The Design of Flat File Feature for PLFS

1. The motivation and benefits

PLFS was originally designed for large N-1 workloads in which a large number of processes write to a shared file concurrently. This caused lock contention in the file system causing severely inefficient utilization of the disk spindles.

PLFS used a container mechanism to allow the N writers of a shared logical file to actually physically write to independent objects. Later, PLFS added distributed metadata by hashing objects across a set of backends.

However, for large N-N workloads in which a large number of processes write to unique files, the containers for each unique file add a fair bit of overhead. Each container consists of a minimum of about 10 physical objects. For an N-1 logical file, an extra 10 objects is completely amortorized by large numbers of N. But for N-N workloads, this is a large amount of overhead.

Luckily, PLFS can fix this by continuing to use the distributed metadata layer but to create normal flat-files for each logical file instead of containers.

1. The high level design

Currently the PLFS code is written based on the container structure with a lot of assumptions that works only for containers. However, in order to implement the flat file mode for PLFS, these assumptions should be identified and disabled or removed for flat file.

After some discussions, all of us agreed that we need a mechanism which could separate the implementation of container and flat-files while not hurting the readability and clarity. For this purpose, we designed a layer structure for PLFS to uncouple the libplfs interface and the implementation details of container.

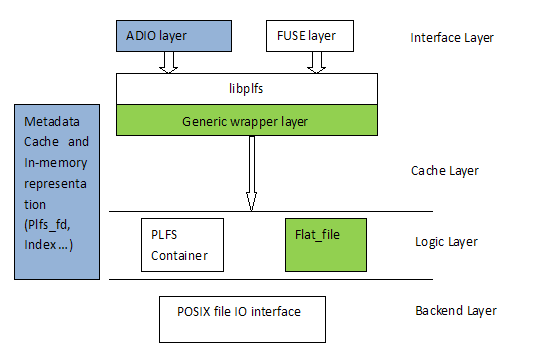


Figure Layered Structure of PLFS

The generic wrapper layer is constructed by two pure virtual classes. Both of the container and flat-file are derived from these two classes with their own implementations. The libplfs interface is based on the two virtual classes and it need not care about whether the underlying files are containers or flat-files.

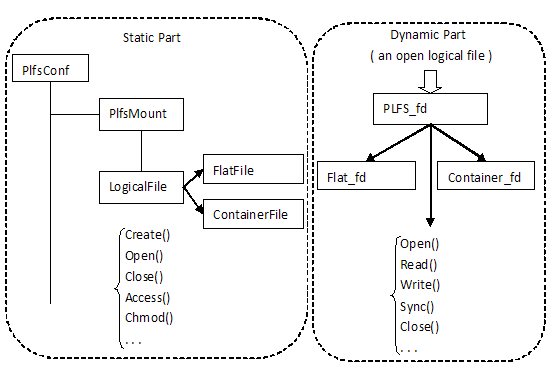


Figure Virtual Classes in Generic Wrapper Layer

The static part can also be called the stateless part. It is called static because once the plfsrc file is loaded, these classes won’t change. And it is called stateless because there isn’t anything we need record, they are operated on pathnames.

And the dynamic part is created when the file is opened and stays valid until the file is closed. So these function calls are stateful, it should record the state of this file in its \*\_fd structures. And once a file is opened, the library will create a Flat\_fd or a Container\_fd object and return to the application as a PLFS\_fd \*, when the application call read/write with PLFS\_fd, the code in library can call PLFS\_fd->read/write directly and it will be converted to Flat\_fd->read/write or Container\_fd->read/write accordingly by the virtual function mechanism in C++.

1. The detailed implementation of flat file
   1. Refactoring the container code

The container code does not fit in the new architecture, so refactoring the container code is necessary. Firstly the two virtual classes are defined. The class LogicalFileSystem is the LogicalFile in Figure 2. And the class Plfs\_fd is the class PLFS\_fd in Figure 2. They are the abstraction of file system operations.

The original libplfs interface functions such as plfs\_open, plfs\_close, plfs\_rename and plfs\_read, are tightly coupled with the implementation details of the container structure. So they would be renamed to container\_open, container\_close, container\_rename and container\_read. And the original class Plfs\_fd is renamed to Container\_OpenFile. These functions and data structures are renamed so that they can be used as much as possible later.

Then the libplfs interface is re-implemented based on these two virtual classes. Take the plfs\_rename() and plfs\_read() for example:

121 int

122 plfs\_rename(const char \*from, const char \*to) {

123 LogicalFileSystem \*logicalfs = plfs\_get\_logical\_fs(from);

124 if (logicalfs == NULL) return -EINVAL;

125 return logicalfs->rename(from, to);

126 }

For these static functions, the configuration file is parsed and loaded at the beginning of it, and according to the pathname of this file, its corresponding mount point information PlfsMount \* can be found. Then we can get the pointer to the LogicalFileSystem object and call its member functions accordingly.

102 ssize\_t

103 plfs\_read(Plfs\_fd \*fd, char \*buf, size\_t size, off\_t offset) {

104 return fd->read(buf, size, offset);

105 }

For the dynamic functions, a file descriptor is allocated when the file is opened. And this descriptor will be passed to following operations such as read or write. Then when we want to read some data from this file, we can directly call the file descriptor’s member functions. Plfs\_fd is a pure virtual class and fd points to a flat file or container file descriptor so that the right member functions will be called.

Then two derived classes: ContainerFileSystem and Container\_fd should be implemented. ContainerFileSystem derived from LogicalFileSystem contains the implementation of static functions for container files, and the Container\_fd derived from Plfs\_fd contains the implementation of dynamic functions. They are implemented using corresponding container\_\* functions. Take the ContainerFileSystem->rename for example:

52 int

53 ContainerFileSystem::rename(const char \*logical, const char \*to) {

54 return container\_rename(logical, to);

55 }

And the Container\_fd class contains a member variable ‘fd’, which is a pointer to Container\_OpenFile object, and this object could be passed to container\_read or container\_write as a file descriptor.

25 ssize\_t

26 Container\_fd::read(char \*buf, size\_t size, off\_t offset) {

27 return container\_read(fd, buf, size, offset);

28 }

By this way, the original code is refactoring to fit in the high level architecture and the old code can be re-used as much as possible.

* 1. Implementing the flat file mode

After the code refactoring, flat file mode can be easily added to PLFS by deriving two classes from LogicalFileSystem and Plfs\_fd. FlatFileSystem derived from LogicalFileSystem contains the implementation of static functions and Flat\_fd derived from Plfs\_fd contains the implementation of dynamic functions.

152 int

153 FlatFileSystem::chmod( const char \*logical, mode\_t mode ) {

154 FLAT\_ENTER;

155 ret = Util::Chmod(path.c\_str(),mode);

156 FLAT\_EXIT(ret);

157 }

Take the chmod for example, the logical file pathname is expanded to physical pathname and stored in the variable path. Then the Util::Chmod() is called to do the operation on the physical file on a backend directly.

65 ssize\_t

66 Flat\_fd::read(char \*buf, size\_t size, off\_t offset) {

67 int ret = Util::Pread(backend\_fd, buf, size, offset);

68 FLAT\_EXIT(ret);

69 }

There is a member variable backend\_fd in class Flat\_fd, which stored the file descriptor of the physical file on a backend. When a PLFS file is opened, the physical file is opened too and its descriptor is assigned to backend\_fd in Flat\_fd. When the PLFS file is to be read, the Util::Pread() is called with backend\_fd directly.

Directory related operations are complicated for flat file. When a directory is created in PLFS, all backends should create a directory with the same name. And when reading a PLFS directory, all of the corresponding directories in each backend should be read. However, there is no difference in directory processing for containers and flat file. So we can re-use the code for containers to implement the directory operations for flat files.

228 int

229 FlatFileSystem::mkdir(const char \*logical, mode\_t mode) {

230 return container\_mkdir(logical, mode);

231 }

232

233 int

234 FlatFileSystem::readdir(const char \*logical, void \*buf) {

235 return container\_readdir(logical, buf);

236 }

246 int

247 FlatFileSystem::rmdir(const char \*logical) {

248 return container\_rmdir(logical);

249 }