NetApp Memo

NetworkAppliance®

pNFS: Extensions to NFSv4 to Support Data Distribution and Parallelism

Peter Corbett

November 27, 2003

Position Statement

The goal of this proposal is to describe how NFSv4 can be extended by the minor versioning process to allow a higher degree of distribution of file system contents, and to support parallel files. We also propose an extension that allows concurrent operations on the server. Preferably, the extensions would be introduced into NFSv4.1. The new features that we add would all be optional. The goal is to allow NFSv4.1 clients to directly access data stored on multiple servers connected to form a single name space and a coordinated file service. This proposal will be completely compatible with NFS RDMA extensions in the RPC layer.

There are four areas that we address in this proposal:

- Unordered operations.
- Data locations, allowing directory contents to be distributed across multiple servers.
- Distribution of parallel files across servers.
- Server-to-server operations.

The basic model we propose is that a number of servers can cooperatively present a single file system with a single fsid. A file can be composed of a single metafile, local to the server containing the directories that link to it, and one or more data forks that can be distributed among the other servers that store portions of the file system. This allows distribution of the data portions of files in the same directory (directory scaling) as well as distribution of the data of individual files (file scaling).

Unordered Operations

DAFS 1.0 defined a batch I/O and I/O list capability in the protocol. This allows multiple regions of a file to be read or written in one I/O operation, and even allows multiple regions of different files to be read or written in one I/O operation. Batched operations are collected into completion groups, which allow polling or waiting upon completion of any of a number of simultaneously issued component I/Os.

We propose that a more limited functionality be made available, possibly in NFSv4.1. The key consideration is to allow a client to request a number of operations simultaneously. Currently, multiple individual read and write operations can be grouped into a compound request in NFSv4. However, the server is required to execute these operations in order, and to return failure if any of the operations fails while ensuring that none of the following operations in the compound has completed. We propose allowing an “unordered” section in the compound, containing a list of operations that the

server is free to execute concurrently or in arbitrary sequence. The order of the operations in the response is unchanged.

The operations in an unordered set can be performed on multiple files. To ensure that sequences of operations on multiple files are unambiguous, the set of operations is ordered with respect to PUTFH. In other words, PUTFH operations in the unordered set of operations apply to the unordered operations that follow them in the request. For example, an unordered sequence of PUTFH, READ, PUTFH, READ... will have the identical effect as same sequence of operations appearing outside of an unordered section, but the server can perform the reads concurrently or in any sequence.

The primary intended use of unordered operations is in the implementation of client-side I/O list operations. The I/O list passes multiple I/Os to the client in one system call. The client can then construct a compound with an unordered set of I/O operations, one per operation in the I/O list.

The unordered capability is useful in at least two application domains. One typical use of the I/O list capability is in scientific computing. When large multi-dimensional data sets are stored in a single file, decomposing those data sets for access by parallel programs often requires strided access through the file. I/O list operations make constructing the calls and issuing the requests simpler. Presenting the list to the server in an unordered section of a compound allows the server to perform the request more efficiently.

A second use of this capability is in cache cleaners and prefetchers, for example, by databases. The cleaner and prefetcher usually run as a set of separate threads or processes on the host database server. If I/O list capability is provided, then the cleaner and prefetcher can issue multiple page I/Os in one system call. This allows the server to perform these operations more efficiently. It also allows the database server to run with fewer cleaner and prefetcher threads.

We suggest adding two new operations to specify the beginning and end of an unordered list of operations. UNORDEREDBEGIN specifies to the server that the following operations can be performed in any order. UNORDEREDEND specifies to the server that the set of unordered operations is complete. Operations after the UNORDEREDEND operation must be performed only after all operations in the preceding unordered section are complete. Status is returned for each operation including UNORDEREDBEGIN and UNORDEREDEND. The server may disregard the UNORDEREDBEGIN operation by choosing to perform all the succeeding operations in order. The status response to UNORDEREDBEGIN must be NFS4_OK or NFS4_UNORDERED_INVAL in all cases. Status and responses for all the unordered operations are returned in the sequence in which they appear in the compound request. The server may return failed statuses for one or more of the operations in the unordered section. In the case where some of the unordered operations have failed, the server may return successful statuses, including read data, for operations after the first failed operation, including operations after the last failed operation. The server is not required to return status for any unordered operation beyond the first operation in the unordered sequence to fail. In the case where one or more of the unordered operations have failed, then the returned status of UNORDEREDEND must be NFS4ERR_UNORDERED_FAILED. The status of the entire compound must be NFS4ERR_UNORDERED_FAILED if the server sends a response to the UNORDEREDEND operation. If the server truncates the response at an earlier failed operation, then the status of that operation must be returned as the status of the compound.

An unordered sequence cannot be implicitly terminated by the end of a compound. Each UNORDEREDBEGIN should be paired with an UNORDEREDEND, and these op-

erations cannot be nested. Improper matching of UNORDEREDBEGIN with UNORDEREDEND results in the error NFS4_UNORDERED_INVAL being returned for the first improper UNORDEREDBEGIN operation in the compound. For example, if the compound contains two consecutive UNORDEREDBEGIN operations, the first will return NFS4OK and the second will return NFS4_UNORDERED_INVAL because of the improper nesting. Similarly the result of the second of two consecutive UNORDEREDEND operations will always be NFS4_UNORDERED_INVAL.

The results of overlapping combinations of reads and writes in the unordered section are nondeterministic; the operations can be performed in any order on the server. This is not an error condition, and the server is not required to check for overlap conflicts among the unordered operations. Operations other than reads and writes may also be performed in an unordered section, including non-idempotent operations such as CREATE and OPEN. Interspersed PUTFH operations are interpreted in the sequence of operation order in the unordered section. If the server cannot unambiguously perform an unordered sequence of operations, the server can choose to perform the operations in sequence. The server may also return the status NFS4ERR_UNORDERED_INVAL for any of the unordered operations. In that case, it will also return NFS4ERR_UNORDERED_INVAL for the UNORDEREDEND operation, as well as for the entire compound.

Data Locations

Data locations is an extension of file system locations. The goal of data locations is to allow transparent (to the application) distribution and migration of the data portion of individual files and directories that reside in the same filesystem to multiple different servers, including files in the same directory. This allows subsequent read and write access to those files and directories to be made directly to the servers that contain them.

The approach we propose is that all the servers present the same fsid for the entire distributed file system. We extend the existing migration mechanism of NFSv4. To do this, we leave a pointer to the remotely located file in the form of a small *metafile*. The metafile is named, and resides in the regular file system namespace, possibly adjacent to regular files. It.is accessible by a filehandle returned via GETFH. The response to read and write operations on the metafile will return NFSV4ERR_DATAMOVED. This prompts the client to perform a GETATTR on the filehandle, asking for the data_locations attribute. The data_locations attribute will return the address information for the server that contains the data portion of the file, along with the filehandle of the data portion of the file. We call the data portion of the file a “data fork”. Data_locations is similar in format to the fs_locations attribute, with the root path replaced by a filehandle field for the relocated data portion of the file. One way to think of this is that just the data portion of the file has migrated. The file data is directly accessible at the specified server using the filehandle specified in the data_locations attribute, without a further LOOKUP or GETFH operation. The client will need to PUTFH on the new server to set the current file handle there to be the filehandle of the data fork.

The metafile has limited capability, and is primarily a redirection point for the relocated data fork. File relocations cannot be chained. If the data fork is subsequently relocated from the server pointed to, the metafile should be updated. Exactly one metafile should be maintained for each migrated file. Hard links are allowed to the metafile, but multiple metafiles cannot point to the same data fork. It is the responsibility of the server to maintain a coherent image of the entire file system, and to clean up all portions of files that have become completely unlinked.

The client opens the data fork using the DISTRIBUTED variant of OPEN. This is similar to the CLAIM_PREVIOUS variant of OPEN, with CURRENT_FH being the file-handle of the data fork.

The client can request that the server distribute individual files by creating files using the CREATE operation with a unique value of the createtype4 switched union, which is: **NF4DFILE void.** The attributes supplied with the CREATE operation must specify a data_locations attribute with one entry. The entry value specified should be null. The server is then responsible for deciding which data server will hold the data fork of the file. The metafile server fills in the data_locations attribute for the metafile. Similarly, the client can request that the server distribute an existing regular file by attempting to set the data_locations attribute with a one element list, with a null value for the single entry in the list. Thus the data_locations attribute is not writable, but can be set by the client in SETATTR and CREATE to request changes on the server.

The server may refuse to relocate the data fork of a file. It indicates that the data is local by setting a null value (a zero length list) for the data_locations attribute of a file. The data_locations attribute is null for all files that do not have relocated data.

The data_locations attribute's data type is data_locations4, which is defined as:

```
struct data_location4 {
        utf8str_cis     server<>;
        nfs_fs4         fh;
};
struct data_locations4 {
        data_location4      locations<>;
};
```

To facilitate the server's ability to ensure that the data forks are only opened by clients that have performed the required access rights negotiation at the metafile, the server may provide data fork filehandles that have a special form of volatility. These are single use file handles, specified with the flag RH4_VOLATILE_SINGLE_USE. The semantic of this file handle is that it is only valid to be used by the client for a single OPEN operation. After this, further attempts to open the data fork with the same handle will return NFS4ERR_BADHANDLE. The single use file handle will expire after the lease period if it is not used in an OPEN operation. Once opened, the file handle remains valid for use by that client only, until the client closes the data fork.

A server that issues only single use volatile file handles for data forks of files has the benefit of knowing when there are no file handles outstanding for a data fork. When no valid file handles are outstanding, the server is free to migrate the file's data fork.

Distribution of Parallel Files

We can extend the same mechanism used to relocate the data portion of files to specify multiple data forks for a single file. The client discovers the data forks in the response to a GETATTR call that the client issues after receiving the error NFSV4ERR_DATAMOVED from the server. The data_locations attribute will provide a list of servers and file handles of multiple data forks of the file. Each of these file handles corresponds to a separate, zero-based byte addressable data fork of the file.

These may be stored on the same or separate servers, all containing file systems with the same fsid. The data forks are implicitly counted and numbered by the length of the data_locations attribute list, and the position of the forks in the list. This allows full flexibility of application parallelism independently of the actual number of servers available to store the data forks: multiple data forks can be stored on the same server since they each have their own file handle.

The NFSv4 client MAY expose the data forks of the file as multiple separate byte addressible data streams of the same file, if there is application or VFS support for that. However, the typical usage of the data forks is to provide containers across which the client can stripe a single byte stream. This preserves the traditional application view of the file as a single byte stream while distributing data across multiple servers. To make the striping arithmetic simpler, the data forks are each sparse files. The position of each byte of data in one of the data forks is exactly the position of the same byte in the merged file. The merged file is simply the overlay of all the data forks. This reduces the byte addressing arithmetic needed to be performed on the client, and it allows a variety of data distribution, including distributions that are not simple striping. Since most server file systems can support sparse files at block granularity without excess space consumption and since the server can choose the distribution granularity, this simplification does not carry a space or performance penalty.

The client determines the distribution of the data among the data forks of the file by retrieving the data_distribution attribute stored in the metafile. This attribute must be present if the file has a non-null data_locations attribute of length greater than one. Data_distribution is a variable length array of uint4s. The first element in this array stores a single uint4 striping factor which specifies the basic stripe unit size in bytes. The default striping is a rotating in-order pattern across all the data forks. However, if another striping pattern is used, it is specified in the trailing uint4s in the data_distribution attribute. Each of these specifies the data fork that holds the next stripe units worth of data, in a zero based numbering scheme. The pattern of data placement specified in the data_distribution attribute is repeated to the end of the data fork files.

For example, a file that is striped across four data forks, with a stripe unit of 4k could have a data_distribution attribute containing a single uint4 with value {4096}. It could also have an array of five uint4s, with values {4096, 0, 1, 2, 3} to achieve the same effect; each successive 4k block would be placed in the next data fork, in a rotating pattern. A data_distribution value of {4096, 0, 1, 2, 3, 3, 2, 1, 0} would result in a data placement pattern that repeats every 8 stripe units, and that zigzags across the data containers. The usage of the data_distribution attribute is to allow servers to redistribute file data, for example by adding a data fork to a file, without having to redistribute the entire file.

Access to data forks is just like access to any other file, given the file handle. Unordered I/O operations can be performed against a data fork.

The data forks of a file can be independently secured. Therefore, a SECINFO negotiation may take place before a data fork can be accessed. Data forks may by default have universal ownership and ACL attributes that go unchecked by the server. This allows the server to perform all access right checking at the metafile when the metafile is opened. The server must check access rights to the data forks if the data forks have non-universal ownership or ACLs. The ACLs are modified by SETATTR calls. The server can determine whether the initial ACL and owner attributes should be the same as those of the metafile, or should be universal.

Locks can be mediated at the data forks. Locks can also be acquired on the metafile, or on byte ranges of the metafile. Mandatory locks on byte ranges of the metafile

should be enforced at the affected data forks. Locks made directly on data forks do not propagate to other data forks.

The striping or other distribution pattern of data among the data forks can be placed in the hands of the application. This requires VFS extensions or user space implementations that allow the application to explicitly specify a data fork to be accessed by an I/O operation. An intermediate I/O library, such as MPI-IO can make good use of the multiple data forks of a single file. Imposing an arbitrary striping on the data forks to construct a single stream of addressable byte space out of the multiple forks would force MPI-IO to first deconstruct the file into its parallel portions, then map its desired data decomposition onto those forks. In some cases, it is more efficient to provide the application library with an explicitly parallel interface to the parallel file, exposing the individual data forks for what they are without an unnecessary additional virtualization. Whether any operating system or client chooses to expose the individual data forks, through the VFS interface or otherwise, is optional and is beyond the scope of the protocol specification.

CREATE is extended to specify the number of data forks to create for a parallel file. The client can ask the server to distribute a file using the CREATE operation. Distributed files have a unique value of the createtype4 switched union: NF4DFILE void; The attributes supplied with the CREATE operation must specify a data_locations attribute with the desired number of entries in the list. The entry values should all be null. The server is then responsible for deciding which server will hold each of the data forks of the file, and will fill in the data_locations attribute for the metafile with a server location and filehandle for each data fork. The server can reduce or increase the number of data forks in the file, and retains ultimate control over data distribution and placement.

The client can request that additional data forks be added to an existing parallel file by increasing the length of the data_locations attribute in a SETATTR call. The server will ignore all the entry values, but will respect the length of the list. SETATTR can also be used to request the server to reduce the number of data forks in an existing parallel file, by sending a shortened data_locations attribute list. The server, if it complies with the request, will ignore the attribute values and reduce the list length from the end of the list forward. The client must retrieve the data_locations and data_distribution attributes after making a change to data distribution. The server should fail all I/O requests to the redistributed file from any client that has not refreshed its data_locations and data_distribution attributes.

Server-To-Server Operations

This proposal implies that some server-to-server operations be performed, to create and remove data forks of files, to set and get attributes of the data forks, to propagate and enforce locks, and most likely for several other operations. The client performs operations that affect both the metafile and its data forks by accessing the metafile only. Thus, we require a capability for all the servers to operate on each other, acting as privileged clients. There are a number of issues that must be addressed here, including security, and what the server-to-server protocol will be.

We believe that the server-to-server protocol should not be part of the NFSv4 specification. It may be most desirable to separately define a server-to-server protocol, similar to NFSv4, but that is designed specifically for the purpose of communication among servers. This allows the likely small community of parallel server builders to build interoperable servers, for example, to build separate data and metadata servers. The community can develop iterations of the protocol without changing NFSv4, allowing more flexibility and more rapid development. It is also possible that the server-to-

server operations can be performed using vendor specific proprietary protocols. Server vendors will be free to conform to a separate server-to-server specification or to implement their own proprietary server-to-server protocols, and can conform to the NFSv4.1 specification in either case. Thus, server-to-server operations, while implied by these protocol extensions, are outside the scope of the client/server protocol, and therefore are not specified in the set of protocol extensions. We are not certain of the need for a server-to-server protocol specification, but would be interested in discussing it further with other server vendors and developers.

Conclusions

The goal of this proposal is to specify how NFSv4 can be simply extended to provide support for distributing file systems, directory contents, and parallel files across multiple servers. It appears that this can be done with a minimum of protocol extensions. We also propose a simple unordered extension to NFSv4 that allows multiple I/O or other operations of the same compound to be performed concurrently at the server.

By leaving much of the semantics of file operations intact, in some cases reducing their scope to a single data fork of the file, we minimize the impact on the NFSv4 protocol, and reduce the amount of differentiated code needed in the V4 client to implement these extensions. It is our belief that the extension proposed could be implemented relatively easily by client and server vendors. We also believe that the extensions map quite well onto the distributed data architectures already present or under development in some servers.

Acknowledgements

Dave Noveck, Tom Talpey, and Brian Pawlowski all provided valuable input and feedback on the proposal.