# Parallel Virtual File System, Version 2

PVFS2 Development Team

September, 2003

# Contents


**1 An introduction to PVFS2** **3**
- 1.1 Why rewrite? . . . 3
- 1.2 What's different? . . . 3
- 1.3 When will this be available? . . . 8

**2 The basics of PVFS2** **9**
- 2.1 Servers . . . 9
- 2.2 Networks . . . 10
- 2.3 Interfaces . . . 10
- 2.4 Client-server interactions . . . 10
- 2.5 Consistency from the client point of view . . . 11
- 2.6 File system consistency . . . 11

**3 PVFS2 terminology** **13**
- 3.1 File system components . . . 13
- 3.2 PVFS2 Objects . . . 14
- 3.3 Handles . . . 14
- 3.4 Handle ranges . . . 14
- 3.5 File system IDs . . . 15

**4 PVFS2 internal I/O API terminology** **16**
- 4.1 Internal I/O interfaces . . . 16
- 4.2 Job interface . . . 16
- 4.3 Posting and testing . . . 16
- 4.4 Test variations . . . 17
- 4.5 Contexts . . . 17
- 4.6 User pointers . . . 17
- 4.7 Time outs and max idle time . . . 17

**5 PVFS2 User APIs and Semantics** **19**
- 5.1 UNIX I/O Interface . . . 19
- 5.2 MPI-IO Interface . . . 21

**6 The code tree** **22**
- 6.1 The top level . . . 22
- 6.2 `src` . . . 23
- 6.3 `src/io` . . . 24
- 6.4 `test` . . . 24
- 6.5 State machines and `statecomp` . . . 25
- 6.6 Build system . . . 25
- 6.7 Out-of-tree builds . . . 25

# 1 An introduction to PVFS2

Since the mid-1990s we have been in the business of parallel I/O. Our first parallel file system, the Parallel Virtual File System (PVFS), has been the most successful parallel file system on Linux clusters to date. This code base has been used both in production mode at large scientific computing centers and as a launching point for many research endeavors.

However, the PVFS (or PVFS1) code base is a terrible mess! For the last few years we have been pushing it well beyond the environment for which it was originally designed. The core of PVFS1 is no longer appropriate for the environment in which we now see parallel file systems being deployed.

While we have been keeping PVFS1 relevant, we have also been interacting with application groups, other parallel I/O researchers, and implementors of system software such as message passing libraries. As a result have learned a great deal about how applications use these file systems and how we might better leverage the underlying hardware.

Eventually we reached a point where it was obvious to us that a new design was in order. The PVFS2 design embodies the principles that we believe are key to a successful, robust, high-performance parallel file system. It is being implemented primarily by a distributed team at Argonne National Laboratory and Clemson University. Early collaborations have already begun with Ohio Supercomputer Center and Ohio State University, and we look forward to additional participation by interested and motivated parties.

In this section we discuss the motivation behind and the key characteristics of our parallel file system, PVFS2.

## 1.1 Why rewrite?

There are lots of reasons why we've chosen to rewrite the code. We were bored with the old code. We were tired of trying to work around inherent problems in the design. But mostly we felt hamstrung by the design. It was too socket-centric, too obviously single-threaded, wouldn't support heterogeneous systems with different endian-ness, and relied too thoroughly on OS buffering and file system characteristics.

The new code base is much bigger and more flexible. Definitely there's the opportunity for us to suffer from second system syndrome here! But we're willing to risk this in order to position ourselves to use the new code base for many years to come.

## 1.2 What's different?

The new design has a number of important features, including:

- modular networking and storage subsystems,
- powerful request format for structured non-contiguous accesses,
- flexible and extensible data distribution modules,

- distributed metadata,
- stateless servers and clients (no locking subsystem),
- explicit concurrency support,
- tunable semantics,
- flexible mapping from file references to servers,
- tight MPI-IO integration, and
- support for data and metadata redundancy.

### 1.2.1 Modular networking and storage

One shortcoming of the PVFS1 system is its reliance on the socket networking interface and local file systems for data and metadata storage.

If we look at most cluster computers today we see a variety of networking technologies in place. IP, IB, Myrinet, and Quadrics are some of the more popular ones at the time of writing, but surely more will appear *and disappear* in the near future. As groups attempt to remotely access data at high data rates across the wide area, approaches such as reliable UDP protocols might become an important component of a parallel file system as well. Supporting multiple networking technologies, and supporting them *efficiently* is key.

Likewise many different storage technologies are now available. We're still getting our feet wet in this area, but it is clear that some flexibility on this front will pay off in terms of our ability to leverage new technologies as they appear. In the mean time we are certainly going to leverage database technologies for metadata storage – that just makes good sense.

In PVFS2 the Buffered Messaging Interface (BMI) and the Trove storage interface provide APIs to network and storage technologies respectively.

### 1.2.2 Structured non-contiguous data access

Scientific applications are complicated entities constructed from numerous libraries and operating on highly structured data. This data is often stored using high-level I/O libraries that manage access to traditional byte-stream files.

These libraries allow applications to describe complicated access patterns that extract subsets of large datasets. These subsets often do not sit in contiguous regions in the underlying file; however, they are often very structured (e.g. a block out of a multidimensional array).

It is imperative that a parallel file system natively support structured data access in an efficient manner. In PVFS2 we perform this with the same types of constructs used in MPI datatypes, allowing for the description of structured data regions with strides, common block sizes, and so on. This capability can then be leveraged by higher-level libraries such as the MPI-IO implementation.

### 1.2.3 Flexible data distribution

The tradition of striping data across I/O servers in round-robin fashion has been in place for quite some time, and it seems as good a default as any given no more information about how a file is going to be accessed. However, in many cases we *do* know more about how a file is going to be accessed. Applications have many opportunities to give the file system information about access patterns through various high-level interfaces. Armed with this information we can make informed decisions on how to better distribute data to match expected access patterns. More complicated systems could redistribute data to better match patterns that are seen in practice.

PVFS2 includes a modular system for adding new data distributions to the system and using these for new files. We're starting with the same old round-robin scheme that everyone is accustomed to, but we expect to see this mechanism used to better access multidimensional datasets. It might play a role in data redundancy as well.

### 1.2.4 Distributed metadata

One of the biggest complaints about PVFS1 is the single metadata server. There are actually two bases on which this complaint is typically launched. The first is that this is a single point of failure – we'll address that in a bit when we talk about data and metadata redundancy. The second is that it is a potential performance bottleneck.

In PVFS1 the metadata server steps aside for I/O operations, making it rarely a bottleneck in practice for large parallel applications, because they are busy writing data and not creating a billion files or some such thing. However, as systems continue to scale it becomes ever more likely that any such single point of contact might become a bottleneck for even well-behaved applications.

PVFS2 allows for configurations where metadata is distributed to some subset of I/O servers (which might or might not also serve data). This allows for metadata for different files to be placed on different servers, so that applications accessing different files tend to impact each other less.

Distributed metadata is a relatively tricky problem, but we're going to provide it in early releases anyway.

### 1.2.5 Stateless servers and clients

Parallel file systems (and more generally distributed file systems) are potentially complicated systems. As the number of entities participating in the system grows, so does the opportunity for failures. Anything that can be done to minimize the impact of such failures should be considered.

NFS, for all its faults, does provide a concrete example of the advantage of removing shared state from the system. Clients can disappear, and an NFS server just keeps happily serving files to the remaining clients.

In stark contrast to this are a number of example distributed file systems in place today. In order to meet certain design constraints they provide coherent caches on clients enforced via locking subsystems. As a

result a client failure is a significant event requiring a complex sequence of events to recover locks and ensure that the system is in the appropriate state before operations can continue.

We have decided to build PVFS2 as a stateless system and do not use locks as part of the client-server interaction. This vastly simplifies the process of recovering from failures and facilitates the use of off-the-shelf high-availability solutions for providing server failover. This does impact the semantics of the file system, but we believe that the resulting semantics are very appropriate for parallel I/O.

### 1.2.6 Explicit support for concurrency

Clearly concurrent processing is key in this type of system. The PVFS2 server and client designs are based around an explicit state machine system that is tightly coupled with a component for monitoring completion of operations across all devices. Threads are used where necessary to provide non-blocking access to all device types. This combination of threads, state machines, and completion notification allows us to quickly identify opportunities to make progress on particular operations and avoids serialization of independent operations within the client or server.

This design has a further side-effect of giving us native support for asynchronous operations on the client side. Native support for asynchronous operations makes nonblocking operations under MPI-IO both easy to implement and advantageous to use.

### 1.2.7 Tunable semantics

Most distributed file systems in use for cluster systems provide POSIX (or very close to POSIX) semantics. These semantics are very strict, arguably more strict than necessary for a usable parallel I/O system.

NFS does not provide POSIX semantics because it does not guarantee that client caches are coherent. This actually results in a system that is often unusable for parallel I/O, but is very useful for home directories and such.

Storage systems being applied in the Grid environment, such as those being used in conjunction with some physics experiments, have still different semantics. These tend to assume that files are added atomically and are never subsequently modified.

All these very different approaches have their merits and applications, but also have their disadvantages. PVFS2 will not support POSIX semantics (although one is welcome to build such a system on top of PVFS2). However, we do intend to give users a great degree of flexibility in terms of the coherency of the view of file data and of the file system hierarchy. Users in a tightly coupled parallel machine will opt for more strict semantics that allow for MPI-IO to be implemented. Other groups might go for looser semantics allowing for access across the wide area. The key here is allowing for the possibility of different semantics to match different needs.

### 1.2.8 Flexible mapping from file references to servers

Administrators appreciate the ability to reconfigure systems to adapt to changes in policy or available resources. In parallel file systems, the mapping from a file reference to its location on devices can help or hinder reconfiguration of the file system.

In PVFS2 file data is split into *datafiles*. Each datafile has its own reference, and clients identify the server that owns a datafile by checking a table loaded at configuration time. A server can be added to the system by allocating a new range of references to that server and restarting clients with an update table. Likewise, servers can be removed by first stopping clients, next moving datafiles off the server, then restarting with a new table. It is not difficult to imagine providing this functionality while the system is running, and we will be investigating this possibility once basic functionality is stable.

### 1.2.9 Tight MPI-IO coupling

The UNIX interface is a poor building block for an MPI-IO implementation. It does not provide the rich API necessary to communicate structured I/O accesses to the underlying file system. It has a lot of internal state stored as part of the file descriptor. It implies POSIX semantics, but does not provide them for some file systems (e.g. NFS, many local file systems when writing large data regions).

Rather than building MPI-IO support for PVFS2 through a UNIX-like interface, we have started with something that exposes more of the capabilities of the file system. This interface does not maintain file descriptors or internal state regarding such things as file positions, and in doing so allows us to better leverage the capabilities of MPI-IO to perform efficient access.

We've already discussed rich I/O requests. "Opening" a file is another good example. `MPI_File_open()` is a collective operation that gathers information on a file so that MPI processes may later access it. If we were to build this on top of a UNIX-like API, we would have each process that would potentially access the file call `open()`. In PVFS2 we instead resolve the filename into a handle using a single file system operation, then broadcast the resulting handle to the remainder of the processes. Operations that determine file size, truncate files, and remove files may all be performed in this same $O(1)$ manner, scaling as well as the MPI broadcast call.

### 1.2.10 Data and metadata redundancy

Another common (and valid) complaint regarding PVFS1 is its lack of support for redundancy at the server level. RAID approaches are usable to provide tolerance of disk failures, but if a server disappears, all files with data on that server are inaccessible until the server is recovered.

Traditional high-availability solutions may be applied to both metadata and data servers in PVFS2 (they're actually the same server). This option requires shared storage between the two machines on which file system data is stored, so this may be prohibitively expensive for some users.

A second option that is being investigated is what we are calling *lazy redundancy*. The lazy redundancy

approach is our response to the failure of RAID-like approaches to scale well for large parallel I/O systems when applied across servers. The failure of this approach at this scale is primarily due to the significant change in environment (latency and number of devices across which data is striped). Providing the atomic read/modify/write capability necessary to implement RAID-like protocols in a distributed file system requires a great deal of performance-hampering infrastructure.

With lazy redundancy we expose the creation of redundant data as an explicit operation. This places the burden of enforcing consistent access on the user. However, it also opens up the opportunity for a number of optimizations, such as creating redundant data in parallel. Further, because this can be applied at a more coarse grain, more compute-intensive algorithms may be used in place of simple parity, providing higher reliability than simple parity schemes.

Lazy redundancy is still at the conceptual stage. We're still trying to determine how to best fit this into the system as a whole. However, traditional failover solutions may be put in place for the existing system.

#### 1.2.11 And more...

There are so many things that we feel we could have done better in PVFS1 that it is really a little embarrassing. Better heterogeneous system support, a better build system, a solid infrastructure for testing concurrent access to the file system, an inherently concurrent approach to servicing operations on both the client and server, better management tools, even symlinks; we've tried to address most if not all the major concerns that our PVFS1 users have had over the years.

It's a big undertaking for us. Which leads to the obvious next question.

### 1.3 When will this be available?

Believe it or not, right now. At SC2004 we released PVFS2 1.0. We would be foolish to claim PVFS2 has no bugs and will work for everyone 100% of the time, but we feel PVFS2 is in pretty good shape. Early testing has found a lot of bugs, and we feel PVFS is ready for wider use.

Note that we're committed to supporting PVFS1 for some time after PVFS2 is available and stable. We feel like PVFS1 is a good solution for many groups already, and we would prefer for people to use PVFS1 for a little while longer rather than them have a sour first experience with PVFS2.

We announce updates frequently on the PVFS2 mailing lists. We encourage users to subscribe – it's the best way to keep abreast of PVFS2 developments. All code is being distributed under the LGPL license to facilitate use under arbitrarily licensed high-level libraries.

# 2 The basics of PVFS2

PVFS2 is a parallel file system. This means that it is designed for parallel applications sharing data across many clients in a coordinated manner. To do this with high performance, many servers are used to provide multiple paths to data. Parallel file systems are a subset of distributed file systems, which are more generally file systems that provide shared access to distributed data, but don't necessarily have this focus on performance or parallel access.

There are lots of things that PVFS2 is *not* designed for. In some cases it will coincidentally perform well for some arbitrary task that we weren't targeting. In other cases it will perform very poorly. If faced with the option of making the system better for some other task (e.g. executing off the file system, shared mmapping of files, storing mail in mbox format) at the expense of parallel I/O performance, we will always ruthlessly ignore performance for these other tasks.

PVFS2 uses an *intelligent server* architecture. By this we mean that servers do more than simply provide clients with blocks of data from disks, instead talking in higher-level abstractions such as files and directories. An alternative architecture is *shared storage*, where storage devices (usually accessed at a block granularity) are directly addressed by clients. The intelligent server approach allows for clever algorithms that could not be applied were block-level accesses the only mechanism clients had to interact with the system because more appropriate remote operations can be provided that serve as building blocks for these algorithms.

In this section we discuss the components of the system, how clients and servers interact with each other, consistency semantics, and how the file system state is kept consistent without the use of locks. In many cases we will compare the new system with the original PVFS, for those who are may already be familiar with that architecture.

## 2.1 Servers

In PVFS1 there were two types of server processes, *mgrs* that served metadata and *iods* that served data. For any given PVFS1 file system there was exactly one active mgr serving metadata and potentially many *iods* serving data for that file system. Since mgrs and iods are just UNIX processes, some users found it convenient to run both a mgr and an iod on the same node to conserve hardware resources.

In PVFS2 there is exactly one type of server process, the *pvfs2-server*. This is also a UNIX process, so one could run more than one of these on the same node if desired (although we will not discuss this here). A configuration file tells each pvfs2-server what its role will be as part of the parallel file system. There are two possible roles, metadata server and data server, and any given pvfs2-server can fill one or both of these roles.

PVFS2 servers store data for the parallel file system locally. The current implementation of this storage relies on UNIX files to hold file data and a Berkeley DB database to hold things like metadata. The specifics of this storage are hidden under an API that we call Trove.

## 2.2 Networks

PVFS2 has the capability to support an arbitrary number of different network types through an abstraction known as the Buffered Messaging Interface (BMI). At this time BMI implementations exist for TCP/IP, Myricom's GM, and InfiniBand (both Mellanox VAPI and OpenIB APIs).

## 2.3 Interfaces

At this time there are exactly two low-level I/O interfaces that clients commonly use to access parallel file systems. The first of these is the UNIX API as presented by the client operating system. The second is the MPI-IO interface.

In PVFS1 we provide access through the operating system by providing a loadable module that exports VFS operations out into user space, where a client-side UNIX process, the *pvfsd*, handles interactions with servers. A more efficient in-kernel version called *kpvfsd* was later provided as well.

PVFS2 uses a similar approach to the original PVFS1 approach for access through the OS. A loadable kernel module exports functions out to user-space where a UNIX process, the *pvfs2-client*, handles interactions with servers. We have returned to this model (from the in-kernel kpvfsd model) because it is not clear that we will have ready access to all networking APIs from within the kernel.

The second API is the MPI-IO interface. We leverage the ROMIO MPI-IO implementation for PVFS2 MPI-IO support, just as we did for PVFS1. ROMIO links directly to a low-level PVFS2 API for access, so it avoids moving data through the OS and does not communicate with pvfs2-client.

## 2.4 Client-server interactions

At start-up clients contact any one of the pvfs2-servers and obtain configuration information about the file system. Once this data has been obtained, the client is ready to operate on PVFS2 files.

The process of initiating access to a file on PVFS2 is similar to the process that occurs for NFS; the file name is resolved into an opaque reference, or *handle*, through a lookup operation. Given a handle to some file, any client can attempt to then access any region of that file (permission checks could fail). If a handle becomes invalid, the server will reply at the time of attempted access that the handle is no longer valid.

Handles are nothing particularly special. We can look up a handle on one process, pass it to another via an MPI message, and use it at the new process to reference the same file. This gives us the ability to make the `MPI_File_open` call happen with a single lookup the the file system and a broadcast.

There's no state held on the servers about "open" files. There's not even a concept of an open file in PVFS2. So this lookup is all that happens at open time. This has a number of other benefits. For one thing, there's no shared state to be lost if a client or server disappears. Also, there's nothing to do when a file is "closed" either, except perhaps ask the servers to push data to disk. In an MPI program this can be done by a single process as well.

Of course if you are accessing PVFS2 through the OS, `open` and `close` still exist and work the way you would expect, as does `lseek`, although obviously PVFS2 servers don't keep up with file positions either. All this information is kept locally by the client.

There are a few disadvantages to this. One that we will undoubtedly hear about more than once is that the UNIX behavior of unlinked open files. Usually with local file systems if the file was previously opened, then it can still be accessed. Certain programs rely on this behavior for correct operation. In PVFS2 we don't know if someone has the file open, so if a file is unlinked, it is gone gone gone. Perhaps we will come up with a clever way to support this or adapt the NFS approach (renaming the file to an odd name), but this is a very low priority.

## 2.5 Consistency from the client point of view

We've discussed in a number of venues the opportunities that are made available when true POSIX semantics are given up. Truthfully very few file systems actually support POSIX; ext3 file systems don't enforce atomic writes across block boundaries without special flags, and NFS file systems don't even come close. Never the less, many people claim POSIX semantics, and many groups ask for them without knowing the costs associated.

PVFS2 does not provide POSIX semantics.

PVFS2 does provide guarantees of atomicity of writes to nonoverlapping regions, even noncontiguous nonoverlapping regions. This is to say that if your parallel application doesn't write to the same bytes, then you will get what you expect on subsequent reads.

This is enough to provide all the non-atomic mode semantics for MPI-IO. The atomic mode of MPI-IO will need support at a higher level. This will probably be done with enhancements to ROMIO rather than forcing more complicated infrastructure into the file system. There are good reasons to do this at the MPI-IO layer rather than in the file system, but that is outside the context of this document.

Caching of the directory hierarchy is permitted in PVFS2 for a configurable duration. This allows for some optimizations at the cost of windows of time during which the file system view might look different from one node than from another. The cache time value may be set to zero to avoid this behavior; however, we believe that users will not find this necessary.

## 2.6 File system consistency

One of the more complicated issues in building a distributed file system of any kind is maintaining consistent file system state in the presence of concurrent operations, especially ones that modify the directory hierarchy.

Traditionally distributed file systems provide a locking infrastructure that is used to guarantee that clients can perform certain operations atomically, such as creating or removing files. Unfortunately these locking systems tend to add additional latency to the system and are often *extremely* complicated due to optimizations and the need to cleanly handle faults.

We have seen no evidence either from the parallel I/O community or the distributed shared memory community that these locking systems will work well at the scales of clusters that we are seeing deployed now, and we are not in the business of pushing the envelope on locking algorithms and implementations, so we're not using a locking subsystem.

Instead we force all operations that modify the file system hierarchy to be performed in a manner that results in an atomic change to the file system view. Clients perform sequences of steps (called *server requests*) that result in what we tend to think of as atomic operations at the file system level. An example might help clarify this. Here are the steps necessary to create a new file in PVFS2:

- create a directory entry for the new file
- create a metadata object for the new file
- point the directory entry to the metadata object
- create a set of data objects to hold data for the new file
- point the metadata at the data objects

Performing those steps in that particular order results in file system states where a directory entry exists for a file that is not really ready to be accessed. If we carefully order the operations:

1. create the data objects to hold data for the new file
2. create a metadata object for the new file
3. point the metadata at the data objects
4. create a directory entry for the new file pointing to the metadata object

we create a sequence of states that always leave the file system directory hierarchy in a consistent state. The file is either there (and ready to be accessed) or it isn't. All PVFS2 operations are performed in this manner.

This approach brings with it a certain degree of complexity of its own; if that process were to fail somewhere in the middle, or if the directory entry turned out to already exist when we got to the final step, there would be a great deal of cleanup that must occur. This is a problem that can be surmounted, however, and because none of those objects are referenced by anyone else we can clean them up without concern for what other processes might be up to – they never made it into the directory hierarchy.

# 3 PVFS2 terminology

PVFS2 is based on a somewhat unconventional design in order to achieve high performance, scalability, and modularity. As a result, we have introduced some new concepts and terminology to aid in describing and administering the system. This section describes the most important of these concepts from a high level.

## 3.1 File system components

We will start by defining the major system components from an administrator or user's perspective. A PVFS2 file system may consist of the following pieces (some are optional): the pvfs2-server, system interface, management interface, Linux kernel driver, pvfs2-client, and ROMIO PVFS2 device.

The `pvfs2-server` is the server daemon component of the file system. It runs completely in user space. There may be many instances of pvfs2-server running on many different machines. Each instance may act as either a metadata server, an I/O server, or both at once. I/O servers store the actual data associated with each file, typically striped across multiple servers in round-robin fashion. Metadata servers store meta information about files, such as permissions, time stamps, and distribution parameters. Metadata servers also store the directory hierarchy.

Initial PVFS2 releases will only support one metadata server per file system, but this restriction will be released in the future.

The `system interface` is the lowest level user space API that provides access to the PVFS2 file system. It is not really intended to be an end user API; application developers are instead encouraged to use MPI-IO as their first choice, or standard Unix calls for legacy applications. We will document the system interface here, however, because it is the building block for all other client interfaces and is thus referred to quite often. It is implemented as a single library, called libpvfs2. The system interface API does not map directly to POSIX functions. In particular, it is a stateless API that has no concept of open(), close(), or file descriptors. This API does, however, abstract the task of communicating with many servers concurrently.

The `management interface` is similar in implementation to the system interface. It is a supplemental API that adds functionality that is normally not exposed to any file system users. This functionality is intended for use by administrators, and for applications such as fsck or performance monitoring which require low level file system information.

The `Linux kernel driver` is a module that can be loaded into an unmodified Linux kernel in order to provide VFS support for PVFS2. Currently this is only implemented for the 2.6 series of kernels. This is the component that allows standard Unix applications (including utilities like `ls` and `cp`) to work on PVFS2. The kernel driver also requires the use of a user-space helper application called `pvfs2-client`.

`pvfs2-client` is a user-space daemon that handles communication between PVFS2 servers and the kernel driver. Its primary role is to convert VFS operations into `system interface` operations. One pvfs2-client must be run on each client that wishes to access the file system through the VFS interface.

The `ROMIO PVFS2 device` is a component of the ROMIO MPI-IO implementation (distributed separately) that provides MPI-IO support for PVFS2. ROMIO is included by default with the MPICH MPI

implementation and includes drivers for several file systems. See http://www.mcs.anl.gov/romio/ for details.

## 3.2 PVFS2 Objects

PVFS2 has four different object types that are visible to users

- directory
- metafile
- datafile
- symbolic link

...

## 3.3 Handles

`Handles` are unique, opaque, integer-like identifiers for every object stored on a PVFS2 file system. Every file, directory, and symbolic link has a handle. In addition, several underlying objects that cannot be directly manipulated by users are represented with handles. This provides a concise, non path dependent mechanism for specifying what object to operate on when clients and servers communicate. Servers automatically generate new handles when file system objects are created; the user does not typically manipulate them directly.

The allowable range of values that handles may assume is known as the `handle space`.

## 3.4 Handle ranges

Handles are essentially very large integers. This means that we can conveniently partition the handle space into subsets by simply specifying ranges of handle values. `Handle ranges` are just that; groups of handles that are described by the range of values that they may include.

In order to partition the handle space among N servers, we divide the handle space up into N handle ranges, and delegate control of each range to a different server. The file system configuration files provide a mechanism for mapping handle ranges to particular server hosts. Clients only interact with handle ranges; the mapping of ranges to servers is hidden beneath an abstraction layer. This allows for greater flexibility and future features like transparent migration.

## 3.5 File system IDs

Every PVFS2 file system hosted by a particular server has a unique identifier known as a `file system ID` or `fs id`. The file system ID must be set at file system creation time by administrative tools so that they are synchronized across all servers for a given file system. File systems also have symbolic names that can be resolved into an fs id by servers in order to produce more readable configuration files.

File system IDs are also occasionally referred to as collection IDs.

# 4 PVFS2 internal I/O API terminology

PVFS2 contains several low level interfaces for performing various types of I/O. None of these are meant to be accessed by end users. However, they are pervasive enough in the design that it is helpful to describe some of their common characteristics in a single piece of documentation.

## 4.1 Internal I/O interfaces

The following is a list of the lowest level APIs that share characteristics that we will discuss here.

- BMI (Buffered Message Interface): message based network communications
- Trove: local file and database access
- Flow: high level I/O API that ties together lower level components (such as BMI and Trove) in a single transfer; handles buffering and datatype processing
- Dev: user level interaction with kernel device driver
- NCAC (Network Centric Adaptive Cache): user level buffer cache that works on top of Trove (*currently unused*)
- Request scheduler: handles concurrency and scheduling at the file system request level

## 4.2 Job interface

The Job interface is a single API that binds together all of the above components. This provides a single point for testing for completion of any nonblocking operations that have been submitted to a lower level API. It also handles most of the thread management where applicable.

## 4.3 Posting and testing

All of the APIs listed in this document are nonblocking. The model used in all cases is to first `post` a desired operation, then `test` until the operation has completed, and finally check the resulting error code to determine if the operation was successful. Every `post` results in the creation of a unique ID that is used as an input to the `test` call. This is the mechanism by which particular posts are matched with the correct test.

It is also possible for certain operations to complete immediately at post time, therefore eliminating the need to test later if it is not required. This condition is indicated by the return code of the post call. A return code of 0 indicates that the post was successful, but that the caller should test for completion. A return code of 1 indicates that the call was immediately successful, and that no test is needed. Errors are indicated by either a negative return code, or else indicated by an output argument that is specific to that API.

### 4.4 Test variations

In a parallel file system, it is not uncommon for a client or server to be carrying out many operations at once. We can improve efficiency in this case by providing mechanisms for testing for completion of more than one operation in a single function call. Each API will support the following variants of the test function (where PREFIX depends on the API):

- PREFIX_test(): This is the most simple version of the test function. It checks for completion of an individual operation based on the ID given by the caller.
- PREFIX_testsome(): This is an expansion of the above call. The difference is that it takes an array of IDs and a count as input, and provides an array of status values and a count as output. It checks for completion of any non-zero ID in the array. The output count indicates how many of the operations in question completed, which may range from 0 to the input count.
- PREFIX_testcontext(): This function is similar to testsome(). However, it does not take an array of IDs as input. Instead, it tests for completion of *any* operations that have previously been posted, regardless of the ID. A count argument limits how many results may be returned to the caller. A context (discussed in the following subsection) can be used to limit the scope of IDs that may be accessed through this function.

### 4.5 Contexts

Before posting any operations to a particular interface, the caller must first open a `context` for that interface. This is a mechanism by which an interface can differentiate between two different callers (ie, if operations are being posted by more than one thread or more than one higher level component). This context is then used as an input argument to every subsequent post and test call. In particular, it is very useful for the testcontext() functions, to insure that it does not return information about operations that were posted by a different caller.

### 4.6 User pointers

`User pointers` are void* values that are passed into an interface at post time and returned to the caller at completion time through one of the test functions. These pointers are never stored or transmitted over the network; they are intended for local use by the interface caller. They may be used for any purpose. For example, it may be set to point at a data structure that tracks the state of the system. When the pointer is returned at completion time, the caller can then map back to this data structure immediately without searching because it has a direct pointer.

### 4.7 Time outs and max idle time

The job interface allows the caller to specify a time out with all test functions. This determines how long the test function is allowed to block before returning if no operations of interest have completed.

The lower level APIs follow different semantics. Rather than a time out, they allow the caller to specify a `max idle time`. The max idle time governs how long the API is allowed to sleep if it is idle when the test call is made. It is under no obligation to actually consume the full idle time. It is more like a hint to control whether the function is a busy poll, or if it should sleep when there is no work to do.

# 5 PVFS2 User APIs and Semantics

Because PVFS2 is designed specifically for performance in systems where concurrent access from many processes is commonplace, there are some differences between the PVFS2 interfaces and traditional file system interfaces. In this section we will discuss the two interfaces provided for applications to use when accessing PVFS2 file systems. We will start with the traditional UNIX I/O interface, which nearly all file systems implement. We will then cover the MPI-IO interface.

## 5.1 UNIX I/O Interface

We provide an implementation of the UNIX I/O interface for clients running Linux versions 2.4 (2.4.19 and later) and 2.6. This interface implements the traditional `open`, `read`, `write`, and `close` interface as well as providing the directory operations necessary for applications such as `ls` to work.

It is important to note that there is a difference between implementing the UNIX I/O API and implementing the POSIX semantics for this API. File systems exported via NFS (versions 2 and 3) do not exhibit many of the POSIX semantics, and even local file systems may not guarantee atomicity of writes that cross disk block boundaries. We also do not implement the full POSIX semantics. Here we will document aspects of the POSIX semantics that we do not implement.

### 5.1.1 Permission Checking

To understand why PVFS2 permission checking behaves differently from the POSIX standard, it is useful to discuss how PVFS2 performs permission checking. PVFS2 does not really implement the `open` etc. interface, but instead uses a stateless approach that relies on the client to `lookup` a file name to convert it into a *handle* that can be used for subsequent read and write accesses. This handle may be used for many read and write accesses and may be cached under certain guidelines. This lookup operation is performed at file open time.

Permission checking is performed in two places. First, the VFS checks permissions on the client and will prevent users from performing invalid operations. Second, the server performs rudimentary checks for specific operations; however, it (currently) relies on the client to provide accurate user and group information to be used for this purpose. No recursive directory permission checking is performed by the servers, for example.

### 5.1.2 Permissions and File Access

POSIX semantics dictate that once a file has been opened it may continue to be accessed by the process until closed, regardless of changes to permissions.

In PVFS2, the effect of permission changes on a file may or may not be immediately apparent to a client holding an open file descriptor. Because of the manner in which PVFS2 performs permission checking and

file lookup, it is possible that a client may lose the ability to access a file that it has previously opened due to permission change, if for example the cached handle is lost and a lookup is performed again.

### 5.1.3 Access to Removed Files

POSIX semantics dictate that a file opened by a process may continue to be accessed until the subsequent close, even if the file permissions are changed or the file is deleted. This requires that the file system or clients somehow keep up with a list of files that are open, which adds unacceptable state to a distributed file system. In NFS, for example, this is implemented via the "sillyrename" approach, in which clients rename a deleted but open file to hide it in the directory tree, then delete the renamed file when the file is finally closed.

In PVFS2 a file that is deleted is removed immediately regardless of open file descriptors. Subsequent attempts to access the file will find that the file no longer exists.

*Neill: Is this completely true, or do clients delay removal in pvfs2-client if someone is still accessing?*

### 5.1.4 Overlapping I/O Operations

POSIX semantics dictate sequential consistency for overlapping I/O operations. This means that I/O operations must be atomic with respect to each other – if one process performs a read spanning a collection of servers while another performs a write in the same region, the read must see either all or none of the changes. In a parallel file system this involves communication to coordinate access in this manner, and because of the number of clients we wish to support, we are unwilling to implement this functionality (at least as part of the core file system).

In PVFS2 we instead implement a semantic we call *nonconflicting writes* semantic. This semantic states that all I/O operations that do not access the same bytes (in other words, are nonconflicting) will be sequentially consistent. Write operations that conflict will result in some undefined combination of the bytes being written to storage, while read operations that conflict with write operations will see some undefined combination of original data and write data.

### 5.1.5 Locks

BSD provides the `flock` mechanism for locking file regions as a way to perform atomic modifications to files. POSIX provides this functionality through options to `fcntl`. Both of these are advisory locks, which means that processes not using the locks can access the file regions.

PVFS2 does not implement a locking infrastructure as part of the file system. At this time there is no add-on advisory locking component either. Thus neither the `flock` function nor the `fcntl` advisory locks are supported by PVFS2.

## 5.2 MPI-IO Interface

We provide an implementation of the MPI-IO interface via an implementation of the ROMIO ADIO interface. This implementation is included as part of MPICH1 and MPICH2 as well as being available as an independent package. The MPI-IO interface is part of the MPI-2 standard and defines an API for file access from within MPI programs.

Our PVFS2 implementation does not include all the functionality of the MPI-IO specification. In this section we discuss the missing components of MPI-IO for PVFS2.

### 5.2.1 MPI-IO Atomic Mode

Atomic mode is enabled by calling `MPI_File_set_atomicity` and setting the atomicity to true. Atomic mode guarantees that data written on one process is immediately visible to another process (as in the POSIX default semantics).

ROMIO currently uses file locking to implement the MPI-IO atomic mode functionality. Because we do not support locks in PVFS2, atomic mode is not currently supported.

### 5.2.2 MPI-IO Shared Pointer Mode

Shared pointers are used in the `_shared` and `_ordered` families of functions.

The ROMIO implementation relies on the use of locking support from the file system to implement both of these families of functions, so these are not currently supported. We are researching alternative implementations.

# 6 The code tree

In this section we describe how the code tree is set up for PVFS2 and discuss a little about how the build system works.

## 6.1 The top level

At the top level we see:

- `doc`
- `examples`
- `include`
- `lib`
- `maint`
- `src`
- `test`

The `doc` directory rather obviously holds documentation, mostly written in LaTeX. There are a few subdirectories under `doc`. The `coding` subdirectory has a document describing guidelines for writing code for the project. The `design` subdirectory has a number of documents describing various components of the system and APIs and more importantly currently houses the Quick Start.

Much of the documentation is out of date.

`examples` currently holds two example server configuration files and that is it.

`include` holds include files that are both used all over the system and are eventually installed on the system during a `make install`. Any prototypes or defines that are needed by clients using the API should be in one of the include files in this directory.

`lib` is empty. It holds `libpvfs2.a` when it is built, prior to installation, if you are building in-tree. More on out-of-tree builds later.

`maint` holds a collection of scripts. Some of these are used in the build process, while others are used to check for the presence of inappropriately named symbols in the resulting library or reformat code that doesn't conform to the coding standard.

`src` holds the source code to the majority of PVFS2, including the server, client library, Linux 2.6.x kernel module, and management tools. We'll talk more about this one in a subsequent subsection.

`test` holds the source code to many many tests that have been built over time to validate the PVFS2 implementation. We will discuss this more in a subsequent subsection as well.

## 6.2 src

The src directory contains the majority of the PVFS2 distribution.

Unlike PVFS1, where the PVFS kernel code was in a separate package from the "core," in PVFS2 both the servers, client API, and kernel-specific code are packaged together.

src/common holds a number of components shared between clients and servers. This includes:

- dotconf – a configuration file parser
- gen-locks – an implementation of local locks used to provide atomic access to shared structures in the presence of threads
- id-generator – a simple system for generating unique references (ids) to data structures
- llist – a linked-list implementation
- gossip – our logging component
- quicklist – another linked-list implementation
- quickhash – a hash table implementation
- statecomp – the parser for our state machine description language (discussed subsequently)
- misc – leftovers, including some state machine code, config file manipulation code, some string manipulation utilities, etc.

src/apps holds applications associated with PVFS2. The src/apps/admin subdirectory holds a collection of tools for setting up, monitoring, and manipulating files on a PVFS2 file system. pvfs2-genconfig is used to create configuration files. pvfs2-cp may be used to move files on and off PVFS2 file systems. pvfs2-ping and pvfs2-statfs may be used to check on the status of a PVFS2 file system. pvfs2-ls is an ls implementation for PVFS2 file systems that does not require that the file system be mounted.

The src/apps/karma subdirectory contains a gtk-based gui for monitoring a PVFS2 file system in real time. The src/apps/kernel/linux-2.6 subdirectory contains the user space component of the PVFS2 kernel driver implementation, which matches the kernel driver code found in src/kernel/linux-2.6. The src/apps/vis contains experimental code for performance visualization.

src/client holds code for the "system interface" library, the lowest level library used on the client side for access. This is in the src/client/sysint subdirectory. The unix-io subdirectory is no longer used. Note that there is other code used on the client side: the ROMIO components (included in MPICH and MPICH2) and the kernel support code (located in src/kernel, discussed subsequently).

Note that the ROMIO support for PVFS2 is included in MPICH1, MPICH2, and ROMIO distributions and is not present anywhere in this tree.

src/server holds code for the pvfs2-server. The request scheduler code is split into its own subdirectory for no particular reason.

src/proto holds code for encoding and decoding our over-the-wire protocol. Currently the "encoding scheme" used is the *contig* scheme, stored in its own subdirectory. This encoding scheme really just puts the bytes into a contiguous region, so it is only good for homogeneous systems or systems with the same byte orders where we have correctly padded all the structures (which we probably haven't).

src/kernel holds implementations of kernel support. Currently there is only one, src/kernel/linux-2.6.

src/io holds enough code that we'll just talk about it in its own subsection.

## 6.3 src/io

This directory holds all the code used to move data across the wire, to store and retrieve data from local resources, to buffer data on servers, and to describe I/O accesses and physical data distribution.

bmi holds the Buffered Messaging Interface (BMI) implementations. The top-level directory holds code for mapping to the various underlying implementations and defining common data structures. Subdirectories hold implementations for GM (bmi_gm), TCP/IP (bmi_tcp), and InfiniBand using either the VAPI or OpenIB API (bmi_ib).

buffer holds the implementation of our internal buffering and caching component. At the time of writing this is not complete and is not enabled.

dev holds code that understands how to move data through a device file that is used by our Linux 2.6 kernel module. This is stored in this directory because it is hooked under the *job* component.

job holds the job component. This component is responsible for providing us with a common mechanism for queueing and testing for completion of operations on a variety of different resources, including all BMI, Trove, and the device listed above.

description holds code used to describe I/O accesses and physical data distributions.

trove holds implementations of the Trove storage interface. The top-level directory holds code for mapping to the various underlying implementations and defining common data structures. Currently there is only a single implementation of Trove, called DBPF (for DB Plus Files). This implementation builds on Berkeley DB and local UNIX files for storing local data.

flow holds the Flow component implementations. These components are responsible for ferrying data between different types of *endpoints*. Valid endpoints include BMI, Trove, memory, and the buffer cache.

## 6.4 test

This directory holds a great deal of test code, most of which is useless to the average user.

`test/client/sysint` has a collection of tests we have used when implementing (or reimplementing) various system interface functions.

`test/correctness/pts` holds the PVFS Test Suite (PTS), a suite designed for testing the correctness of PVFS under various different conditions. There are actually quite a few tests in here, and the vision is that we will run these in an automated fashion relatively often (but we aren't there quite yet). This is probably the second most useful code (after pvfs2-client) in the `test` directory.

### 6.5 State machines and `statecomp`

The PVFS2 source is heavily dependent on a state machine implementation that is included in the tree. We've already noted that the parser, statecomp, is located in the `src/common/statecomp` subdirectory. Additional code for processing state machines is in `src/common/misc`.

State machine source is denoted with a `.sm` suffix. These are converted to `.c` files by statecomp. If you are building out of tree, the `.c` files will end up in the build tree; otherwise you'll be in the confusing situation of having both versions in the same subdirectory. If modifying these, be careful to only modify the `.sm` files – the corresponding `.c` file can be overwritten on rebuilds.

### 6.6 Build system

The build system relies on the "single makefile" concept that was promoted by someone or another other than us (we should have a reference). Overall we're relatively happy with it.

We also adopted the Linux 2.6 kernel style of obfuscating the actual compile lines. This can be irritating if you're trying to debug the build system. It can be turned off with a "make V=1", which makes the build verbose again. This is controlled via a variable called `QUIET_COMPILE` in the makefile, if you are looking for how this is done.

### 6.7 Out-of-tree builds

Some of the developers are really fond of out-of-tree builds, while others aren't. Basically the idea is to perform the build in a separate directory so that the output from the build process doesn't clutter up the source tree.

This can be done by executing `configure` from a separate directory. For example:

```
# tar xzf pvfs2-0.0.2.tgz
# mkdir BUILD-pvfs2
# cd BUILD-pvfs2
# ../pvfs2/configure
```