server fails, then the manager must send an ABORT to all those that succeeded so that they will destroy the newly created link. If all servers succeed, then the manager must send a CONFIRM to all of them so that they can unblock this file for other control operations.

- MOVEFILE When the server receives the original MOVEFILE request in the first exchange, it creates the link to the new name if at all possible, but does not destroy the old name. It then reports back its success or failure to the manager. If any server fails, then the manager must send an ABORT to all those that succeeded so that they will destroy the newly created link. If all servers succeed, then the manager must send a CONFIRM to all of them so that they can delete the old name.

Both the CONFIRM and ABORT requests sent by the manager consist of just a 24-byte header in the following format:

Field	Value	Meaning
total length	24	length of header
op code	> 0	CONFIRM or ABORT
file id	≥ 0	open file id
id	identification	to match reply with request
block number	0	unused
block length	≥ 0	ordinal number of this server in file
more blocks	0	unused

The "file id" field contains the open file identification code sent back to the manager by this server in its "success" reply to this operation. Its presence in the CONFIRM or ABORT request allows the server to identify which operation is being confirmed or aborted. Note that neither a CONFIRM nor an ABORT will ever be sent to a server that sent back a "failure" reply.

A CONFIRM always requires a reply, which consists of just a 24-byte header in the following format:

Field	Value	Meaning
total length	24	length of header
op code	0	requested op was successful
file id	CONFIRM	reply to manager's CONFIRM
id	id of request	to match reply with request
block number	0	unused
block length	≥ 0	ordinal number of this server in file
more blocks	0	unused

Note that this is a "success" reply — it is not possible for the server to send back a "failure" reply to a CONFIRM because it is expected that the operation will succeed (otherwise this server would not have replied "success" in the first exchange).

6.6 The Protocol between Server and Agent

The relationship between a server and its agents is different than the relationship between any other active components because these two components normally will reside on the same logical node and therefore do not have to communicate via a network. This is due to the fact that, in most implementations, the agent is expected to be a child process or thread spawned by the server. At the time it creates an agent, the server is able to communicate information to the new agent via shared memory. The server-agent protocol therefore only involves subsequent exchanges of information after the initial creation. The exact mechanism used for this exchange is implementation dependent, but is expressed in this report by using a message passing paradigm with the expectation that, in UNIX at least, pipes will be the chosen mechanism.

When the server first creates an agent, it assigns to that agent a unique 16-bit "open file identification" number. This number must be made known to the agent at the time of its creation, since the agent must include it in the header of all replies it sends to the server. This is called an "open file identification", but it could equally well be called an "active agent identification", because each file opened by the server is associated with a single agent, and all open requests for the same file are directed by the server to the same agent. This identification number is used by the server to associate the agent and messages from the agent with the internal information it keeps about the file.

In this protocol there are only two types of request that a server can send to an agent:

1. READBLOCK, WRITEBLOCK or RDWRBLOCK — to inform the agent that the server has successfully accepted from the manager a new open in one of these three modes.

2. ABORT — to inform the agent that the server has just received an ABORT request from the manager that aborts a previously accepted open.

After acting on the request, the agent is expected to reply to a READBLOCK, WRITEBLOCK or RDWRBLOCK, but not to an ABORT. The agent will also send "unsolicited" replies to the server whenever a client connects or disconnects with the agent. This enables the server to keep track of the number of clients connected to the agent and ultimately to decide when to terminate the agent process or thread. As explained below, with this protocol it is possible for a server to reuse an idle agent on a new file rather than to terminate an agent each time it becomes idle and then recreate it on each new file open.

6.6.1 READBLOCK, WRITEBLOCK, RDWRBLOCK

The server sends a READBLOCK, WRITEBLOCK, or RDWRBLOCK request to an existing agent whenever the server itself successfully processes one of these new file open requests from the manager. The server will delay sending its reply for this request back to the manager until it receives the corresponding reply from the agent. This is clearly necessary, since if the agent replies with a "failure", the server must report this back to the manager. Note that these open requests are not sent to a newly created agent, only to an existing agent that is expected to handle an additional request beyond the one that caused the server to create the agent in the first place.

The open request to the agent consists of a 24-byte header that may be followed by the name of the file. The header has the following format:

Field	Value	Meaning
total length	$24 + n$	length of header and data
op code	$\neq 0$	one of 3 open op codes
file id	≥ 0	open file id
id	identification	to match reply with request
block number	0	unused
block length	0	unused
more blocks	0	unused

The "file id" field contains the open file identification code generated by the server when the agent was created. It should always be the same in every request sent by the server to this particular agent. The server will also send this code in a reply back to the manager if the agent sends a "success" reply to this operation.

If the value of n in the "total length" field is zero, this request is reporting a new open of the same file in the same mode as the previous open and there is no additional data in the message following the header. However, if this value is non-zero, then it will have the value $n = (strlen(name) + 1)$, where *name* is the null-terminated C string containing the name of a file to be opened by the agent in place of the current file. (It could, of course, be the same name as the current file, but in that case the open mode will be different). This string follows the message header.

Upon receiving an open request that includes a file name, the agent will attempt to close the existing file and open the named file in the new mode. The agent will never have to create or truncate a file in such an open, because those tasks are always done by the server — by the time the open request arrives at the agent all the agent has to do is open the named file for reading, writing or updating. Note that a server will never ask an agent to close a file that is still in use. This mechanism exists so that the server can effectively "reuse" idle agent processes or threads without incurring the overhead of terminating and recreating them.

A READBLOCK, WRITEBLOCK or RDWRBLOCK request always requires a reply, which can be either a "success" or "failure" reply depending on whether or not the agent was able to process the open request successfully. The "success" reply consists of just a 24-byte header in the following format:

Field	Value	Meaning
total length	24	length of header
op code	0	requested op was successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	CONFIRM	reply to an open request
block length	0	unused
more blocks	0	unused

The "failure" reply consists of a 24-byte header followed by the text of an error message in the form of a null-terminated C string. The format of the "failure" reply header is as follows:

Field	Value	Meaning
total length	$24 + n + 1$	$n = strlen(error\ message)$
op code	$\neq 0$	requested op was not successful
file id	≥ 0	same as in request
id	id of request	to match reply with request
block number	CONFIRM	reply to an open request
block length	0	unused
more blocks	0	unused

6.6.2 ABORT

The server forwards an ABORT request to an agent whenever the server itself receives such a request from the manager after a successful file open. It means that some other server participating in this parallel file was not able to open its data component successfully, and therefore the previously successful open on this server and agent must be aborted. For the agent, this means that it should no longer expect a connection from a new client.

The ABORT request consists of just a 24-byte header in the following format:

Field	Value	Meaning
total length	24	length of header
op code	ABORT	abort a previous open
file id	≥ 0	open file id
id	0	no reply to this request
block number	0	unused
block length	0	unused
more blocks	0	unused

The "file id" field contains the open file identification code sent back to the manager by this server in its "success" reply to the operation being aborted. It should always be the same in every request sent by this server to this particular agent.

The server does not require or expect a reply from the agent after an ABORT request.

6.6.3 Unsolicited Replies from Agent to Server

Whenever a client opens or closes a network connection to an agent, the agent reports this to the server in charge of that agent by sending an "unsolicited reply" message to that server. The server is able to identify the source of these replies because the header must contain the unique "open file identification" number of the agent that was assigned by the server when it first created the agent. At the time a new connection is opened to a client, the agent assigns the client an arbitrary identification number that it will copy into the "block length" field in the header of all unsolicited replies it sends to the server.

The unsolicited replies an agent sends to the server consist of a 24-byte header followed by an 8-byte data part containing the location of the client as an implementation-defined opaque structure in network byte order.

The 24-byte header of these unsolicited messages has the following format:

Field	Value	Meaning
total length	32	length of header plus client location
op code	0	successful operation
file id	≥ 0	open file id
id	0	unsolicited reply
block number	$\neq 0$	RDWRBLOCK or CLOSEFILE
block length	≥ 0	client identification
more blocks	0	unused

The "block number" field will contain RDWRBLOCK to indicate a client connect, and CLOSEFILE to indicate a client disconnect.

7 Conclusions and Future Work

This report has described the architecture of BPFS, a basic parallel file system. The design is specified in terms of four protocols between four active components. It provides for the basic services needed to create, manage, read, write and delete parallel files stored on a set of disks distributed throughout a network. It is an "open" set of specifications to encourage experimentation with and evolution of the design.

BPFS itself is "low level", and is not designed for direct programming at the application level. Indeed, no programming interface is included in the specifications. It is expected that some features often found in parallel file systems, such as client side caching, parity redundancy for fault recovery, and collective I/O, will be implemented as separate layers on top of BPFS.

We have created an application programmer interface called "API0" that allows higher-level access to the functionality provided by BPFS. Several test programs have been written using this interface, and some performance measurements have been taken with these programs. In addition, a UNIX-like "Standard I/O" interface, and an MPI-IO interface (based on the ADIO concept [18]) have been written to go on top of API0. All these interfaces and their performance tests will be described in a forthcoming report.

The first implementation of BPFS, for UNIX using TCP/IP, is described briefly in the Appendix. Because this first implementation is on "generic UNIX", it does not use any of the advanced features now appearing in many UNIX implementations, such as threads, asynchronous I/O, and other POSIX real-time features. However an attempt was made to follow at least the spirit of the design specifications in providing an asynchronous "look and feel" to the active components and the API0 layer. This was done by interposing an extra software layer just above the UNIX operating system that simulates asynchronous network I/O. This too will be described in a separate report.

Finally, a group at ENS-Lyon is working on the more general parallel computation system of which BPFS is just one part. We plan to implement video streaming and video editing applications, as well as out-of-core array manipulation packages. We also plan to try other networking technologies which promise much higher network bandwidth and lower latency than is possible with TCP/IP. We would also like to try some of the newer disk technologies, such as network attached disks, to increase the performance of the file system.

References

[1] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serveless network file systems. In *5th Symposium on Operating Systems Principles, ACM Transactions on Computer Systems* (1995).

[2] Chen, Y., Winslett, M., Seamons, K. E., Kuo, S., Cho, Y., and Subramaniam, M. Scalable message passing in Panda. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems* (Philadelphia, May 1996), ACM Press, pp. 109–121.

[3] Choudhary, A., Bordawekar, R., More, S., Sivaram, K., and Thakur, R. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference* (June 1995).

[4] Cormen, T. H., and Colvin, A. ViC*: A preprocessor for virtual-memory C*. Tech. Rep. PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[5] Elford, C., Kuszmaul, C., Huber, J., and Madhyastha, T. Portable parallel file system detailed design. Tech. rep., University of Illinois at Urbana-Champaign, November 1993.

[6] Forum, M. P. I. MPI-2: Extensions to the message-passing interface. Tech. rep., University of Tennessee, Knoxville, Tennessee, July 1997.

[7] Foster, I., Kesselman, C., and Tuecke, S. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing 37*, 1 (August 1996), 70–82.

[8] Gibson, G. A., Stodolsky, D., Chang, P. W., Courtright II, W. V., Demetriou, C. G., Ginting, E., Holland, M., Ma, Q., Neal, L., Patterson, R. H., Su, J., Youssef, R., and Zelenka, J. The Scotch parallel storage systems. In *Proceedings of 40th IEEE Computer Society International Conference (COMPCON 95)* (San Francisco, Spring 1995), pp. 403–410.

[9] Huber, J., Elford, C. L., Reed, D. A., Chien, A. A., and Blumenthal, D. S. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing* (Barcelona, July 1995), ACM Press, pp. 385–394.

[10] Johnston, W. E., Herzog, H., Hoo, G., Jin, G., Lee, J., Chen, L. T., and Rotem, D. Distributed parallel data storage system: A scalable approach to high speed image servers. In *Proceedings of ACM Multimedia - San Francisco* (October 1994).

[11] Kotz, D., and Nieuwejaar, N. Flexibility and performance of parallel file systems. *ACM Operating Systems Review 30*, 2 (April 1996), 63–73.

[12] Miller, E. L., and Katz, R. H. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing 23*, 4 (June 1997), 419–446.

[13] MPI-2: Extensions to the message-passing interface. The MPI Forum, July 1997.

[14] Nieuwejaar, N., and Kotz, D. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing* (Philadelphia, PA, May 1996), ACM Press, pp. 374–381.

[15] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), ACM Press, pp. 79–95.

[16] Prylli, L., and Tourancheau, B. BIP: A new protocol designed for high performance networking on myrinet. In *Workshop PC-NOW, IPPS/SPDP* (1998).

[17] Russell, R. D., and Hatcher, P. J. Efficient kernel support for reliable communication. In *13th ACM Symposium on Applied Computing* (1998).

[18] Thakur, R., Gropp, W., and Lusk, E. An abstract-device interface for implementing portable parallel-I/O interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation* (October 1996), pp. 180–187.

Appendix

This appendix gives a brief overview of the first implementation of BPFS. All the software was written in ANSI Standard C for a basic "POSIX 1003.1 standard" UNIX extended with the BSD 4.3 socket interface to the TCP/IP protocol stack. This software was tested on the following platforms:

- Linux 2.0.29 on Intel Pentium processors interconnected by Myrinet and 10 Mbps ethernet.
- SunOS 5.5.1 on SUN SPARC and i86pc processors interconnected by 10 Mbps ethernet.
- OSF1 4.0 on DEC ALPHA processors interconnected by 10 Mbps ethernet.

In principle this software should be trivially portable onto any collection of UNIX-based workstations interconnected by a TCP/IP network.

Standard UNIX does not provide real-time or asynchronous I/O facilities. These have been simulated in this implementation using an extra software layer called the "q-interface" (described in a separate report). This layer provides a time resolution of 1 microsecond for data streaming operations. It also serves to isolate the use of TCP/IP networking from the rest of BPFS, so that to utilize a different protocol stack would require changing only this layer. Part of this isolation includes providing message transmission to the rest of BPFS by efficiently maintaining message boundaries in the TCP byte stream.

A.1 Network Locations

A network "location" is represented internally as a 32-bit binary IP address and a 16-bit binary port number, both in network byte order. Externally, a "location" is represented as an IP address in "dotted-decimal" notation, followed by a colon (:) followed by a port number as an integer, for example "140.77.11.38:3775". The colon and port number are optional when a "location" is provided by the user, in which case a "well know port number" of 3774 is assigned to manager components, and 3775 is assigned to server components.

A.2 Data and Metadata Storage

Both metadata and data components are stored as normal UNIX files in the host file system on each node. The following naming convention has been adopted. If the client wishes to call a file "xxx", then BPFS will store the metadata in a file called "xxx.meta" on the manager node, and each of the data components in a file called "xxx" on each server node. Before using a file name sent to it by a client, the manager explicitly looks for and strips off the suffix ".meta".

The BLOCKSIZE is 16384, although this value is simply a compile-time constant, so it can be easily changed by just modifying one header file and recompiling everything. This BLOCKSIZE allows BPFS to handle files containing up to 68 terabytes of data, assuming that much disk space is available and addressable on the server nodes.

UNIX stores a file as an ordered sequence of bytes. BPFS partitions this sequence into fixed-size subsequences called blocks, each containing BLOCKSIZE bytes. The byte position of the first byte in a block is determined by simply multiplying the block number by BLOCKSIZE. It is common for a UNIX implementation on a 32-bit machine to limit byte position numbers to 32 bits, which means a data component stored as a UNIX file can contain at most 4 Gigabytes (assuming there is enough disk space to store this much data on a single server node). Therefore, on such systems, BPFS with a BLOCKSIZE of 16384 is restricted to data components containing no more than 262,144 blocks.

A.3 Active Components

The client is implemented as a library whose interface is called "API0". This interface will be described in a separate report. User applications use BPFS by calling functions in this library.

The manager and server components are implemented as separate daemons running as user processes — they have not yet been implemented as "root" processes. They store relative file names (i.e., names starting

with a character other than "/") relative to the directories in which the manager and server daemons are started. It is therefore essential that the manager and all servers be started on their respective nodes in the same position in the directory hierarchy. Absolute file names (i.e., names starting with "/") are used directly as full absolute path names on each node. Because they are not "root" processes, the manager, server and agent components all access files with the user id and group id they were started with. If these ids are not the same as those of the client, the client may not be able to access his files outside of BPFS.

Each agent component is a child process "forked" (but not "execed") by a server. On the agent, caching is off by default. If a client turns caching on, the default "cache replacement strategy" is FIFO (1), the default "write-through policy" is WRITE_THRU (1), the default number of hash bins is 251, and the default maximum number of buffers allowed in the cache is 128.

A.4 Functions

The "getdefaultserverlist" function, which is called by the manager, obtains its list of servers by looking for the environment variable "BPFS_SERVERS" in the environment of the manager. The value of this environment variable must be a comma- or blank-separated list of DNS names and/or dotted-decimal IP addresses which the manager will use as its list of default servers. Optional port numbers can also be supplied as part of each item in this list using the "colon" notation. This function returns a pointer to this list. If this variable is not defined in the environment, this function will return a NULL pointer.

The "getdefaultstriping" function, which is called by the manager, obtains the number of the default mapping function by looking for the environment variable "BPFS_MAPPING" in the environment of the manager. If this variable is not defined in the environment, this function will return a value of 0 for it. This function also obtains the default thickness by looking for the environment variable "BPFS_THICKNESS" in the environment of the manager. If this variable is not defined in the environment, this function will return a value of 0 for it.

The "q_strtolocation" and "q_locationtostr" functions, which convert between the internal and external representations of a location and are used throughout BPFS, are provided by the q_interface layer. The formats were explained in Section A.1 above.

The default "getmanagerlocation" function, which is called by a client to locate the manager for a file, maps all file names onto a single manager process. It obtains the location of this manager by looking for the environment variable "BPFS_MANAGERS" in the environment of the client. The value of this variable must be the DNS name or dotted-decimal IP address of the node on which the manager daemon is running. An optional port number can also be supplied. If no such variable is defined in the environment, the client looks for the manager on the same node as the client.

The default "gatheruserinfo" function, which is called by a client, obtains the current "umask" of the client process and the login name of the client as a text string.

The two standard file striping functions (regular, pseudo-random) have been implemented. There is currently no mechanism to dynamically load new mapping functions. A user can provide his own functions, but they must be compiled and loaded with the client software. There are three tables, one for each of the three striping functions, defined in the source file for API0. The standard mapping functions occupy slots one and two in each table. To add new functions, simply add references to them in the initialization associated with each table.

The three standard cache replacement strategies (FIFO, LRU, RANDOM), and the two standard write-through policies have been implemented. There is currently no mechanism to dynamically load new replacement functions. A system administrator can provide his own functions, but they must be compiled and loaded with the server software. There are three tables, one for each of the three replacement functions, defined in the server source file. The standard replacement functions occupy slots one, two, and three in each table. To add new functions, simply add references to them in the initialization associated with each table.

A.5 Startup

In order to run BPFS, the server daemons must be started first, then the manager daemon, and then the user applications (which are linked to the API0 client interface). Once started, the server and manager

daemons run "forever", and must be manually terminated (by the use of Control-C, for example). All user applications utilize the same manager and server daemons.

When the server daemon is started, it can be provided with a command-line parameter giving the external representation of a location at which it is supposed to register itself as a server. Clearly this must be an interface (with optional port number) available on the node on which it is running. If no such parameter is provided, the server listens on "any" interface available on the node on which it is running.

When the manager daemon is started, it can be provided with one, two or three optional command-line parameters. (Obviously if the first parameter is missing, the second and third parameters must also be missing.)

1. The first command-line parameter is a comma- or blank-separated list of DNS names and/or dotted-decimal IP addresses which the manager will use as its list of default servers. Optional port numbers can also be supplied as part of each item in this list using the "colon" notation. If this parameter is missing, or if the list is empty, the manager will call the "getdefaultserverlist" function to obtain a list of default servers. If this function returns NULL or an empty list, the manager expects to find a single server on the same node as the manager to use as its default server.

2. The second optional command-line parameter is the number of the default mapping function. If this parameter is missing or has the value 0, the manager uses the value returned by the "getdefaultstriping" function. If this function returns a value of 0 for this variable, the manager uses a default file mapping function of 1, which performs regular striping with uniform thickness across the indicated set of servers.

3. The third optional command-line parameter is the default stripe thickness. If this parameter is missing or has the value 0, the manager uses the value returned by the "getdefaultstriping" function. If this function returns a value of 0 for this variable, the manager uses a default file striping thickness of 1.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399