December 13, 2002

To the Graduate School:

This thesis entitled "Design and Implementation of the System Interface for PVFS2" and written by Harish Ramachandran is presented to the Graduate School of Clemson University. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Computer Engineering.

Walter B. Ligon III, Advisor

We have reviewed this thesis
and recommend its acceptance:

Ron Sass

Harlan Russell

Accepted for the Graduate School:

Design and Implementation of the System Interface for PVFS2

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

by

Harish Ramachandran

December 2002

Advisor: Dr. Walter B. Ligon III

Abstract

As Linux clusters emerged as an alternative to traditional supercomputers one of the problems faced was the absence of a high-performance parallel file system comparable to the file systems on the commercial machines. The Parallel Virtual FileSystem(PVFS) developed at Clemson University has attempted to address this issue. PVFS is a parallel file system currently used in Parallel I/O research and as a parallel file system on Linux clusters running high-performance parallel applications.

An important component of parallel file systems is the file system interface which has different requirements compared to the normal UNIX interface particularly the I/O interface. A parallel I/O interface is required to provide support for non-contiguous access patterns, collective I/O, large file sizes in order to achieve good performance with parallel applications. As it supports significantly different functionality, the interface exposed by a parallel file system assumes importance. So, the file system needs to either directly provide a parallel I/O interface or at the least support for such an interface to be implemented on top.

The PVFS2 System Interface is the native file system interface for PVFS2 - the next generation of PVFS. The System Interface provides support for multiple interfaces such as a POSIX interface or a parallel I/O interface like MPI-IO to access PVFS2 while also allowing the benefits of abstraction by decoupling the System Interface from the actual file system implementation. This document discusses the design and implementation of the System Interface for PVFS2.

Dedication

To my family who have supported and always encouraged me to achieve my goals.

Acknowledgments

I would like to thank my advisor Dr.Walt Ligon for his guidance and support. I would also like to thank Phil Carns for his invaluable help.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Clusters

The availability of high-speed networks and increasingly powerful commodity processors at low prices have enabled the development of low-cost clusters. Clusters typically used in parallel processing are collections of independent computers connected by a network and dedicated to parallel processing. The cluster as a whole uses commercial-off-the-shelf(COTS) hardware and is managed as a single administrative entity thus easing system configuration. Since clusters themselves consist of off-the-shelf parts they have also been able to utilize the software and hardware developed for broad use. An upside to using broad based components is that the cluster can benefit from the advances in technology and price fluctuations of the components unlike supercomputers which often use custom-built components. All these advantages translate into a considerable reduction in the overall cost of building and maintaining a cluster.

The usage of commodity components in clusters has provided lots of flexibility for system configuration. This along with the emergence of open source software has helped researchers in high-performance computing experiment with various options.

While low-cost industrial standard hardware has been abundant, the software systems and tools for clusters have evolved more slowly. One such area where clusters have lacked a comparable option to supercomputers is parallel file systems.

1.2 Parallel File Systems

Scientific applications typically use multiprocessor computers to satisfy their computational needs. Many of them, however, also deal with large amounts of data such as data from satellites, checkpointing output, and visualization output. In addition, some applications may need to work with data too large to fit in main memory, needing virtual memory support. In all the above cases, the I/O system is the bottleneck due to the disparity between processor speed and disk speed. The UNIX derived file systems largely in use for parallel computing are unsuited for parallel, scientific workloads[12]. They fail to address the I/O bottleneck in parallel computing as they are not designed for concurrent data accesses by multiple processes. In the UNIX model, a file is considered as a linear, addressable sequence of bytes and read/write requests act on that sequence of bytes. As the entire file is located on a single disk, all accesses are serialized even though they don't involve the same bytes. In this case, file access is a bottleneck that affects the bandwidth of I/O operations. Hence, the application is unable to harness the processing power available in the cluster. Thus, parallel file systems have been developed that are able to support a scientific workload. A parallel file system scatters the blocks of each file across multiple disks(declusters), enabling parallel access to the file. This parallel access lessens the effect of the bottleneck due to the slow disk speed and larger bandwidth can be obtained for I/O operations. Some of the other features of such file systems are concurrency with guaranteed consistency, user controllable data distribution parameters, and multiple I/O interfaces.

Most of this development, however has been done in commercial file systems that are specific to the vendor's platform or are restricted to research prototypes. As clusters begin to replace supercomputers in scientific applications, there is lack of a high-performance production file system for clusters. The parallel virtual file system(PVFS) has filled the void nicely and serves to provide high-speed access to file data for parallel applications in a production environment. Features of PVFS are a consistent namespace across the cluster, control of file distribution by the user, and a transparent user space.

1.3 PVFS2 - The Next Generation

PVFS2, a collaboration between Clemson University and Argonne National Laboratory, is the next generation of the parallel virtual file system. PVFS2 is the result of a total redesign that has come about mainly due to changes in technology, both hardware and software, and also shortcomings in the previous design. PVFS2 seeks to address both technical and design issues in the new version so that it can maintain its goal of being a vehicle for parallel I/O research and a production quality file system for Linux based clusters. As of the time of this writing, PVFS2 is a work in progress and some of the ideas being discussed are still evolving and have not yet assumed their final form. Table 1.1 shows a comparison of several design issues between PVFS1 and PVFS2.

One of the important lessons from the previous system's software architecture was the need for separation of functionality. This has resulted in a modular design so that modules can be replaced as needed. Also, the implementations have been hidden behind abstractions and clear interfaces defined to access the individual components. Although a software engineering issue, the design framework is nevertheless important

Issue	PVFS1	PVFS2
Software Architecture	Monolithic	Layered Modular
I/O requests	Strided List-based	Completely general MPI Datatype based
Data distribution	Striping	Modular distribution code
Networking	TCP/IP	Abstract network layer(BMI)
Storage	UNIX files	Abstract storage layer(Trove)
Servers	Stateful Separate metaserver	Stateless Combined metaserver
Interfaces	POSIX-based with extensions Others layered	Low-level System Interface Replaceable user interfaces

Table 1.1: Comparison of PVFS1 and PVFS2

to ensure easy maintenance and ability to modify modules for research purposes, additional features, or performance improvements.

Parallel scientific programs often need to access non-contiguous regions of a file. Thus, we want the I/O description capability provided by PVFS2 to be as expressive as possible. The idea is to encapsulate as many access patterns as possible using the I/O description. This would allow it to support parallel I/O interfaces such as MPI-IO[5] and SIO[9] that require the underlying system to support non-contiguous accesses in a convenient manner. The older PVFS1 requests were limited as to what they could express.

In PVFS1, the default distribution is striped. A file striped on N I/O nodes has N files, one on each I/O node, containing the file's data. PVFS2 has a modular distribution mechanism that can support multiple distributions. In addition, PVFS2 allows new distributions to be added without changing the major file system components. Thus, it allows the distribution to be tuned to the file access pattern of an application.

The PVFS1 client library was designed to use TCP/IP for network communication. The network transfer layer in PVFS2 provides an abstract interface that allows the file system design to be independent of the networking mechanism used. The network transfer layer uses an abstraction called BMI[3] to handle both short and long messages with equal efficiency and to support a wide range of protocols and devices.

PVFS2 has a storage layer called Trove[17] that abstracts the storage of data. This allows the details of the storage mechanism to be hidden from the rest of the design. Thus, the servers can use raw disk, native file system, or relational databases for storage implementation.

In PVFS1, the servers keep up with the state of open files. This has resulted in a lot of complexity as the servers end up having to recover to a consistent state after a crash. In PVFS2, the servers are stateless. This will hopefully reduce the complexity and ease the design of multi-process servers.

In PVFS1, the client interface was intended to be a replacement for the standard POSIX interface. This interface lacks many features critical to use in parallel applications. Parallel I/O interfaces such as MPI-IO had to be implemented as a layer on top of the interface. The PVFS2 System Interface is designed at a lower level than a user interface. It has a full set of features for parallel computing, though it is very complex. Several different user interfaces can be neatly implemented on top of it including POSIX and MPI-IO. In addition, this interface is designed to interact with the Linux VFS to facilitate kernel implementations.

1.4 The System Interface

Figure 1.1 shows the overall system architecture for PVFS2. It is seen from the figure that the System Interface is the layer between the user level interface and the job or networking abstraction layer. This interface defines the logical operations supported by the underlying file system. There are two parts to the System Interface where it merges with the PVFS2 architecture. One is the top part of the interface where it meets the user interface or PVFSD. This represents the API defined by the

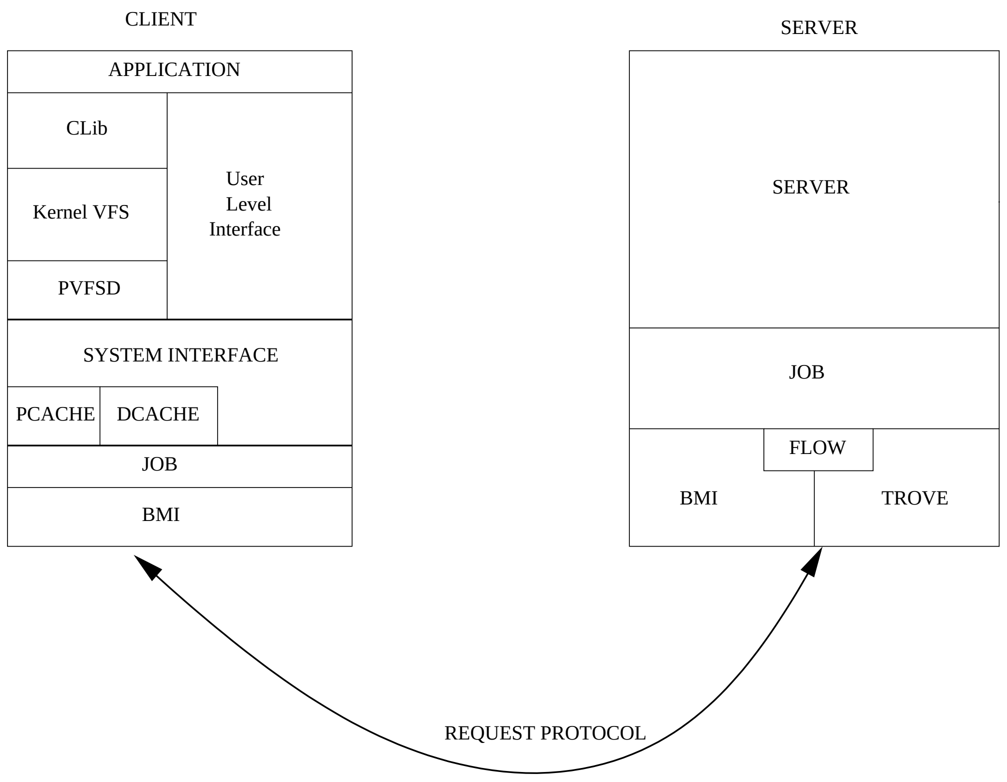

Figure 1.1: PVFS2 Architecture

System Interface that allows user-level applications to access the parallel file system. The lower part of the System Interface is responsible in dealing with the network abstraction layer(job layer). The design goals for the System Interface are outlined as follows.

- *Abstraction:* The file system needs to export a standard interface that can be used by the high-level application interfaces. This would also provide the necessary abstraction by decoupling the interface from the file system implementation details.

- *Flexibility:* To allow parallel file systems to maximize their performance, the file system interface needs to be as flexible as possible. This means that applications need to be able to exercise a measure of control on the file system parameters through the interface. This would enable applications to tune the file system policies to achieve the best fit for themselves.

- *Support for user-level interfaces:* This refers to the ability of the parallel file system interface to support multiple high-level interfaces. The high-level interfaces should support as much of the functionality as possible and the file system interface should only provide the necessary support for the high-level interfaces by defining primitives to enable non-contiguous access and read-write with high throughput.

- *Robustness:* Any application using the file system interface expects that errors during execution of operations are handled gracefully and the system interface shuts down cleanly. Error and if possible debugging information needs to be returned by the system interface to the application.

- *Performance:* The design of the file system interface plays a major part in the overall performance of the file system. The interface design needs to make use of optimizations wherever possible and also make efficient use of the underlying subsystems to achieve the best possible performance.

1.5 Approach

We propose the System Interface in PVFS2 as a solution to meet the demands made of a parallel file system interface. Our objective in proposing this interface is to provide a flexible and efficient interface that meets our stated goals. In this thesis, we present the design and implementation of the PVFS2 System Interface. After discussing the design goals in the previous section, we list the features of the PVFS2 file system interface which are as follows.

- file abstraction
- abstract distribution
- non-contiguous file access using complex I/O patterns

- stateless design
- control over distribution and caching
- extensible attributes
- extensible request format
- protection of file system data structures
- thread safety

In this section we discuss how the System Interface is designed to provide the features listed previously while meeting the design goals. File abstraction is provided by presenting the user library with a single logical file even though the actual file is declustered over multiple servers. The distribution is abstracted as a string identifier at the user library with the System Interface handling the details using the distribution module. The availability of a well-defined API provides file and distribution abstraction to the user library. The API implementation hides file-system specific details from the user while at the same time providing control through the API parameters. It is intended that changing the file system implementation does not heavily impact the API itself. The design of the well-defined API helps achieve the goal of abstraction along with providing the desired features.

System Interface features such as control over distribution parameters, support for complex I/O patterns, control over caching consistency, and extensible attributes add flexibility to the System Interface. Control over distribution parameters is needed to match the distribution to access patterns. This is necessary for the application to realize the performance benefits from declustering. The distribution parameters are dependent on the distribution used but usually include the file locations on disk and disk locations in the cluster. For a striped distribution they contain the count of I/O nodes across which the file is striped, the first I/O node, and the unit(in bytes) by which the file is divided among the various I/O nodes. These parameters can be set during the file creation. The System Interface supports an I/O request format that can encapsulate any MPI based derived datatype for non-contiguous I/O accesses which are a frequent occurrence in parallel applications. The cache consistency is controlled by using a timeout based scheme and caching can even be disabled by making the timeout zero. The ability to add new attributes is provided by the support for extended attributes in all attribute related functions. Other features that make the System Interface flexible are the stateless design and extensible requests. Extensible requests are made possible by the specification of the server request protocol which defines each System Interface operation as a collection of server requests and responses. Adding a new request in this scenario would entail just adding a few server requests. The definition of the interface as a low-level API providing primitives rather than semantics has led to a stateless design. Thread safety has been provided in the interface by removing dependence on global system variables and using locks to arbitrate access.

In addition to the features described above, the PVFS2 interface has been implemented to address the goals of robustness and performance. The implementation of the interface handles graceful recovery from error conditions and uses an error handling mechanism that aggregates the information from each subsystem to return error and debugging information to the application. The above features add to increased robustness. After the addition of features to the interface it is also necessary to add optimizations to ensure that good performance is achieved for operations those take the most time and also occur frequently. In our case, we have identified network requests to fetch metadata and the object handle in the above category. Two design features that speed them up are the pinode cache and PVFS directory entry cache. The pinode cache performs attribute caching thus preventing attribute requests over the network. The PVFS directory entry cache allows the file name to handle mapping to be cached. This saves lookup requests from going over the network.

In this section, we listed the major features to give an idea of the issues involved and what we have to implement. This outlines what we plan to do in this thesis. The remainder of this document is organized as follows. In Chapter 2, we present a review of parallel file systems and their file system interfaces that served as a background for the design of the system interface. Chapter 3 details the design details of the system interface and its related interfaces such as the PVFS dcache interface, pinode cache interface and configuration management interface. An outcome of the system interface design in PVFS2 is the server request protocol. The server request protocol is a standardization of the request exchange mechanism between the client(the system interface) and the servers. This chapter also contains a discussion of the server request protocol. Chapter 4 presents the evaluation of the System Interface and gauges its usefulness as the file system interface for PVFS2. Chapter 5 concludes this document by presenting the conclusions and suggesting the direction for future work.

Chapter 2

Background

2.1 Parallel File Systems

This section discusses parallel file systems and dedicated interfaces for parallel I/O that cover related work and served as background for the design of PVFS2 and the System Interface.

2.1.1 PIOUS

Goals

The basic goal of the Parallel Input/Output System(PIOUS) is to provide scalable bandwidth in a parallel environment using a framework built on the idea of declustering of data[13].

Design

The main components of PIOUS are a set of data servers, a service coordinator, and a library to link with the clients. The data server runs on each machine used for declustering and enables access to file data. Each server is independent and uses the local file system for data storage. The independence of the data servers allows asynchronous operation and increased parallelism. The service coordinator initiates major system operations but is not involved in general file access. The library provides an API for the clients to use the services of the parallel file system. PIOUS uses a transaction based model to guarantee consistency semantics.

File System Interface

The library in PIOUS is the interface provided to the client applications. This allows the file structure and selected logical views of a declustered file to be specified to the client.

2.1.2 Vesta

Goals

Vesta distinguishes itself as a file system that caters to shared file access by multiple processes. Thus, it is different from a traditional distributed file system that either offers weaker concurrency semantics or offers concurrency with costly synchronization mechanisms. The goal of Vesta is to provide shared file access to I/O intensive scientific applications while keeping performance in mind[7].

Design Overview

The Vesta parallel file system concentrates on 3 aspects to work towards its goals of providing high-performance - parallelism, scalability, and layering. Vesta provides parallelism by declustering of files. However, in contrast to its peer systems Vesta allows the user to view the parallel nature of the files. Compared to its peer systems that abstract the declustered nature from the users, Vesta allows its users to adjust the file declustering based on the I/O access patterns by allowing partitioning of file data among the various processes. Vesta uses a hashing scheme to locate files to minimize conflict on high-level directories. This is done to increase the scalability of the file system. The design also prevents serializing of operations for purposes of concurrency control and uses decentralized lookups during file accesses. This allows direct accesses to the I/O servers containing the metadata or data without any interactions between the nodes themselves. Vesta provides the abstractions necessary to create and manage the parallel views of the files.

The actual implementation details of Vesta are briefly described as follows. Vesta is implemented as 2 units - a client library linked to the application and a server that runs on the I/O nodes. Vesta uses a hash based scheme to locate the file instead of the name server or path traversal approach. The file objects are files, cells, and Xrefs(instead of directories). A hash on the filename locates the server containing the metadata. The metadata is distributed across all I/O servers and needs to be looked up prior to data access. Partitioning parameters are specified during a file open. This allows multiple processes to share a file without any synchronization provided the partitions are disjoint. Concurrency control is provided by a token-passing mechanism among the I/O nodes that guarantees atomicity across all nodes while providing sequential consistency among requests.

File System Interface

The Vesta client library also acts as the file system interface. The major idea in Vesta is the two-dimensional structure of files. The two dimensional file structure allows multiple processes to separate the file into non-overlapping segments using the facility of partitions provided in Vesta. This in turn allows the processes to extract maximum parallelism during file access. Also, disjoint partitions eliminate the costly synchronization mechanisms otherwise needed to arbitrate access.

2.1.3 PPFS 1

Goals

The principal goal of PPFS1 is to be a tool to explore and experiment with various system policies in parallel input/output systems. To successfully experiment with the policies the user needs to have broad control over file management policies. PPFS1 aims to provide a flexible API, the ability to explore policies such as caching, distribution, prefetching, and the ability to dynamically adapt the policies to match the access pattern[8].

Design Overview

PPFS1 is designed as a user library in order to avoid making frequent changes to the system software that are time-consuming and also because of the increased flexibility provided by working in user space. The basic software components in PPFS1 are the clients, servers, metadata server, and the caching agents. PPFS1 uses a client/server model. A PPFS1 client consists of the user application along with the local caching and prefetching software used to access the file system. The server actually resolves the requests made by the client and is made up of the server cache, prefetching engine, and the storage mechanism which handles data and metadata storage. The metadata servers service open and close requests for files in PPFS1. They maintain state for each open file. Clients can also directly exchange metadata without involving the metadata server. The caching agents are shared caches that serve multiple clients and all requests to the shared file pass through the caching agent rather than directly to the I/O servers. The caches ensure the coherence of the data provided to the clients. All the policies in use such as prefetching or caching can be controlled by the user and aid in the search for the best fit for each particular application.

File System Interface

The file system interface in PPFS1 is an API that tries to let the application advertise its information and also control the system policies that will ultimately affect I/O performance. The API contains functions to allow the application to specify its access patterns, control data distribution over servers, control caching policies at clients, servers, and control prefetching policies at clients, servers.

2.1.4 Galley

Goals

Galley attempts to make use of studies of parallel application workloads and performance evaluations of contemporary parallel file systems[14]. Galley's goals include the following

- give applications control over declustering
- handle various access patterns and sizes
- provide scalability
- transfer specific functionality to libraries implemented on top of Galley instead of making them part of the file system
- obtain good performance.

Design Overview

The Galley parallel file system consists of sets of clients and servers. Processors are dedicated either to computing or I/O. Clients run on the compute processors and the servers run on the I/O processors. A Galley client is an application linked to the Galley library and passes on file system requests after converting them to messages to the servers. Each client is independent of the others. Galley's I/O servers are composed of multiple units - compute threads, a cache manager, and the disk manager. A compute thread handles only requests from a specific compute processor and passes on the list of disk blocks needed to satisfy the request to the cache manager. The cache manager maintains a separate block list for each thread and also implements the cache replacement policy. For any block not in the cache, the cache manager makes a request to the disk manager. The disk manager services requests from the cache manager by relying on the underlying system to provide these services. Each Galley file has a 3-dimensional structure. Each file consists of subfiles which in turn consist of forks. Each subfile is placed on a disk and provides I/O parallelism. The number and placement of subfiles can be controlled by the application. A subfile contains one or more forks which are named, addressable, and linear sequences of bytes. Forks allow related information to be stored logically together but accessed separately.

File System Interface

Galley provides a specialized interface to the application. The interface provides for operations on files, operations on forks, standard data access primitives, and support for strided access patterns. Galley supports three structured strided requests and one unstructured request. These strided requests allow grouping of multiple requests to minimize network overhead and also allow for efficient disk scheduling decisions.

2.1.5 PPFS 2

Goals

Computational grids involve resources that are not always available and also applications with complex and varying demands. The motivation for PPFS2 is to create a adaptive control system that uses the current state of the system to adjust system policies specifically related to the I/O system. This is expected to tailor policies to application needs and hence yield maximum performance[15].

Design Overview

PPFS2 uses the Globus distributed system framework as the basic communication architecture among the various computational grids. Over Globus it uses the Autopilot real-time adaptive control system. The metadata manager and I/O servers perform their usual functions which is managing metadata and data respectively. Autopilot contains performance sensors to capture raw performance data, and decision procedures to choose a policy and set its parameters and actuators (to implement policy decisions). In addition to the quantitative data, the adaptive system uses a neural network to perform access pattern classification. The decision server uses both the file access pattern and the data to make a decision on choosing the striping and caching policies for the file. PPFS2 uses the relation between inter-requests intervals and striping as a factor in its policy selection. In addition to the adaptive mechanism, there is a feature for user steering of policies. Some other ideas being experimented with in PPFS2 are related to predicting access patterns and trends. This information can then be used to prefetch data.

File System Interface

PPFS2 uses the Scalable I/O initiative's low level API. Other high level libraries can be implemented on top of this interface. The SIO interface allows easy description of complex parallel I/O patterns and includes specification of hints regarding the access patterns.

2.1.6 PVFS

Goals

PVFS is designed as a production level parallel file system for Linux clusters. The design goals of PVFS are[4]

- high-speed concurrent read/write with multiple processes
- support for multiple APIs
- ability to run common UNIX commands like ls, cp
- ability to access the file system with utilities developed using the UNIX I/O API
- robustness
- ease of use

Design Overview

PVFS is a user space implementation. It is a client-server system with 2 types of servers. A metadata server that handles metadata operations involving permission checks, open, and close. The metadata server does not take part in I/O operations. The I/O server provides access to the data using the native file system for data storage. An I/O server runs on each I/O node. The I/O nodes have disks attached to them. Each file is distributed across the disks on I/O nodes. The applications interact through the client library.

File System Interface

PVFS supports multiple API's that include the native PVFS API, the UNIX/POSIX API, and MPI-IO. The other API's are built on top of the native PVFS API. The native API supports UNIX like contiguous read/write and uses a partitioned-file interface for simple strided access patterns. The partitioning allows a non-contiguous request to be made in a single call which would otherwise take multiple calls to the file system. Support for MPI-IO was provided so that the full range of MPI-IO's non-contiguous access patterns could be used. But, it has been since realized that the partitioned interface only supports a subset of access patterns possible through MPI-IO.

2.2 Specialized Interfaces

2.2.1 MPI-IO

MPI-IO is an API specified by the MPI forum to aid portable parallel programming. The need for MPI-IO is because a normal UNIX like API is not suitable for parallel I/O. So, MPI-IO provides functions those enable I/O parallelism, portability, and good performance. ROMIO is a portable implementation of MPI-IO. To ensure portability ROMIO uses an interface called the abstract device interface to separate the architecture dependent and independent parts. ADIO is just a collection of functions to enable parallel I/O. As long as ADIO is implemented for each file system, any parallel I/O API can be implemented on top of ADIO. Some of the features of MPI-IO are collective I/O, non-contiguous accesses, and non-blocking I/O. To provide better support for MPI-IO Thakur, et. al.[18] have proposed a list of features for file systems to provide. They include concurrent high- performance read/write, data consistency and atomicity semantics, an interface supporting non-contiguous accesses, large file support, and control over file striping.

2.2.2 SIO low-level API

This API[9] has been proposed by the Scalable I/O Initiative, a consortium of universities and companies. The main objective of this API is to define primitives so that the full potential of high-bandwidth network and storage devices can be realized. The document proposing the API states that the final goal after reaching a consensus on the interface is standardization of a parallel I/O interface. This interface chooses performance and parallelism over ease of use. Some of the features of this interface are application controls over cache consistency, application hints about access patterns, scatter-gather, collective I/O, and asynchronous operations. Ultimately, many of these ideas have been merged in the MPI-2 specification to create the MPI-IO interface.

2.3 Lessons from prior work

After looking at the work done in various parallel file systems and also at the various specialized interfaces for parallel I/O it is clear that there is no particular solution that deals with all parallel applications. Experience with parallel file systems has also made clear that there is no clear consensus on the file system interface or structure of parallel file systems[12]. So, the key is flexibility in configuration. It is required that an application make the choice of its interface and also be able to control the data distribution. This makes it essential that a native parallel file system interface only provide the primitives essential for parallel access with good performance and leave the functionality details to higher-level application libraries.

At the same time, our experience with PVFS1 has taught us that abstraction is necessary for flexibility so that changes can be made without disrupting the normal functioning of a production file system. The presence of a well-defined interface also makes a distinction between the functionality and implementation so that applications need not change their code if the implementation changes. We believe that the system interface in PVFS2 provides the right mix of abstraction and flexibility for parallel applications in a production environment.

Chapter 3

Design of the System Interface

3.1 Introduction

The System Interface is the low level client side interface that interacts with applications that want to access the Parallel Virtual File System. The System Interface is the native file system interface of PVFS2. This interface resembles the Linux VFS interface closely so as to enable PVFS2 to be supported as a file system under Linux. The System Interface defines the operations that can be requested of the parallel virtual file system. Most of the System Interface operations actually act on file system objects whereas the remaining few act on the file system as a whole such as retrieving statistics from the file system.

3.2 File system objects

PVFS2 has five file system objects - metafiles, datafiles, directories, symbolic links, and collections. The various file system objects except the symbolic links are shown in figure 3.1. As mentioned earlier the System Interface provides an API for the application to manipulate the system objects. The figure shows the file system objects

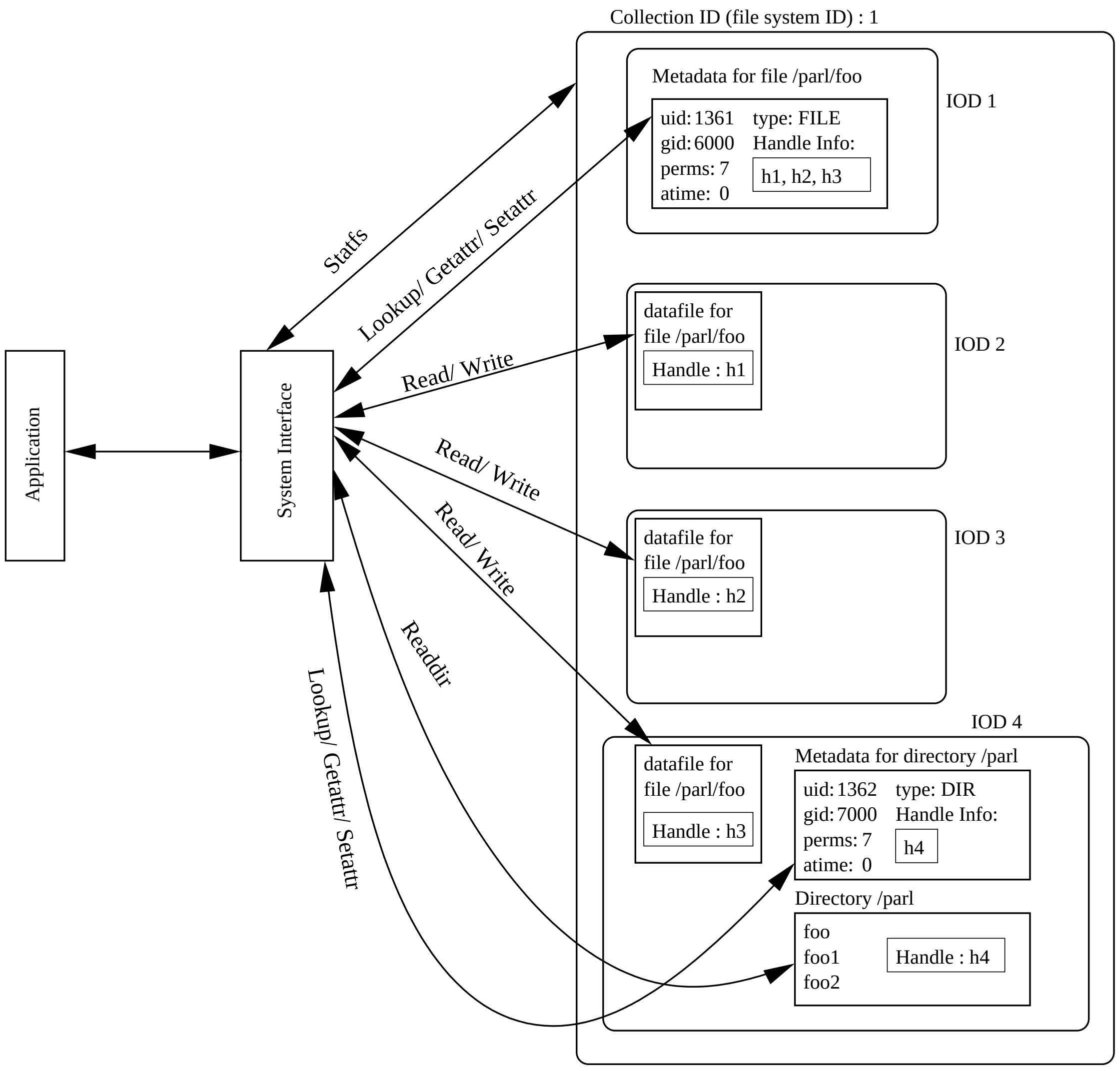

Figure 3.1: PVFS2 File System Objects

and System Interface functions to manipulate them. A detailed explanation of the file system objects follows.

3.2.1 Metafiles

Metadata can be described as properties of a group of data in a file system and allows the group of data to be treated collectively as a file. It typically contains information such as uid, gid, permissions, access time, creation time, modification time, type of file system object, and the distribution parameters. The System Interface allows metadata access or modification through functions getattr and setattr. The logical entities that store the metadata are referred to as the metafiles. Metafiles are implemented as Trove objects which in turn may be implemented as files if the Linux file system stores the metadata or data spaces if a database is used to store the metadata. The figure shows metafiles for a PVFS2 file and directory. Functions to access metafiles are also shown.

3.2.2 Datafiles

A datafile refers to the logical entity containing the actual data that makes up a file. The datafile could be a file in Linux or a data space in a database. A datafile in PVFS2 contains part of the data of the original file along with attributes necessary for acting on the datafile. Hence, a logical file in PVFS2 is declustered and may consist of several datafiles as shown in the figure. The way in which the file is declustered into datafiles is decided by the distribution used. The System Interface defines operations on files like read, write, create, and remove that may eventually end up as operations on individual datafiles.

3.2.3 Directories

A directory is a file that contains directory entries. The directory specific operations supported by the System Interface are creating a directory, removing a directory, and reading a directory. Besides these, operations such as lookup to get the directory handle and getattr/setattr to read/modify the attributes can also act on directories. A directory is not declustered and I/O is not allowed on directories.

3.2.4 Symbolic Links

Symbolic links are special files that are just pointers or shortcuts to other files. These files contain no data. Symbolic links are currently not supported.

3.2.5 Collection

A collection is an abstraction for a file system or group of file systems. A collection encompasses all the file system objects mentioned earlier. To differentiate the file system objects belonging to a particular collection each object is associated with a collection ID. The collection ID is a unique identifier for a collection. It is guaranteed that the handle space within a collection is unique. The System Interface provides a function to query the collection(file system) statistics called statfs.

3.3 Architecture

The overall architecture is shown in figure 3.2. The System Interface is designed as a set of functions that allow an application to interact with PVFS2. These functions only provide primitives to access the parallel file system. Further functionality needs to be layered on top of the System Interface by defining application libraries. Examples of such libraries could be a POSIX library or a parallel I/O library like MPI-IO. PVFS2 can also be incorporated as a file system under Linux as the System Interface tries to closely resemble the Linux VFS interface. Hence, all that would need to be done to support PVFS2 under Linux is to develop a layer under the kernel VFS interface that uses the functions of the System Interface for all operations on PVFS2 files. Apart from the System Interface API, there are other interfaces that the System Interface depends on. The pinode and dentry cache interfaces provide the PVFS2 interface with client side caching and minimize network traffic. The configuration management interface is used to access configuration parameters. This interface

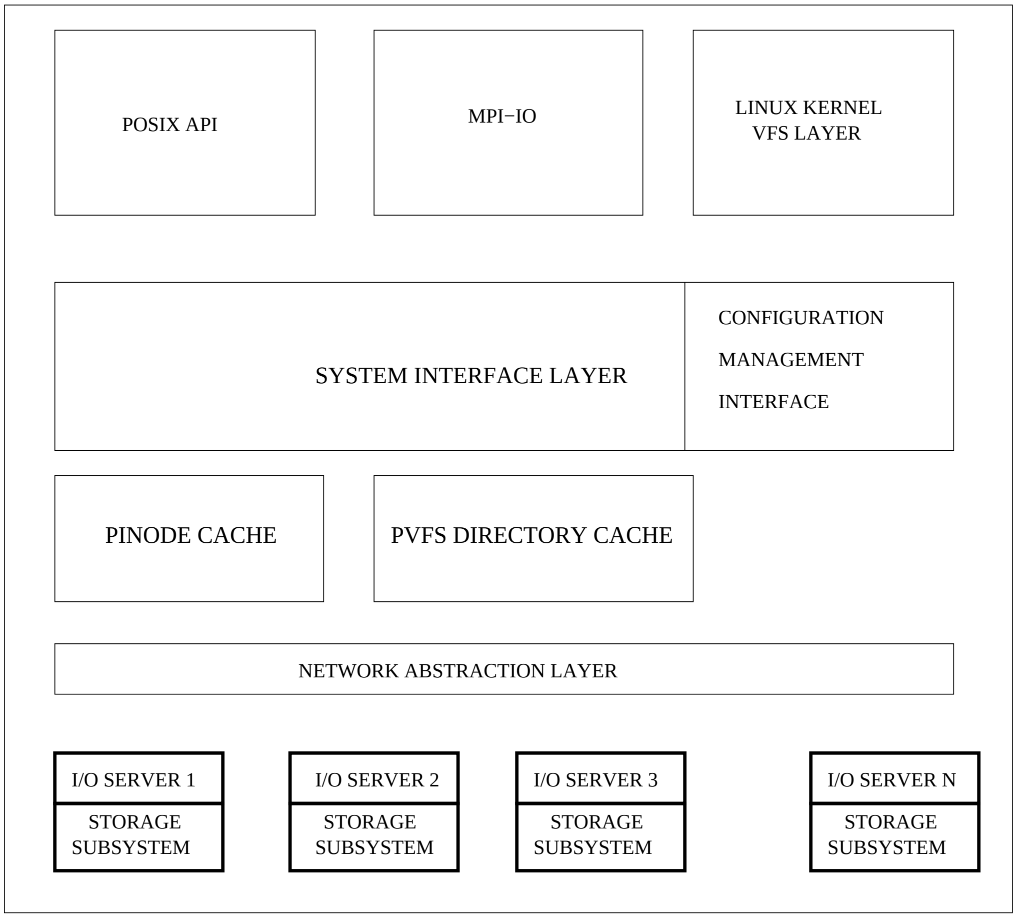

Figure 3.2: System Interface Architecture

principally consists of bucket table related functions that query and manipulate the mappings between buckets and servers.

The System Interface communicates with the file system servers via the request protocol. This protocol defines a set of request and response messages that operate on the file system objects define in 3.2. In turn, the interface provided to the user library consists of a set of request/ response pairs that operate on a logical file. Thus, the System Interface must interact with the file system abstractions and implement those for the user library.

3.4 System Interface Concepts

3.4.1 Handle

A handle in PVFS2 uniquely identifies a file system object within a collection. A handle and a collection identifier are required to identify the object across collections. The handle is visible at the System Interface layer but not to the user library. The System Interface manages the handles of the file system objects and provides the user library with a single abstract handle.

3.4.2 Pinode

A pinode in PVFS2 is equivalent to a Linux inode but its visibility is restricted to the System Interface. It is used as a mechanism to aggregate information about a PVFS2 file system object for a particular client. It can also be considered as linking a handle to its metadata. A mechanism is also in place in ensure the consistency of pinodes.

3.4.3 Pinode Reference

A pinode reference is an opaque type that acts as a unique identifier to a PVFS2 object across all file systems. All references to a PVFS2 object at the system interface level are either in terms of the object name or the pinode reference. A pinode reference is currently implemented as a combination of the metafile object handle and the collection id. The pinode reference is passed out of the System Interface to the application and the application uses it to refer to an object thereafter.

3.4.4 Bucket

A bucket can be thought of as a virtual disk. The basic idea is to associate file system objects with buckets rather than physical disks. This way objects can be decoupled from the actual storage details. This allows buckets to be moved from one physical disk to another or even be duplicated on multiple disks as the need arises without needing to change the metafiles of the contained objects. The relation between a logical file in PVFS2 and a bucket is as follows. A file in PVFS2 is declustered into buckets according to the specified distribution. The decision as to which I/O server the bucket is placed on is made separately.

3.4.5 Credentials

Credentials collectively refer to the permission and owner information for a PVFS object. The idea is to use this to verify permissions for access.

3.5 System Interface Function Specification

The System Interface API can be organized into 5 groups.

- Interface management operations
- Object creation, query and destruction operations
- I/O operations
- Object locking operations
- File system query operations

3.5.1 Interface management operations

The parameters to most of the System Interface API functions contain a request and response structure. A few functions though, have only a request structure and have no response. The request structures contain the inputs to the request and the response structures contain the data returned after the request is serviced by the server. We show the fields of the request and response structures for the System Interface functions below.

- **PVFS_sys_initialize(pvfs_mntlist mntent_list)**

  The parameters for PVFS_sys_initialize are shown below. The pvfs_mntlist structure contains a count of the number of mount entry structures and a pointer to the mount entry structures.

  The fields of the pvfs_mntlist structure are

  - int nr_entry // number of entries in pvfstab
  - pvfs_mntent *ptab_p // pointer to entries in pvfstab
  - gen_mutex_t *mt_lock // mutex lock

  The fields of the pvfs_mntent structure are

  - PVFS_string meta_addr // metaserver address
  - PVFS_string serv_mnt_dir // root mount point
  - PVFS_string local_mnt_dir // local mount point
  - PVFS_string fs_type // file system type
  - PVFS_string opt1 // options
  - PVFS_string opt2 // options

  PVFS_sys_initialize initializes the system interface data structures. Its parameter is a structure containing configuration information either from a pvfstab file or the mount command line. PVFS_sys_initialize needs to be called before calling any other system interface function. It initializes the BMI and flow messaging interfaces. It also makes a GETCONFIG request to the server to obtain configuration information and set up the received information to be accessed by the configuration management interface. This function is also responsible for initializing and setting up the pinode and directory entry caches.

- **PVFS_sys_finalize(void)**

  PVFS_sys_finalize shuts down the System Interface. This function needs to be called after all system interface operations are finished. It deallocates memory referenced by the system interface. It also closes down all other interfaces such as the BMI interface, flow messaging interface, pinode cache interface, and directory cache interface.

3.5.2 Object creation, query and destruction operations

- **PVFS_sys_lookup(PVFS_sysreq_lookup *req, PVFS_sysresp_lookup *resp)**

  The parameters for PVFS_sys_lookup are shown below.

  The fields of the request structure are

  - PVFS_string name // object name
  - PVFS_fs_id fs_id // file system id
  - PVFS_credentials credentials // uid, gid, permissions

  The fields of the response structure are

  - pinode_reference pinode_refn // handle, file system id

  PVFS_sys_lookup returns the pinode reference for a file, directory or symlink given the object name and file system id. It is the equivalent of the namei function in Linux that translates a file name to an inode number. Lookup employs path traversal to obtain the pinode reference while also doing permission checks for the entire path traversed.

- **PVFS_sys_getattr(PVFS_sysreq_getattr *req, PVFS_sysresp_getattr *resp)**

  The parameters for PVFS_sys_getattr are shown below.

  The fields of the request structure are

  - pinode_reference pinode_refn // handle, file system id
  - PVFS_bitfield attrmask // attributes to be fetched
  - PVFS_credentials credentials // uid, gid, permissions

  The fields of the response structure are

  - PVFS_object_attr attr; // attributes fetched
  - PVFS_attr_extended extended // extended attributes

  The fields of the attr structure in the response are

  - PVFS_uid owner
  - PVFS_gid group
  - PVFS_permissions
  - PVFS_time atime // access time
  - PVFS_time mtime // modification time
  - PVFS_time ctime // creation time
  - int objtype // type of file system object
  - The fields below are part of a union
  - PVFS_metafile_attr meta // metafile specific attributes

  - PVFS_datafile_attr data // datafile specific attributes
  - PVFS_directory_attr dir // directory specific attributes
  - PVFS_symlink_attr sym // symlink specific attributes

  PVFS_sys_getattr obtains the properties of the file, directory or symlink identified by the pinode reference passed as input. There is an option to obtain attributes other than the generic information such as owner, permission information, creation, access, and modification times by specifying attribute masks. Attribute masks enable getting attributes such as size or object specific information such as distribution and data file handles.

- **PVFS_sys_setattr(PVFS_sysreq_setattr *req)**

  The parameters for PVFS_sys_setattr are shown below.

  The fields of the request structure are

  - pinode_reference pinode_refn // handle, file system id
  - PVFS_object_attr attr // new attributes
  - PVFS_bitfield attrmask // attributes to be modified
  - PVFS_credentials credentials // uid, gid, permissions
  - PVFS_attr_extended extended // extended attributes

  PVFS_sys_setattr allows the manipulation of the properties of a file, directory, or symlink specified by the pinode reference input. As in PVFS_sys_getattr, an attribute mask may be used to narrow the attributes to be modified.

- **PVFS_sys_mkdir(PVFS_sysreq_mkdir *req, PVFS_sysresp_mkdir *resp)**

  The parameters for PVFS_sys_mkdir are shown below.

  The fields of the request structure are

  - PVFS_string entry_name // directory entry name
  - pinode_reference parent_refn // handle, fs id of parent directory
  - PVFS_object_attr attr // attributes of new entry
  - PVFS_bitfield attrmask // attribute mask
  - PVFS_credentials credentials // uid, gid, permissions

  The fields of the response structure are

  - pinode_reference pinode_refn // handle, file system id

  PVFS_sys_mkdir creates a directory with given attributes and obtains a pinode reference to the created directory. An entry for the newly created directory is added to the parent directory.

- **PVFS_sys_rmdir(PVFS_sysreq_rmdir *req)**

  The parameters for PVFS_sys_rmdir are shown below.

  The fields of the request structure are

  - PVFS_string entry_name // directory entry to be removed
  - pinode_reference parent_refn // handle, fs id of parent directory
  - PVFS_credentials credentials // uid, gid, permissions

  PVFS_sys_rmdir removes the directory indicated by the object name, parent directory, and file system id. A directory can be removed only if it contains no objects. The entry for the removed directory is deleted from the parent directory.

- **PVFS_sys_create(PVFS_sysreq_create *req, PVFS_sysresp_create *resp)**

  The parameters for PVFS_sys_create are shown below.

The fields of the request structure are

- PVFS_handle entry_name // name of file to create
- pinode_reference parent_refn // handle, fs id of parent directory
- PVFS_object_attr attr // attributes of new object
- PVFS_bitfield attrmask // attribute mask
- PVFS_credentials credentials // uid, gid, permissions

The fields of the response structure are

- pinode_reference pinode_refn //handle, file system id

PVFS_sys_create creates a new file with specified attributes and obtains a pinode reference to it. This involves creating both the metadata and also creating the datafiles on the various I/O servers.

- **PVFS_sys_remove(PVFS_sysreq_remove *req)**

  The parameters for PVFS_sys_remove are shown below.

  The fields of the request structure are

  - PVFS_string entry_name // name of file to remove
  - pinode_reference parent_refn // handle, fs id of parent directory
  - PVFS_credentials credentials // uid, gid, permissions

  PVFS_sys_remove removes the file specified by the object name, parent directory and file system id passed as input. This involves removing all the datafiles from the I/O servers, removal of the metafile, and deleting the directory entry from the parent.

- **PVFS_sys_rename(PVFS_sysreq_rename *req)**

  The parameters for PVFS_sys_rename are shown below.

The fields of the request structure are

- PVFS_string old_entry // old name of entry
- pinode_reference old_parent_reference // old entry's directory
- PVFS_string new_entry // new name of entry
- pinode_reference new_parent_reference // new entry's directory
- PVFS_fs_id fs_id // file system id
- PVFS_credentials credentials // uid, gid, permissions

PVFS_sys_rename renames an existing file or directory given the old and new object names along with the old and new parent pinode references and the file system id.

- **PVFS_sys_symlink(PVFS_sysreq_symlink *req, PVFS_sysresp_symlink *resp)**

  The parameters for PVFS_sys_symlink are shown below.

  The fields of the request structure are

  - PVFS_string name // name of link
  - PVFS_fs_id fs_id // file system id
  - PVFS_string target // name of file link points to
  - PVFS_object_attr attr // attributes of link
  - PVFS_bitfield attrmask // attribute mask
  - PVFS_credentials credentials // uid, gid, permissions

  The fields of the response structure are

  - pinode_reference pinode_refn

PVFS_sys_symlink creates a symbolic link to a file or directory.

- **PVFS_sys_readlink(PVFS_sysreq_readlink *req, PVFS_sysresp_readlink *resp)**

  The parameters for PVFS_sys_readlink are shown below.

  The fields of the request structure are

  - pinode_reference pinode_refn
  - PVFS_credentials credentials // uid, gid, permissions

  The fields of the response structure are

  - PVFS_string target

  PVFS_sys_readlink reads out the contents of a symbolic link.

3.5.3 I/O operations

- **PVFS_sys_read(PVFS_sysreq_read *req, PVFS_sysresp_read *resp)**

  PVFS_sys_read reads data from a file given the pinode reference and the I/O request pattern.

- **PVFS_sys_write(PVFS_sysreq_write *req, PVFS_sysresp_write *resp)**

  PVFS_sys_write writes data to a file given the pinode reference and the I/O request pattern.

- **PVFS_sys_allocate(PVFS_sysreq_allocate *req, PVFS_sys resp_allocate *resp)**

  PVFS_sys_allocate is not yet implemented. The function allocates specified size of data for file on the I/O servers indicated by the pinode reference passed in as input.

- **PVFS_sys_duplicate(PVFS_sysreq_duplicate *req, PVFS_sysresp_duplicate *resp)**

  The parameters for PVFS_sys_duplicate are shown below.

  - pinode_reference old_reference // entry to duplicate
  - PVFS_string new_entry // name of new entry
  - pinode_reference new_parent_reference // new directory

  The fields of the response structure are

  - pinode_reference pinode_refn // new handle, file system id

  PVFS_sys_duplicate is not yet implemented. The function creates a new file with name as specified in input and with the same distribution and attributes as file indicated by the pinode reference passed in as input.

3.5.4 Object locking operations

- **PVFS_sys_lock(PVFS_sysreq_lock *req, PVFS_sysresp_lock *resp)**

  PVFS_sys_lock is not yet implemented. The function obtains a lock on the file specified by the pinode reference passed in as input.

- **PVFS_sys_unlock(PVFS_sysreq_unlock *req, PVFS_sys resp_unlock *resp)**

  PVFS_sys_lock is not yet implemented. The function removes the lock on the file specified by the pinode reference passed in as input.

3.5.5 File system query operations

- **PVFS_sys_statfs(PVFS_sysreq_statfs *req, PVFS_sysresp _statfs *resp)**

  The parameters for PVFS_sys_statfs are shown below.

The fields of the request structure are

- PVFS_fs_id fs_id // file system id
- PVFS_credentials credentials // uid, gid, permissions

The fields of the response structure are

- PVFS_statfs statfs // file system statistics

The fields of the PVFS_statfs structure are

- PVFS_meta_stat mstat // metaserver statistics
- PVFS_io_stat iostat // I/O server statistics

The fields of the PVFS_meta_stat structure are

- PVFS_count32 filetotal // total number of metafiles

The fields of the PVFS_io_stat structure are

- PVFS_size blksize // file system block size
- PVFS_count32 blkfree // number of free blocks
- PVFS_count64 blktotal // total number of blocks available
- PVFS_count32 filetotal// maximum number of files
- PVFS_count32 filefree // number of free files

PVFS_sys_statfs obtains the statistics for a file system specified by the file system id passed in as input. The information obtained regarding the file system is organized as meta server info and I/O server info.

- **PVFS_sys_readdir(PVFS_sysreq_readdir *req, PVFS_sys resp_readdir *resp)**

  The parameters for PVFS_sys_readdir are shown below.

The fields of the request structure are

- pinode_reference pinode_refn // directory to read entries from
- PVFS_token token // token passed in
- PVFS_count32 pvfs_dirent_incount // number of entries to read
- PVFS_credentials credentials // uid, gid, permissions

The fields of the response structure are

- PVFS_token //token returned
- PVFS_count32 pvfs_dirent_outcount // number of entries returned
- PVFS_dirent *dirent_array // entries returned

PVFS_sys_readdir reads specified number of directory entries from directory indicated by the pinode reference passed in as input. This function can be called repeatedly on a directory with the token returned each time passed in as input the next time. The number of directories returned is specified separately in case the number of directory entries requested is greater than the actual number of entries present.

- **PVFS_sys_fhdump(PVFS_sysreq_fhdump *req, PVFS_sys resp_fhdump)**
  PVFS_sys_fhdump is not yet implemented.

3.6 System Interface Implementation

The usefulness of the System Interface lies in the functionality afforded by the interface. The role of the implementation is to make sure that the functionality is implemented in a way so as to achieve good performance. The interface affords us an abstraction by which we can modify the implementation to reflect our understanding of the underlying issues involved in improving PVFS2. At the same time, the applications need not change to benefit from the changes made for the better. Ultimately, we expect that any changes made will only improve on the performance and production level standards of the overall file system. Other issues we need to consider in the implementation are thread safety and robustness. It is possible that multiple clients may simultaneously use the system interface. So, the system interface must implement thread safety by making all global data structures it uses thread safe. The robustness of a production level file system depends on the robustness of its components. As the file system is accessed through the system interface there must be a provision in the interface to handle errors gracefully and return debugging info to the user application. The system interface attempts to provide robust error handling and debugging ability to the application by handling operations so that an error does not leave the system in an inconsistent state. This does not mean that the system is fault tolerant.

The system interface functions implement their functionality by constructing a request, sending the request to the server and processing the response from the server. This process is repeated as many times as it is needed. The request exchange mechanism between the client(in this case, the system interface) and the servers is standardized in the form of the server request protocol. This also ensures stateless working of the server. The communication mechanism from the client to the server is abstracted by the network transfer layer and hence is transparent to the client. The client just hands off the requests to the network layer and collects responses from it. This frees up the client from a lot of complexity.

To minimize network traffic due to client requests and responses the system interface utilizes the pinode cache. It is the responsibility of the client to ensure that the data obtained from the cache is correct. To prevent traversing the path each time to obtain the corresponding pinode reference the client uses the PVFS dcache. So, in short the client first looks up information in the above caches and only if necessary contacts the server. The configuration management interface is used by the client to associate the files to their location both while creating or performing any other operations on them.

Full path permission checking is handled only during lookup but object level permissions are verified during each operation by passing on the credentials information to the server. Lookup also uses an optimization to lookup multiple path segments in a single server request.

3.7 Server Request Protocol

The server request protocol standardizes the client/server request exchange mechanism. The protocol is a combination of the stateless server design and the design of the system interface in PVFS2. The client/server exchanges usually consist of predefined server request or response structures. In most cases, the exchange starts off with a request by the client specified using a server request structure. The I/O transfer which is in raw bytes is an exception to the above described method of passing requests and responses. It is to be noted that the request or response structures in the protocol do not exactly parallel those in the system interface as some objects are not visible at the system interface level and vice versa.
The server request and response structures are shown below. The union in the structures depends on the particular request or response being sent.

```
struct PVFS_server_req_s {
  PVFS_server_op op;
  PVFS_size rsize;
  PVFS_credentials credentials;
  union {
    PVFS_servreq_lookup_path lookup_path;
    ...
  } u;
};
struct PVFS_server_resp_s {
  PVFS_server_op op;
  PVFS_size rsize;
  PVFS_error status;
  union {
    PVFS_servresp_lookup_path lookup_path;
    ...
  } u;
};
```

We now illustrate the methodology to construct a server request and response for lookup_path and the way in the server request and response are laid out contiguously for transfer over the network.

As shown in figure 3.3, the actual request parameters for lookup_path are preceded by the generic parameters listing the request id, the size of the entire request, and the credentials information. As the pathname is a string, a contiguous buffer is allocated that totals the sizes of the server request structure and the path name string. The server request parameters are first filled in and at the end of that the pathname string is copied in. In the case of the response the handle array and the attribute array are variable length quantities. The contiguous buffer for the response is allocated taking into account the maximum amount of data that could be returned. This would be in the event of the handle and attribute information for all the path segments being returned. The networking layer uses the allocated buffer to fill in the response returned from the server. In a similar way, the other requests and responses can be constructed.

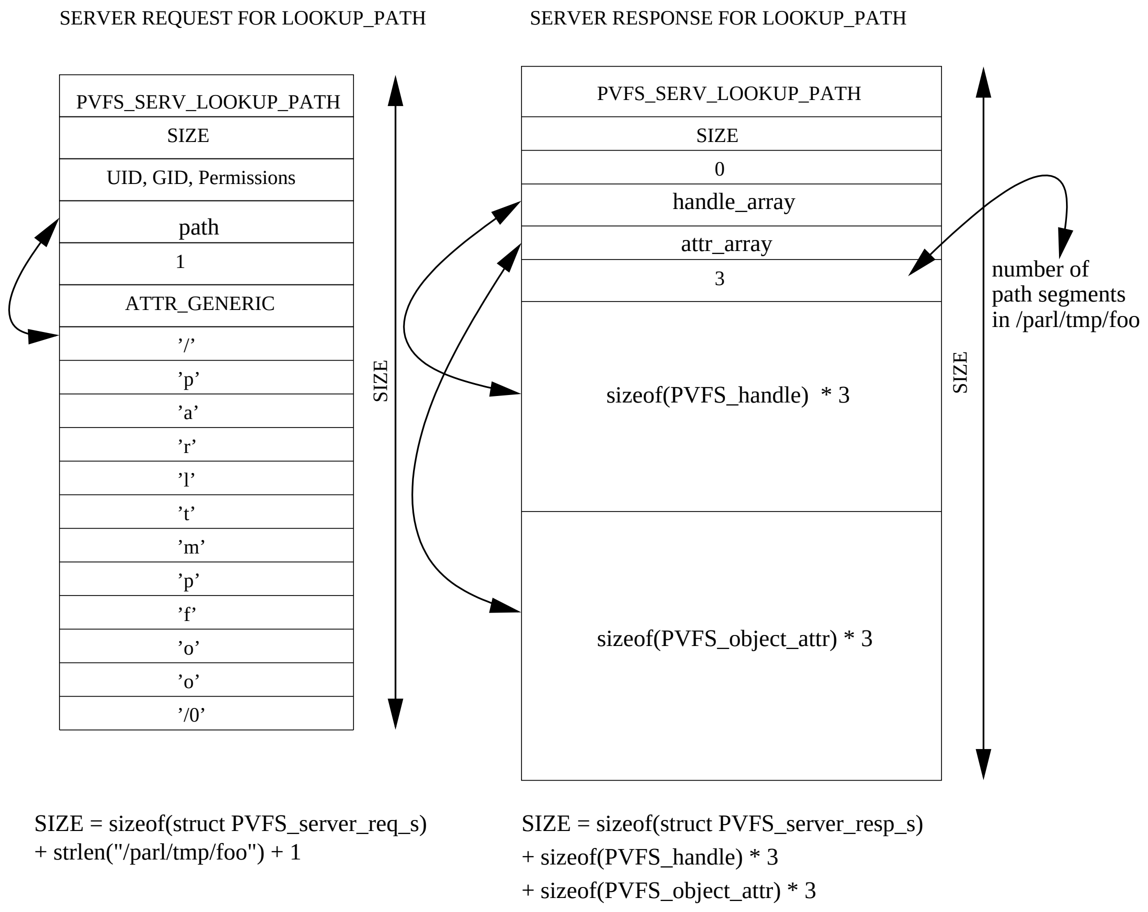
SERVER REQUEST FOR LOOKUP_PATH
SERVER RESPONSE FOR LOOKUP_PATH
PVFS_SERV_LOOKUP_PATH
SIZE
UID, GID, Permissions
path
1
ATTR_GENERIC
'/'
'p'
'a'
'r'
'l'
't'
'm'
'p'
'f'
'o'
'o'
'/0'
SIZE
PVFS_SERV_LOOKUP_PATH
SIZE
0
handle_array
attr_array
3
sizeof(PVFS_handle) * 3
sizeof(PVFS_object_attr) * 3
SIZE
number of path segments in /parl/tmp/foo
SIZE = sizeof(struct PVFS_server_req_s) + strlen("/parl/tmp/foo") + 1
SIZE = sizeof(struct PVFS_server_resp_s) + sizeof(PVFS_handle) * 3 + sizeof(PVFS_object_attr) * 3

Figure 3.3: Server request and response for lookup_path

The server request protocol specification follows. We show only those fields specific to each request.

3.7.1 Lookup Path

Type	Name	Description
PVFS_handle	starting_handle	Handle of starting directory in path
PVFS_string	path	Full path to be traversed
PVFS_fs_id	fs_id	File system identifier
PVFS_bitfield	attrmask	Mask to specify desired attributes

Table 3.1: Lookup Path Request

Type	Name	Description
PVFS_handle*	handle_array	Ordered array of handles per segment traversed
PVFS_object_attr*	attr_array	Array of object attributes
PVFS_count32	count	Count of number of handles returned

Table 3.2: Lookup Path Response

3.7.2 Get Attributes

Type	Name	Description
PVFS_handle	handle	Handle of object to fetch attributes for
PVFS_fs_id	fs_id	File system identifier
PVFS_bitfield	attrmask	Mask to specify desired attributes

Table 3.3: Get Attributes Request

Type	Name	Description
PVFS_object_attr	attr	Attributes of the object specified in request
PVFS_attr_extended	extended	Extended attributes

Table 3.4: Get Attributes Response

3.7.3 Set Attributes

Type	Name	Description
PVFS_handle	handle	Handle of object to set attributes for
PVFS_fs_id	fs_id	File system identifier
PVFS_object_attr	attr	Attribute values to be set
PVFS_bitfield	attrmask	Mask to specify desired attributes
PVFS_attr_extended	extended	Extended attributes

Table 3.5: Set Attributes Request

No response for setattr

3.7.4 Get Configuration

Type	Name	Description
PVFS_string	fs_name	Name of file system to get config info for
PVFS_count32	max_strsize	Max string size allowed for response mappings

Table 3.6: Get Configuration Request

Type	Name	Description
PVFS_fs_id	fs_id	File system identifier
PVFS_handle	root_handle	Root handle for the file system
PVFS_count32	meta_server_count	Number of metaservers in system
PVFS_string	meta_server_mapping	Ordered list of metaservers
PVFS_count32	io_server_count	Number of I/O servers in system
PVFS_string	io_server_mapping	Ordered list of I/O servers

Table 3.7: Get Configuration Response

3.7.5 Make Directory

Type	Name	Description
PVFS_handle	bucket	Bucket to associate object with
PVFS_handle	handle_mask	Number of bucket bits in handle
PVFS_fs_id	fs_id	File system identifier
PVFS_object_attr	attr	Attribute values of new object
PVFS_bitfield	attrmask	Mask to restrict attributes to be set

Table 3.8: Make Directory Request

Type	Name	Description
PVFS_handle	handle	handle of new directory created

Table 3.9: Make Directory Response

3.7.6 Remove Directory

Type	Name	Description
PVFS_string	entry_name	Name of directory to remove
PVFS_handle	parent_handle	Handle of parent directory
PVFS_fs_id	fs_id	File system identifier

Table 3.10: Remove Directory Request

3.7.7 Create Directory Entry

Type	Name	Description
PVFS_string	name	Name of directory entry to create
PVFS_handle	new_handle	Handle of object
PVFS_handle	parent_handle	Handle of directory to add entry to
PVFS_fs_id	fs_id	File system identifier

Table 3.11: Create Directory Entry Request

3.7.8 Remove Directory Entry

Type	Name	Description
PVFS_string	entry	Name of directory entry to remove
PVFS_handle	parent_handle	Handle of directory to remove entry from
PVFS_fs_id	fs_id	File system identifier

Table 3.12: Remove Directory Entry Request

3.7.9 Create

Type	Name	Description
PVFS_handle	bucket	Bucket to associate PVFS object with
PVFS_handle	handle_mask	Number of bucket bits in handle
PVFS_fs_id	fs_id	File system identifier
int	type	Type of PVFS object

Table 3.13: Create Request

Type	Name	Description
PVFS_handle	handle	Handle of file created

Table 3.14: Create Response

3.7.10 Remove

Type	Name	Description
PVFS_handle	handle	Handle of PVFS file to remove
PVFS_fs_id	fs_id	File system identifier

Table 3.15: Remove Request

No Response for remove

3.7.11 File System Statistics

Type	Name	Description
int	server_type	Metaserver or I/O server
PVFS_fs_id	fs_id	File system identifier

Table 3.16: Statfs Request

Type	Name	Description
PVFS_serv_statfs	stat	File System Statistics

Table 3.17: Statfs Response

Type	Name	Description
PVFS_mserv_stat	mstat	Meta server statistics
PVFS_ioserv_stat	iostat	I/O server statistics

Table 3.18: Contents of PVFS_serv_statfs

Type	Name	Description
PVFS_count32	filetotal	Total number of files

Table 3.19: Contents of PVFS_mserv_stat

Type	Name	Description
PVFS_size	blksize	File system block size
PVFS_count64	blkfree	Number of free blocks
PVFS_count64	blktotal	Total number of blocks available
PVFS_count32	filetotal	Maximum number of files
PVFS_count32	filefree	Number of free files

Table 3.20: Contents of PVFS_ioserv_stat

3.7.12 Readdir

Type	Name	Description
PVFS_handle	handle	Handle of directory to read entries from
PVFS_fs_id	fs_id	File system identifier
PVFS_token	token	Current position in directory
PVFS_count32	pvfs_dirent_count	Number of entries to read

Table 3.21: Readdir Request

Type	Name	Description
PVFS_token	token	Updated token reflecting current position
PVFS_count32	pvfs_dirent_count	Number of entries actually read
PVFS_dirent*	pvfs_dirent_array	Array of entries read

Table 3.22: Readdir Response

3.8 Related Interfaces

In this section we talk of the various interfaces the System Interface depends on in its implementation. These interfaces provide functionality for caching of pinodes, caching of directory entries, storing of configuration parameters, and mapping of buckets to servers. The System Interface is the only layer that makes use of them and the application cannot directly access these APIs.

3.8.1 Pinode cache

The role of the pinode cache is to serve as a shorter path to the metadata for the client. Instead of making a server request for the metadata each time and in turn incurring network overhead for the request, the client first looks in the pinode cache. If the entry is found in the cache, it is tested for validity. Fetching the pinode is handled by the pinode helper functions layer which provides a pinode fetch and validate mechanism. The validate is done using timestamps instead of a cache-coherence protocol. This layer refreshes the pinode by getting the attributes and filling in the pinode. The appropriate timestamp is also updated. The pinode cache implementation is thread safe. Currently, we have implemented a simple stack based cache. The various pinode cache operations supported are as follows.

- **pcache_initialize(pcache *cache)**

  pcache_initialize initializes the pinode cache interface and also sets up the cache data structures.

- **pcache_finalize(pcache *cache)**

  pcache_finalize shuts down the pinode cache interface.

- **pcache_lookup(pcache *cache, pinode_reference refn, pinode *pinode_ptr)**

  pcache_lookup searches for a specified pinode in the cache and returns the pinode if found.

- **pcache_insert(pcache *cache, pinode *pnode)**

  pcache_insert adds/merges a pinode to the cache. The merge operation merges 2 pinodes based on the timestamps of their contents.

- **pcache_remove(pcache *cache, pinode_reference refn, pinode **item)**

  pcache_remove removes a specified pinode from the cache and returns the removed item.

3.8.2 PVFS Directory Entry cache

The purpose for the PVFS Directory Entry cache is to prevent lookup operations on files traversing the network each time. The idea is to cache already resolved file names so that the next time the resolution of the name to the pinode happens in the cache itself. The cache implementation for the PVFS dcache is currently quite similar to the pinode cache. The dcache operations supported are as follows.

- **dcache_initialize(struct dcache *cache)**

  dcache_initialize initializes the dcache interface and also sets up the cache data structures.

- **dcache_finalize(struct dcache *cache)** dcache_finalize shuts down the dcache interface.

- **dcache_lookup(struct dcache *cache, char *name, pinode_reference parent, pinode_reference entry)**

  dcache_lookup searches for a specified directory entry in the cache and returns the entry if found.

- **dcache_insert(struct dcache *cache, char *name, pinode_reference entry, pinode_reference parent)**

  dcache_insert adds an entry to the cache if not already present. If the entry is already present, just updates its timestamp and returns successfully.

- **dcache_remove(struct dcache *cache, char *name, pinode_reference parent, unsigned char *item_found)**

  dcache_remove removes a specified entry from the cache.

3.8.3 Configuration Management Interface

The Configuration Management Interface exports functions to the system interface to manage all server related configuration information mainly obtained through the Getconfig server request. Most of the interface is now dedicated to handling the bucket to server mapping and vice versa. We know from the definition of a bucket that the datafile or metafile of a file system object is associated to a bucket and not the actual server. This bucket identifier is embedded in the object handle. So, to determine the bucket and in turn the server that holds the metafile or datafile for an object, it is required that given a handle we be able to find the bucket identifier. This is the reason that the configuration management interface provides functions to determine the server given a bucket identifier. It is expected that other functions not directly related to buckets would be added later on and listed under the configuration management interface. The interface exports the following functions.

- **config_bt_initialize(pvfs_mntlist mntlist_list)**

  config_bt_initialize initializes the interface related data structures.

- **config_bt_finalize(void)**

  config_bt_finalize shuts down the interface by deallocating the interface related data structures.

- **config_bt_get_next_meta_bucket(PVFS_fs_id fsid, PVFS_handle *bucket, PVFS_handle *handle_mask)**

  config_bt_get_next_meta_bucket takes a file system identifier as input and returns the bucket, handle mask, and the metaserver to use while creating a new PVFS system object.

- **config_bt_get_next_io_bucket_array(PVFS_fs_id fsid, int num_servers, char **io_name_array, PVFS_handle **bucket_array, PVFS_handle *handle_mask)**

  config_bt_get_next_io_bucket_array takes a file system identifier and the number of servers as input and returns the requested number of buckets, handle masks, and I/O servers needed to create datafiles for a PVFS file.

- **config_bt_map_bucket_to_server(char **server_name, PVFS_handle bucket, PVFS_fs_id fsid)** config_bt_map_bucket_to_server takes a bucket and file system identifier as input and returns the server name which is associated with the bucket.

- **config_bt_map_server_to_bucket_array(char **server_name, PVFS_handle **bucket_array, PVFS_handle *handle_mask)**

  config_bt_map_server_to_bucket_array takes a server name as input and returns the buckets and their handle masks associated with the server.

- **config_bt_get_num_meta(PVFS_fs_id fsid, int *num_meta)**

  config_bt_get_num_meta takes the file system identifier as input and returns the metaservers in the file system.

- **config_bt_get_num_io(PVFS_fs_id fsid, char **io_server_array)**

  config_bt_get_num_io takes the file system identifier as input and returns the ioservers in the file system.

- **config_fsi_get_root_handle(PVFS_fs_id fsid, PVFS_handle *fh_root)**

  config_bt_get_root_handle takes the file system identifier as input and returns the root handle for the file system.

- **config_fsi_get_io_server(PVFS_fs_id fsid, char **io_server_array, int *num_io)**

  config_fsi_get_io_server takes the file system identifier as input and returns the I/O servers for the file system.

- **config_fsi_get_meta_server(PVFS_fs_id fsid, char **meta_server_array, int *num_meta)**

  config_fsi_get_meta_server takes the file system identifier as input and returns the meta servers for the file system.

- **config_fsi_get_fsid(PVFS_fs_id fsid, char *mnt_dir)**

  config_fsi_get_fsid takes the mount directory of a file system as input and returns the file system identifier for the file system.

3.9 Summary

We have presented the design of the System Interface in this chapter. Initially, we discussed the overall design and how the individual file system objects fit into the design. We then introduced terms those were frequently used and relevant to the discussion. Subsequently, we presented the functions in the actual interface grouped by functionality followed by details of the implementation covering various issues such as name resolution, caching, and permissions. After the implementation, we moved on to an overview of the request protocol used in the exchanges between the System Interface and the servers. Finally, we mentioned the various related interfaces used for client side, name resolution caching, and configuration management.

Chapter 4

Evaluation

The System Interface was primarily designed to serve as a file system interface for PVFS2 and at the same time allow us to further our research on parallel I/O. The objective of this research was twofold.

- To utilize the experience gained from the implementation of PVFS1 in the implementation of a more powerful file system interface. This would help address deficiencies in the earlier file system interface as well as provide newer features that would allow PVFS2 to be used effectively.

- To evaluate the new design and determine the degree of success we have achieved in our goals

4.1 System Interface Implementation

As described in the System Interface Specification[ch.3] 14 out of the 19 functions in the System Interface have been implemented. In addition, the pinode and PVFS directory caches have also been implemented along with the configuration management interface. The 19 functions are part of the System Interface API specification, whereas the other parts like the pinode and PVFS directory caches are underlying modules used by the API. Please refer chapter 3 for details on any of the above interfaces. This makes up the first implementation of the System Interface.

As full-scale testing could not be done, the functions have been evaluated with a test harness that simulates the working of the job interface(the networking layer) and also the server. This testing only covers correctness and usability. No profiling or performance specific testing is planned until a working prototype of a full-fledged PVFS2 system is ready. The role of the testing using the harness was to verify that the API exposed the right primitives to the application using it and that the amount of complexity involved in the calling code was manageable. The complexity of the application code would help us determine if the API provided the correct level of abstraction to the higher layers.

The role of the test harness is to test if the System Interface works with the job interfaces and the server request protocol. This involves checking arguments that are passed to the job layer and subsequently to the server part of the harness. After the testing, it is expected that when the System Interface links to the actual job layer it would be able to send requests and receive responses correctly. In the implementation this is done by first checking the parameters to the job layer in turn handed over to the server portion of the harness. The server portion of the harness validates the parameters passed by the System Interface and then using a simple implementation returns the requested information though the response structures of the Server Request Protocol. The returned information is then interpreted by the System Interface to decide the next action. The action could consist of either further processing or simply passing on the response to the client invoking the API. Thus, the testing is also a validation of the Server Request Protocol.

4.2 Evaluation of the System Interface

In order to evaluate the System Interface we compare it to the file system interface in PVFS1 which is a combination of a file system interface and a POSIX file I/O library. As various stages of the System Interface are still in progress we are unable to obtain any actual performance results so we proceed to provide qualitative arguments. Our approach will consist of giving a case study of how a particular operation is handled in both PVFS1 and PVFS2, and how the presence of a particular feature in the PVFS2 file system interface enables the operation to be executed more efficiently. Finally, our intent is to illustrate that the System Interface is a significant improvement over the PVFS1 interface.

Firstly, we mention the significant distinguishing features of the PVFS2 file system interface from that of PVFS1.

- abstraction
- support for multiple interfaces
- flexibility and modularity
- description of complex I/O patterns
- clearly defined semantics
- thread safety
- PVFS directory cache
- pinode cache

Next, we provide comparative arguments to show that each of the above features indeed leads to the improvement of the PVFS2 file system interface over that of PVFS1.

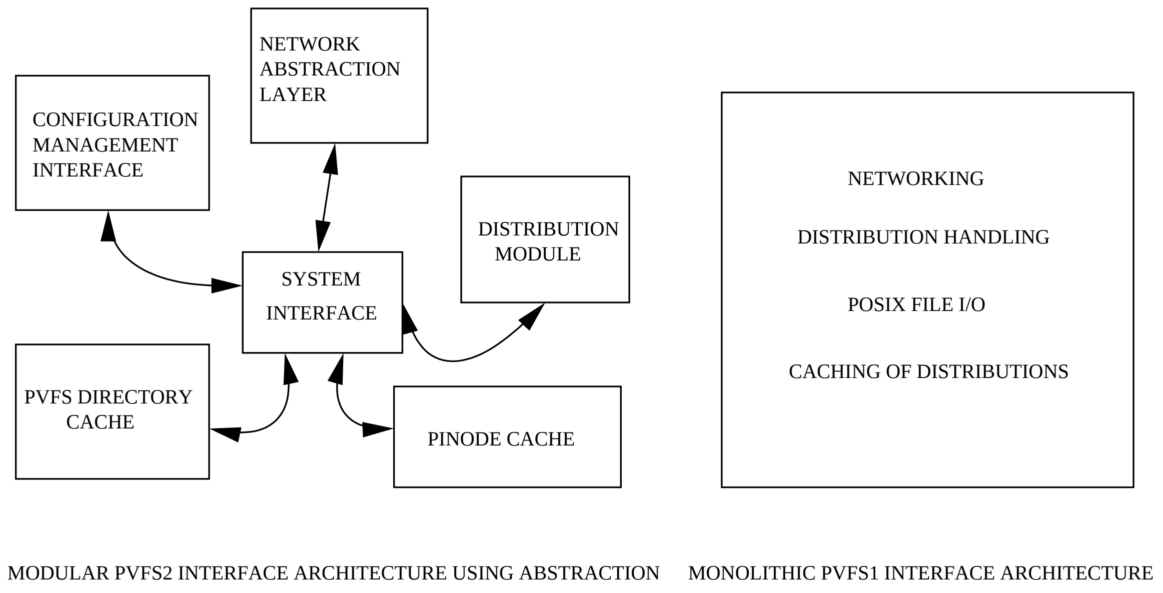
NETWORK ABSTRACTION LAYER
CONFIGURATION MANAGEMENT INTERFACE
DISTRIBUTION MODULE
SYSTEM INTERFACE
PVFS DIRECTORY CACHE
PINODE CACHE
NETWORKING
DISTRIBUTION HANDLING
POSIX FILE I/O
CACHING OF DISTRIBUTIONS
MODULAR PVFS2 INTERFACE ARCHITECTURE USING ABSTRACTION
MONOLITHIC PVFS1 INTERFACE ARCHITECTURE

Figure 4.1: Modular System Interface vs. monolithic PVFS1 interface

4.2.1 Abstraction(Data hiding)

The PVFS1 interface is a combination of a file I/O library and a file system interface. Due to this, each function in the library needs to be directly involved with details such as keeping up with sockets and storing the distribution information for PVFS files. This introduces unnecessary complexity in the library code. Also, PVFS1 uses many system specific structures and hence it is tied down to the operating system used. The specification of a System Interface provides the necessary abstraction so that higher level libraries do not have to be concerned about dealing with the file system data structures. The System Interface in itself abstracts the file system to the libraries and the libraries can be restricted to providing only the semantics in their functionality. The System Interface takes on the responsibility of dealing with the communication subsystem and providing features such as caching. This leads to cleaner library code and the abstraction insulates the library code from changes in the file system implementation.

4.2.2 Support for multiple interfaces

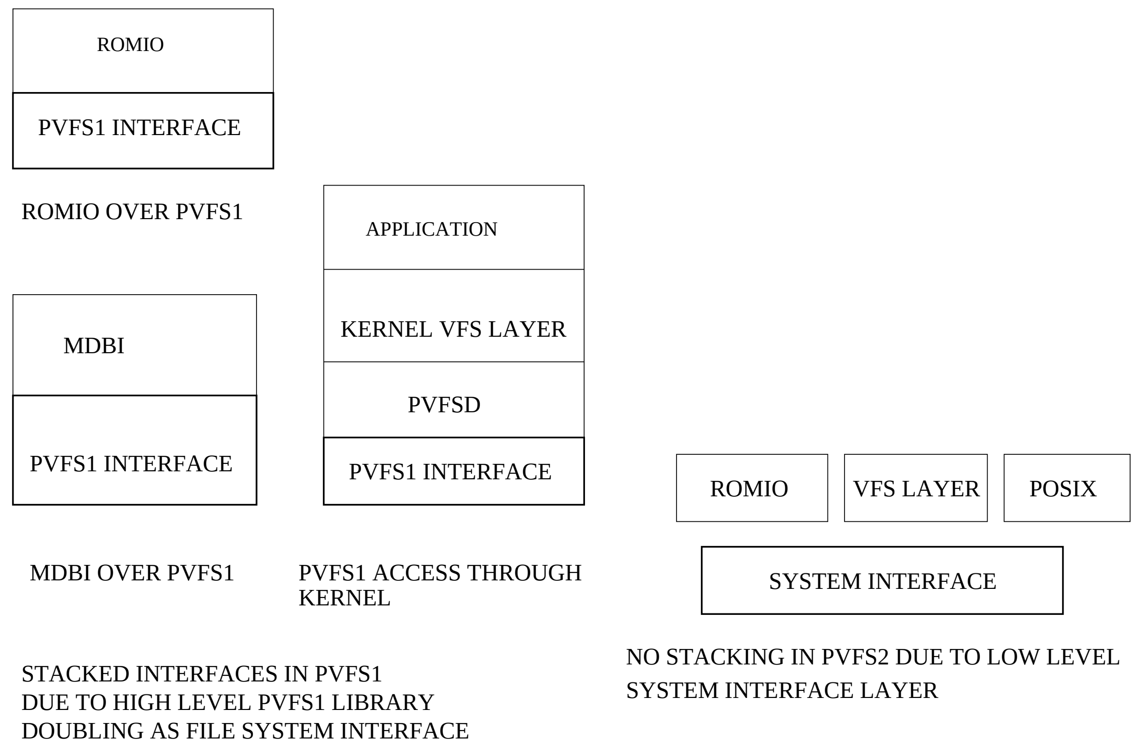
ROMIO
PVFS1 INTERFACE
ROMIO OVER PVFS1
APPLICATION
KERNEL VFS LAYER
PVFSD
PVFS1 INTERFACE
MDBI
PVFS1 INTERFACE
MDBI OVER PVFS1
PVFS1 ACCESS THROUGH KERNEL
ROMIO
VFS LAYER
POSIX
SYSTEM INTERFACE
STACKED INTERFACES IN PVFS1 DUE TO HIGH LEVEL PVFS1 LIBRARY DOUBLING AS FILE SYSTEM INTERFACE
NO STACKING IN PVFS2 DUE TO LOW LEVEL SYSTEM INTERFACE LAYER

Figure 4.2: Stacking of interfaces in PVFS1

As most parallel applications will use high-level libraries rather than directly use the file system interface, the effort involved in implementing libraries on top of the basic parallel file system interface is indeed an important issue to consider. The PVFS1 interface as mentioned earlier is a higher level interface in that it combines the features of a POSIX file I/O library and hence includes a lot more functionality than a file system interface. This makes the implementation difficult as it limits the facilities that the file system can expose to the library. In PVFS1 interfaces end up being stacked over one another as each interface is atop both the parallel file system interface and a POSIX library. In the System Interface, the limitations of PVFS1 have been avoided by making the file system interface provide low-level functions that can be used by higher level libraries. The libraries are given the job of implementing any functionality such as a POSIX layer or an MPI-IO layer. This allows the System

Interface to support a much larger number of interfaces than the previous version in PVFS1 and at the same time does not lead to stacked interfaces.

4.2.3 Flexibility and Modularity

The file system interface in PVFS1 is not modular in its design and there is no clear separation of functionality. This results in the interface implementation itself handling file I/O, distributions, and networking. The application does not have the ability to set options for consistency, choose the distribution, or adjust the data layout. The System Interface along with the rest of the PVFS2 subsystems encapsulates functionality in modules and uses interfaces to communicate with modules. The various System Interface components like the pinode cache and PVFS dcache define clean interfaces for the System Interface functions to use. The advantage that modularity provides is the ability to replace one module with another seamlessly. In addition the System Interface is quite flexible in providing the application options to tune consistency, control the distribution, and support to suggest data layout. These allow the application to make better use of the file system by tuning the policies to its benefit.

4.2.4 Description of complex I/O patterns

Non-contiguous file access is a frequent requirement of parallel applications so the file system interface has to support an efficient way of achieving it. We compare the support for complex I/O patterns provided by both PVFS1 and PVFS2 interfaces. First, we discuss a few functions in the PVFS1 interface API that provide non-contiguous access. Non-contiguous memory access is provided by the functions pvfs_readv and pvfs_writev. These functions do not however provide non-contiguous file access. The pvfs_read_list and pvfs_write_list give higher level parallel I/O libraries like MPI-IO basic support for non-contiguous accesses. Other than the above, non-contiguous support in PVFS1 is provided through logical partitioning. Logical partitioning allows the application to create a partition comprising regions of interest from a file. This partition is then accessed as a single unit thus saving multiple seek-access operations. But, the partitioning interface is only useful for single dimensional data or simple distributions of two dimensional data.

PVFS2 provides a highly expressive I/O request description facility that can support any data layout that can be constructed using MPI derived datatypes such as indexed or struct which can't be described using the partitioning. This is supported through the functions PVFS_sys_read and PVFS_sys_write. This can allow MPI-IO to take full advantage of PVFS2 features as MPI-IO derived datatypes allow the application to create complex I/O patterns to describe non-contiguous file accesses. The interpretation of the request description is handled by a separate module and the System Interface only exposes the ability to the application. The advantage of being able to describe complex non-contiguous I/O accesses to a file is important as such non-contiguous accesses are quite common in a parallel application. Supporting non-contiguous accesses in a single function call reduces the number of calls needed as well as the number of requests over the network. Speeding up a frequently occurring access pattern is hence a big win.

4.2.5 Clearly defined semantics

In this section we discuss specific semantic features that are clearly specified in the PVFS2 interface compared to the PVFS1 interface. The benefit of having well-defined semantics is clearly felt by the user of the parallel file system. The user can now tailor his applications likewise. In PVFS1 the semantics for file system operations is not clearly defined. Error handling, consistency, atomicity, handle reuse, and permission checking semantics are not specified. Also, the point where PVFS differs from POSIX is not defined. In the PVFS2 System Interface, consistency, atomicity, error handling handle reuse, permission checking semantics are quite clearly specified in a PVFS2 semantics document that eliminates confusion. As caching is performed at the System Interface level it is also necessary to define the mechanism followed for consistency of the cached entries. This is dealt with in the System Interface design document. The caches in PVFS2 have tunable consistency semantics implemented by means of a timeout that can be set to 0 to indicate that no caching is to be performed. With the semantics and its implementation specified in design documents we expect the utility of PVFS2 to increase.

4.2.6 Thread Safety

PVFS1 is not thread safe. This just means that the interface functions cannot be called by multiple threads. A threaded application can use the PVFS1 library by calling a single thread for the library functions and serializing operations. In comparison, the PVFS2 System Interface is designed to be thread safe. The System Interface functions can be called by multiple threads and will provide consistent results. The caches and configuration management structures in the System Interface use locking to provide thread safety. Global variables like errno are strictly avoided in the System Interface.

4.2.7 The PVFS2 directory cache

We demonstrate the motivation for the PVFS2 dcache by showing that the PVFS2 design leads to lookups of increased number and longer duration. The directory cache in PVFS2 is an outcome of having multiple servers storing metadata. In PVFS1, a single metaserver is theoretically a bottleneck during metadata servers so PVFS2 has multiple metaservers. In PVFS2, metadata is spread out over all metaservers without overlap so as to maximize parallelism. In the worst case, this could mean that the metadata for each object in a pathname is on a different server. With a simplistic lookup approach as in PVFS1 this leads to more requests over the network in PVFS2.

Thus, this does not scale well with increasing number of clients performing lookups and longer pathnames. This is where the PVFS2 dcache comes in. The dcache caches the name to handle mapping for each object after a successful lookup. As more lookups occur and as more entries are cached there will be further saving of time spent over network traversal. This approach also scales well with increased number of clients, longer pathnames, and increased number of metaservers. This is because the increased cache hits would balance out the increased requests. As an optimization, if metadata for successive segments in a path name is on the same metaserver then lookup recursively goes through the path until it finds a segment whose metadata is not on the same server.

4.2.8 The PVFS2 pinode cache

The function of the pinode cache is to cache the attributes encapsulated in a pinode. In PVFS1, the stat function is used to fetch metadata but no attempt is made to cache the attributes. Thus, each time there is a request for metadata a request is sent over the network to obtain them. To sum it up, the number of network messages involved in fetching metadata would be the double the number of actual metadata requests assuming one message each for a request and response. In PVFS2, as the System Interface function getattr could be used by many library calls, metadata may be fetched quite frequently. In addition, it is needed to validate handles in PVFS2, calculate directory size, and in I/O operations. The pinode cache makes use of the sizeable temporal locality exhibited at the System Interface. With each metadata request being satisfied from the cache itself, the number of network messages due to metadata requests are reduced and also the load on metadata servers is eased.

4.3 Summary

We believe that the objectives of the research have been satisfied by the implementation of the System Interface. The above comparative study proves that we have learned from the experience of the PVFS1 file system interface and have successfully corrected the shortcomings and also provided new features that will allow us to further explore issues in parallel I/O. We also hope to have demonstrated that the newer design features in the System Interface indeed make it better.

Chapter 5

Conclusion and Future Work

This document discussed the design and implementation of the System Interface for PVFS2. The System Interface was proposed as a file system interface for PVFS2 to meet the following demands.

- Flexibility
- Support for multiple interfaces
- Abstraction
- Robustness
- Performance

It has been shown that the System Interface meets the intended goals. The System Interface has been evaluated by comparing it with the file system interface in PVFS1. Through qualitative arguments we have tried to demonstrate that the System Interface is an improvement over the PVFS1 interface.

The functions in the first cut of the System Interface have been implemented. As work on the some of the underlying subsystems is currently in progress it was not possible to develop a working prototype of the entire PVFS2 system. We hope as the prototype is near completion that integration of the System Interface with the other modules can be done and quantitative results obtained.

5.1 Future work

As further testing is done on the System Interface there is scope for further improvements. Future work can be broadly divided into three areas - getting quantitative performance results using various libraries, implementing the remaining functions in the specification and adding features to make the interface more flexible.

5.1.1 Performance Results

This needs to be done with higher priority as this will eventually allow us to evaluate our design and identify bottlenecks in the System Interface. Some possible work in this regard may be the implementation of a POSIX compliant library and comparison of the results with PVFS1, comparison of ROMIO implementation results with PVFS1, VFS interface implementation results. Specific tests to determine the improvements due to client side caching could also be done.

5.1.2 Implement the remaining functions

The remaining functions in the specification pertaining to locking and symbolic link support need to be implemented so as to provide richer functionality in the file system interface.

5.1.3 Provide features to increase flexibility

There is scope to add features to make the interface more flexible. The configuration management interface can be provided with hints on access patterns so as to make intelligent choices while deciding the data layout. Support for non-blocking I/O could be provided. The error handling scheme needs to be implemented to provide info to the user when a problem occurs in a production environment.

Bibliography

[1] Mark Baker. Cluster computing white paper, 2000.

[2] Peter J. Braam. File systems for clusters from a protocol prespective. In *Second Extreme Linux Topics Workshop*, June 1999.

[3] P. H. Carns. Design and Analysis of a Network Transfer Layer for Parallel File Systems. *Clemson University Master's Thesis*, December 2001.

[4] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327. USENIX Association, 2000.

[5] The MPI-IO Committee. MPI-IO: A Parallel File I/O Interface for MPI Version 0.5.

[6] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong. Overview of the MPI-IO parallel I/O interface. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 477–487. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[7] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[8] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394. ACM Press, July 1995.

[9] Scalable I/O Initiative. SIO low-level application programming interface, November 1996.

[10] Michael J. Karels and Marshall Kirk Mckusick. Toward a compatible filesystem interface. In *Proceedings of the European Unix User's Group*, September 1986.

[11] David Kotz. Disk-directed I/O for MIMD multiprocessors, February 1997.

[12] David Kotz and Nils Nieuwejaar. Flexibility and performance of parallel file systems. In *Proceedings of the Third International Conference of the Austrian Center for Parallel Computation (ACPC)*, volume 1127 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, September 1996.

[13] Steven A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[14] Nils Nieuwejaar and David Kotz. The Galley parallel file system. *Parallel Computing*, 23(4):447–476, June 1997.

[15] Huseyin Simitci, Daniel A. Reed, Ryan Fox, Mario Medina, James Oly, Nancy Tran, and Gouyi Wang. A framework for adaptive storage input/output on computational grids. In *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, April 1999.

[16] Hal Stern. *Managing NFS and NIS*. O' Reilly, June 1991.

[17] PVFS2 Development Team. Trove: The PVFS2 Storage Interface. *PARL Internal Documentation*.

[18] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.