Operating System Support for Fan-Out File Systems

Charles P. Wright, Puja Gupta, Harikesevan Krishnan,Kiran-Kumar Muniswamy-Reddy
Mohammad Zubair, and Erez Zadok
Stony Brook University

Stackable file systems are a portable mechanism to extend file system functionality without the complexity of creating an entirely new disk-based or network-based file system. Normally, applications execute system calls, then the VFS calls, the file system directly. With stacking, the VFS calls the stackable file system which in turn calls the lower-level file system. An important class of stackable file systems is *fan-out* file systems in which one upper-level object can be represented by more than one lower-level object.

Traditional stackable file systems form a linear stack of layers. Fan-out file systems, however, use multiple underlying directories. Examples include unification, replication, load balancing, RAID-like file systems, sandboxing (privilege or access separation), and others. Fan-out can even occur within a single directory: for example, checksums, indexes, or extra mappings can be stored in databases or auxiliary files.

Unfortunately, current OS infrastructure is insufficient to support fan-out file systems: error handling and reporting needs to improve in the face of partial errors; file systems need to be able to disambiguate between conflicting data; and remote file system components need improved communication.

Partial Failures The POSIX API essentially has a binary return value (0 for true and -1 for false); but by their very nature, fan-out file systems are more complicated than a simple binary return value. When an operation fails in a fan-out file system, there are two distinct cases: (1) the operation fails completely, and (2) the operation only partially fails. The current APIs are not equipped to pass the complex status of a fan-out operation, which describes whether the operation was successful on each underlying object and if not, report the appropriate error code; existing error codes are too rigid and cannot be extended easily. File systems, indeed all kernel components, should be able to dynamically register new error codes and associated messages with the kernel.

There are three approaches that we are exploring to deal with partial failures. First, we are designing fan-out file systems that hide as possible partial failures from user-space. Second, we are modifying the POSIX API to return complex error codes. Third, we are developing a generic transaction mechanism for file systems, such that they roll back partially completed actions. In our unification file system, *unionfs*, we are using various methods to reduce the number of partial errors. Unionfs merges the contents of several directories, called *branches*. Unionfs can be used to merge the contents of split CD-ROMs or to manage source code and object files. We assign each branch a precedence, the branches with a higher precedence are to the left of directories with a lower precedence. When an error occurs, we require that the view of the file system does not change, rather than the more stringent requirement that nothing changes on disk. To reduce the number of partial errors, we use careful ordering of operations. For example, to remove a file in unionfs we must ensure that the user's view of the file system never changes unless the operation succeeds. To satisfy this invariant, we remove files going from right to left. If the operation fails, then the leftmost file is not deleted and the file system remains the same. If the operation proceeds all the way to the leftmost branch, then the operation is successful. Ordering operations preserves the contents of Unionfs, but not the underlying file systems. To handle partial errors with no on-disk changes, we must implement VFS transactions and rollback. Presently several file systems have support for transactions, but only deep inside the internals of the file system. Transactions should be exposed to other kernel (and user-space) components, and should be part of the VFS rather than being reimplemented in each individual file system.

Disambiguating Data In a fan-out file system, data comes from various sources and can conflict. In Unionfs, for example, an object can exist as a file in one branch and a directory in the next. We use the left-to-right precedence rule to resolve this conflict, but in other file systems (e.g., replication) this indicates data corruption and the method for resolving it is not clear. We are exploring a file-system level checker (ala `fsck`) to resolve such conflicts.

A more subtle form of conflict occurs when enumerating directory entries. If there are two branches with distinct permissions making up a union, then the following can occur: user cpw may read only the left branch, but user ezk may read both branches. Since the directory entry caches are global, not per-user, cpw could gain access to entries that he should not or entries will be missing from ezk's view. To resolve this we will create per-user views of a single file system.

Communication For several new file systems, communication between file systems across the network will be required. For example, a replication file system needs to re-synchronize replicas after errors and a secure network file system needs to transmit authentication tokens from client to server. We are developing methods to communicate using only standard VFS methods. The two methods of communication we are developing are using reserved file names that act as pipes and extended attributes. Each file system can open a special file, for example `/.vfs_pipe`, on the remote stackable file system. Data can then be transferred using the standard read and write VFS calls. This can be useful for bulk data transfers, but results in complex overloading of existing VFS operations. Alternatively, *Extended Attributes* (EAs) are divided into separate namespaces, so we can choose a unique namespace and not modify any existing semantics. Setting an EA invokes special methods, and retrieving an EA obtains a return code (similarly to `/proc` entries on Linux). EAs are tied to an inode, so an operation is invoked on the object it uses.