# Trove Database + Files (DBPF) Implementation

PVFS Development Team

October 11, 2026

## 1 Introduction

The purpose of this document is to sketch out the Database + Files (DBPF) implementation of the Trove storage interface. Included will be discussion of how one might use the implementation in practice.

DBPF uses UNIX files and Berkeley DB (3 or 4 I think) databases to store file and directory data and metadata.

## 2 Entities on Physical Storage

The locations of these entities are defined in dbpf.h, although in future versions they should be relative to the storage name passed in at initialize time.

For a given server there are single instances of each of the following:

- storage attributes DB – holds attributes on the storage space as a whole
- collections DB – holds information on collections and is used to map from a collection name to a collection ID

For each collection there are one of each of the following:

- collection attributes DB – holds attributes for the collection as a whole, including handle usage information and root handle
- dataspace attributes DB – holds dbpf_dspace_attr structure for each dataspace, referenced by handle

In addition, dataspaces have a one to one mapping to the following, although they are created lazily (so they will not exist if not yet used):

- bstream file – UNIX files that holds bstream data, just as in the PVFS1 iod
- keyval DB – holds keyvals for a specific dataspace

At this time both bstream files and keyval DBs are stored in a flat directory structure. This may change to be a hashed directory system as in the PVFS1 iod if/when we see that we are incurring substantial overhead from lookups in a directory with many entries.

# 3 Hooking to Upper Trove Layer

The DBPF implementation hooks to the upper trove "wrapper" layer through a set of structures that contain pointers to the functions that make up the Trove API. These structures are defined in the upper-layer trove.h and are TROVE_bstream_ops, TROVE_keyval_ops, TROVE_dspace_ops, TROVE_mgmt_ops, and TROVE_fs_ops.

The TROVE_mgmt_ops structure includes a pointer to the function initialize(). This is used to initialize the underlying Trove implementation (e.g. DBPF). The initialize function takes the storage name as a parameter; this should be used to locate the various entities that are used for storing the DBPF collections.

*Additionally a method ID is passed in...why???*

The method name field is filled in by the DBPF implementation to describe what type of underlying storage is available.

# 4 Operation Queue

The DBPF implementation has its own queue that it uses to hold operations in progress. Operations may have one of a number of states (defined in dbpf.h):

- OP_UNITIALIZED
- OP_NOT_QUEUED
- OP_QUEUED
- OP_IN_SERVICE
- OP_COMPLETED
- OP_DEQUEUED

# 5 Dataspaces Implementation

Dataspaces are stored as:

- an entry in the dataspace DB
- zero or one keyval DBs (zero if no keyvals have been set)
- zero or one bstream UNIX files (zero if no data has been written to the bstream)

Keyval and bstream files are named by the handle (because they don't really have names...).

# 6 Keyval Spaces Implementation

Currently all keyval space operations use blocking DB calls. They queue the operation when the I/O call is made, and they service the operation when the service function is called (at test time).

On each call the database is opened and closed, rather than caching the FD. This will obviously need to change in the near future; probably we'll do something similar to the bstream fdcache.

The dbpf_keyval_iterate_op_svc function walks through all the keyval pairs. The current implementation uses the btree format with record numbers to make it easy to pick up where the last call to the iterator function left off. The iterator function will process *count* items per call: if fewer than *count* items are left in the database, we set *count* to how many items were processed. The caller should check that *count* has the same value as before the function was called.

# 7 Bytestreams Implementation

The read_at and write_at functions are implemented using blocking UNIX file I/O calls. They queue the operation when the I/O call is made, and they service the operation when the service function is called (at test time).

The read_list and write_list functions are implemented using lio_listio. Both of these functions actually call a function dbpf_bstream_rw_list, which implements the necessary functionality for both reads and writes. This function sets up the queue entry and immediately calls the service function.

The necessary aiocb array is allocated within the service function if one isn't already allocated for the operation. The maximum number of aiocb entries is controlled by a compile-time constant AIOCB_ARRAY_SZ; if there are more list elements than this, the list will be split up and handled by multiple lio_listio calls.

If the service function returns to dbpf_bstream_rw_list without completing the entire operation, the operation is queued.

Both the _at and _list functions use a FD cache for getting open FDs.

# 8 Creating a "File System"

Theoretically Trove in general doesn't know anything about "file systems" per se. However, it's helpful for us to provide functions intended to ease the creation of FSs in Trove. These functions are subject to change or removal as we get a better understanding of how we're going to use Trove.

There's more than one way to build a file system given the infrastructure provided by Trove; we're just going through a single example here.

At this time, a single file system is associated with one and only one collection, and a single collection is associated with one and only one file system.

## 8.1 Creating a Storage Space

First a storage space must be created. Storage spaces are created with the storage_create mgmt operation. *This function can be used without calling the initialize function?* This function creates the storage attributes database and the collections database.

## 8.2 Creating a Collection

Given that a storage space has been created, the next thing to do is create a collection. This is done with the collection_create mgmt operation. This function takes a collection name and an associated collection ID. First it looks up the storage space (which currently is done by mapping to the single possible storage space). Next it checks to see that a collection with that name does not already exist (and errors out if it does). Next it will create the collection by writing a new entry into the collections DB. Following this it will create a dataspace attributes database and the directories for keyval and bstream spaces for the collection. Then it creates the collection attributes database and puts a last handle value into it, which is used in subsequent dataspace creates.

Once a collection is created, it can be looked up with collection_lookup. Collections are looked up by name. This will return collection ID that can be used to direct subsequent operations to the collection. Currently the implementation only supports creation of one collection.

*The collection lookup routine opens the collection attribute database and finds the root handle. This really isn't right...the collection routines don't need to know about this...the fs routines should instead.*

It also opens the dataspace database and returns the collection ID.

In addition to creating one collection for the file system, a second "administrative" collection will be created. Currently, the handle allocator uses the admin collection to store handle state in a bstream.

## 8.3 Creating a Root Directory

There is a collection of file system helper functions too...

## 8.4 Creating Dataspaces

At this point dataspaces can be created with dspace_create.

The dataspace create code checks to see if the handle is in use. *Currently this code will not try to come up with a new handle if the proposed one is used...fix?*

An entry is placed in the dataspace attribute DB for the collection.

*What is the "type" field used for?*

I think that the basic metadata for a file will be stored in the dataspace attribute DB by adding members to the dbpf_dspace_attr structure (defined in dbpf.h).

### 8.4.1 Creating a Directory

First we create a dataspace. Then we add a key/value to map the name of the directory to the handle into the parent directory keyval space.

### 8.4.2 Creating a File

First we create a dataspace. Then we add a key/value to map the name to the handle into the parent directory keyval space.

### 8.4.3 Deleting a Directory

We check to make sure the "directory" has no elements in it. Then we remove the key/value mapping the name of the directory to the handle from the parent directory keyval space. After removing the key/value, we mark the dataspace as ready for deletion.

### 8.4.4 Deleting a File

Given a file name, we retrieve the corresponding handle from the parent directory keyval space. Entries in the parent directory keyval space map the name of the file to a handle – the handle of the dataspace for that keyval space. We remove the key/value for the given name. We then use the handle we retrieved to find the dataspace and mark it as ready for deletion.

# 9 Current Deficiencies

Only one collection can be used with the current implementation.

Error checking is weak at best. Assert(0)s are used in many places in error conditions.