PVFS 2 File System Semantics Document

PVFS Development Team

May 2002

1 Introduction

This document describes the file system semantics of PVFS2, both in terms of how it behaves and in terms of how this behavior is implemented. Rationale for decisions is included in order to motivate decisions.

We start by discussing the semantics of server operations. We follow this with a discussion of one client side implementation and its semantics.

The discussion is broken into use cases. First we will cover the issue. Second we will discuss the semantics (to be) implemented in PVFS2. Finally we will discuss the implications of these semantics on the file system design and implementation.

In some cases we will provide alternatives for semantics and/or the implementation.

2 Definitions

We will define *overlapping writes* to be concurrent writes that modify the same bytes in an object. We will define *interleaved writes* to be concurrent writes that modify different bytes within a common extent (but do not modify the same bytes).

3 Server Semantics

In this section we discuss the semantics that are enforced by a server with respect to operations queued for service on the server. At times this will delve down into the trove and/or BMI semantics.

The server scheduler component is responsible for enforcing these semantics/policies.

Note: we're not counting on inter-server communication at this time.

3.1 Permissions and permission checking

The server will perform any permission checking on incoming operations before queuing them for service.

Permission checking on the server is limited to checking that can occur on the object itself (as opposed to checking that would occur, for example, to verify access to a file through a given path).

It's not clear if datafiles have permissions yet.

Operations on the metafile use the permissions on the metafile at the time of the operation.

Probably datafiles don't have permissions for now.

3.2 Removing an object that is being accessed

The server will not remove an object while it is being accessed. For example, if a trove operation is in progress reading data from a datafile, the server will queue a subsequent remove operation on that object until the read operation completes.

The server is free to return "no such file" results to future operations on that object even if it has not completed the remove operation (assuming that permission checking has been performed to verify that the remove will occur) or to queue these operations until the remove has occurred, then allow them to fail. Obviously the first of these options is preferable.

3.3 Overlapping writes

The server will allow overlapping and interleaved writes to be concurrently processed by the underlying storage subsystem (trove). Trove will ensure that in the interleaved case, the resulting data pattern is the union of the modified bytes of both operations. In the overlapping case trove is free to ignore all but one of the data values to be written to each byte or to write them all in some undefined order.

3.4 Handle reuse

Servers will guarantee that handles spend a minimum amount of time out of use before they are reused. This time value will be known to clients.

3.4.1 Implementation

Need to handle the case where a handle has been used, we're in the middle of this wait time, and the server gets restarted. To handle this we will need some kind of disk-resident list of handles along with some lower bound on how recently they were put in the unused list.

3.5 Symbolic links

Symbolic links will be stored on servers. The “target” of the link need not exist, as with traditional symlinks.

3.6 Top level scheduler semantics

Where does this go?

We need a list of types of operations that shouldn’t overlap. This is the rule set for the scheduler, or at least part of it.

4 Client-side library without locks or inter-client communication

This section describes what will be our first, non-locking approach to metadata caching that does not involve client file system code communicating with other client instances. This scheme relies on timeouts. There are potentials for inconsistencies, just as there are in NFS, if the timeouts don’t match well with access patterns.

Obviously this is just one of many possible client-side library implementations. It just happens to be the first one we’re going to implement.

Note the operations at this level

Timeouts will be tunable at runtime to allow administrators to tailor their system to the workloads presented. A zero timeout is always possible, meaning that that type of information is never cached.

One should think of these timeouts as providing a window of time during which the view of one client can differ from the view of another client. There are a number of aspects to the view of interest:

- attributes of objects
- locations of objects in the name space
- existence of objects
- data in objects

In this section we cover one mechanism for limiting the potential inconsistencies between client views.

Do we want some kind of optional data caching? If so, this changes our concurrent write model.

4.1 Caching of file and directory attributes

The most obvious and important data to cache from a performance standpoint is attributes associated with file system objects. Here we are referring to information such as the owner and group of the file, the permissions, and PVFS-specific information like file data distribution.

We will assume that the data distribution for a file does not change over the lifetime of the file (?).

A timeout will be associated with this information.

Implications on file size.

4.1.1 Implementation

Possible to use vtags for verification that data hasn't changed.

4.2 Caching of directory hierarchy

Implications on renaming of files, directories, parent directories.

4.3 Handle reuse

It is important that clients be able to identify when a handle has been reused by the system and is no longer a reference to the original object. Given a finite handle space, we know that we will eventually run out of unique handles and have to reuse old, freed ones.

We will impose a timeout on reuse of handles by servers in order to allow clients to have an expectation of how long they can hold onto handle data while assuming that the handle still refers to the same object (if it is still valid). This timeout will create a window of time during which a client can detect that the handle has been freed.

This timeout should be quite large so that it in general does not affect the client.

Note need for revisiting client code to allow for refresh.

If this timeout does occur, then the client needs to check to see if the handle that it has still refers to the same object as before. This can be done by comparing the create time of the cached object handle with the create time stored on the server. Create times will be accurate and nonmodifiable on servers. An equality test can be used for this test; this avoids potential problems with clock skew/reset. It does not imply that clocks must be in sync between the various servers.

Name the timeout.

4.3.1 Implementation

Server must keep a time associated with freed handles. Groups of handles can be put together with a single time, because our space is big enough that this shouldn't be a problem. This will need to be written to disk so that this policy can be upheld in the face of a restart.

We should do the math on how long we should wait, work out the specifics of what goes to disk (how it goes to disk doesn't have to be here).

On the client side, we will want the ability to get control to the client code even if there aren't ops to perform. This could be a function call, or there could be a thread in the client code, or maybe the client code is its own entity (e.g. pvfsd). All options to list here.

4.4 Metadata not in cache

A common occurrence in PVFS1 is that one client will create a file that is subsequently accessed by a number of other clients. It is important semantically that it is never the case that an explicit attempt to access a recently created file fails because of our caching policy. Because of this, any time a file system object is explicitly referenced that is not in the cache, we will assume that the cache is not up to date (regardless of the timeout values described above) and attempt to obtain metadata for the object. Only if this fails will we return an error indicating that the object does not exist.

This is an instance of an implicit hint from the user that something in the cache might not be up to date. We should consider where other such hints occur.

Any time a request is received on a client to operate on a file system object that is not known to the client-side cache, the client will attempt to retrieve metadata for the object.

4.5 Concurrent, byte-overlapping writes to a single file

One of the most inconvenient of the POSIX I/O semantics is its specification of how concurrent, overlapping writes should be handled. To be POSIX compliant, sequential consistency must be maintained in the face of concurrent, overlapping writes.

In PVFS1 we didn't support this semantic. The premise was that application programmers are not really doing this, that a well-written application does not have multiple processes writing to the same bytes in the file. Instead the PVFS1 semantics said that writes that do not have overlapping bytes will occur exactly as requested, even if the bytes are interleaved, but that if overlapping does occur the result can be any combination of the bytes from the two writers.

In practice we have found with PVFS1 that while some people seem concerned that we do not meet the POSIX semantics, no actual application groups have pointed to this as a show-stopper. Perhaps some DB groups have been concerned, but then PVFS is really poorly designed for the types of patterns that these applications would produce anyway (probably).

For these reasons we will begin in PVFS2 with the same concurrent, overlapping write semantics.

Concurrent, non-overlapping writes will result in the desired data being written to disk regardless of interleaving of data down to the byte granularity. Concurrent, overlapping writes (at the byte level) will result in data from one of the requests being written to each byte in the file, but for every given byte the data written could be from any one of the concurrent operations.

4.5.1 Implementation

We will assume that trove is able to handle the interleaved writes and support these semantics, so the higher level components of the server may pass down any combination of writes that it likes.

4.6 Concurrent file create

This happens all the time in parallel applications, even with ROMIO at the moment.

Only one instance must be created. Everyone must then get the right handle. When a handle is returned, it must be ready to be used for I/O.

4.6.1 Implementation

To create a PVFS2 "file", there are actually three things that have to happen. A metafile must be created to hold attributes. A collection of datafiles must be created to hold the data. A dirent in the parent directory must be created to add the file into the name space.

There are a bunch of options for implementing this correctly:

- dirent first
- dirent second
- dirent last
- hash to metafile
- server-supported w/server communication

In the first four schemes, the clients are totally responsible for creating all the components of a PVFS2 file.

In the dirent first scheme, the dirent is created as the first step. Following this the other objects are allocated and filled in. The advantage of this approach is that clients that lose the race to create the file will do so on the first step (as opposed to the dirent last case, described below). This means that the minimum amount of redundant work occurs. However, the dirent can't even have a valid handle in it if it is created first, meaning that the dirent will have to be modified a second time by the creator to fill in the right value (once the metafile is allocated). This leaves a window of time during which the dirent exists but refers to a file that has no attributes and cannot be read.

In the dirent second scheme, clients first allocate a metafile with parameters indicating that it isn't complete, then allocate the dirent. This means that losing clients will all allocate a metafile (and then have to free it). However, it also provides a valid set of attributes that could be seen during the window of time that the file is being created. Datafiles would be allocated last, meaning that the client would have to modify the distribution information in the metafile after it has been added into the namespace; however, a valid handle would already exist in the name space, resulting in a cleaner client-side mechanism for updating the distribution information once it is filled in. Clients attempting to read/write a file with cached distribution information that isn't filled in will necessarily need to update their cache and potentially wait for this to finish.

In the dirent last scheme clients first allocate datafiles, then the metafile (filling in the distribution information), then finally fill in the dirent. This scheme has the benefit of at all times providing a consistent, complete name space. It has the drawback of a lot of work on the client side for the losers to "undo" all the allocation that they performed before failing to obtain the dirent.

The hash to metafile scheme relies on the use of the vesta-like hashing scheme for directly finding metafiles. This scheme is listed here just to keep it in mind; we don't expect to use the hashing scheme at this time. In this scheme the metafile is created first. Since all clients will hash to the same server, and a path name is associated with the

metafile (in this scheme), the server would allow only one metafile to be created. After this the winner could allocate datafiles and finally create the dirent. It's not a bad scheme, but we're not doing the hashing right now because of costs in other operations.

In the last scheme server communication is used to coordinate creation of all the objects that make up a file. The server holding the directory is told to create the PVFS file. It creates the metafile and datafiles before adding the dirent. The server scheduler can ensure that only one create completes.

We will implement the dirent second scheme.

4.7 Moving files

Concurrent moves can be tricky. The biggest concern is eliminating any point during which a file might have two references in the namespace.

A secondary concern is that of a concurrent create of the destination file while the move is in progress.

4.7.1 Implementation

Clients will perform moves in the following way:

- delete original dentry
- create new dentry

By performing the operations in this order, we preserve the "no more than one reference" semantic listed above.

Need to handle the create/move in some way. How?

A second approach is possible given inter-server communication. In this approach, a scheme can be applied that eliminates the create/move problem. In this description we denote the server that originally holds the dentry as sv1 and the new holder of the dirent as sv2.

- sv1 receives move request
- sv1 ensures no other operations will proceed on old dirent until complete (through scheduler)
- sv1 creates new dirent on sv2
- on success, deletes original dirent
- on failure, returns failure to client

4.8 Deleting a file that is being accessed

POSIX semantics dictate that a file deleted while held open by another process remains available through the reference that the process holds until the process dies or closes the file (verify that this is a POSIX thing).

PVFS1 actually attempts to support this semantic.

We will not try to support this in PVFS2. Clients with it open will all of a sudden get ENOFILE or something similar. Too much state must be maintained to provide this functionality (either on client or server side). We're not going to do this sort of thing on the server side, so unless we have communicating clients, we aren't going to get this behavior.

4.8.1 Implementation

Delete the dirent first, then the metafile, then the datafiles. I think.

4.9 Permissions and permission checking

Whole path permission checking is performed at lookup time (i.e. when someone attempts to get a handle). This will verify that they can read the metadata.

Object-level permission checking will be performed at read/write. This is when we determine if the user can access data. An implication of this is that at the UNIX level successfully opening a file for writing does not guarantee that the user will continue to be able to write to the file through the file descriptor.

Both of these checked are performed using cached data.

Metafile operations get metafile permission checks using cached data.

We should look at NAS authentication mechanisms and try to find one that we can leverage as a future project.

4.10 Readdir with concurrent directory changes

Vtags will be used to ensure that directory changes are noticed on client side.

This means that we need a vtag parameter in the request.

We will restart the directory read process in the event of a change.

4.11 Time synchronization

NOTE: For now we are setting ctime, atime, and mtime at creation using time values computed on the client side. We may need to change this later...

How do we get the atimes and mtimes right for files? Do we make this a derived value (as in PVFS1)? If so, we need to have tight clock synchronization, or we need some way for adjusting for clock skew (e.g. passing current time plus atime or mtime, letting server do the math).

4.12 Computing file size

How? It's a derived value. If no server communication, then we'll need to talk to all the owners of datafiles and get their sizes, then do some dist-specific math to get the actual size.

When do we want to ensure that the file size is exact? What does truncate do WRT the datafiles? Does it just make sure that the right one is big enough to show that the file should be so big, or do we do something to all the datafiles?

4.13 Non-blocking calls

We should have some for read/write in addition to the blocking calls.

We may want a thread under the API to handle progress.

5 UNIX-like interface

In this section we describe the implications of the server semantics and caching client semantics on a hypothetical UNIX-like interface.

Note the operations at this level.

5.1 Implementing`O_APPEND` with and without concurrent access

Implications of attribute caching on O_APPEND. Notes on concurrent O_APPEND vs. not.

Concurrent O_APPEND is nondeterministic.

5.2 Permissions and permission checking

Permissions can change while file is open, can all of a sudden fail.

5.3 Truncate

Our truncate always resizes when possible (man pages indicate that it might or might not grow files).

5.4 Hard links

No such thing.

5.5 Symlinks

What to say? Can have targets that don't exist.

6 Misc.

Don't quite know where this stuff goes yet.

6.1 Adding I/O servers

Is there anything tricky here? There is if the servers communicate.

6.2 Migrating files and changing distributions

Distributions don't change for a given metafile. So we need to get a new metafile and go from there. This is also the appropriate way to move metafiles around in order to balance the metadata load (if necessary).

6.3 Metafile stuffing

This is our version of "inode stuffing", the technique used to store small files in the inode data space rather than allocating blocks for data (in local file systems and such).

In our version for small files we can simply store the data in the bytestream space associated with the metafile. In fact, one type of file system that one could build with PVFS2 would always to this.

How do we know when to use this? Do we ever switch from doing this to the tradational datafile approach? How do we do that? It might not be so hard with server communication, but how do we do it without?

6.4 Adding metaservers

At the moment we have said that we will have a static metaserver list. How might we approach using a dynamic list?