Elements of a Scalable Network File System Protocol

Position Paper

Stephen Fridella, Xiaoye Jiang, and David Black
EMC Corporation
{sfridell, xjiang, Balck_David}@emc.com

1. Introduction

Standardized network file system protocols (e.g. NFS[4] [5], CIFS[6]) have been highly successful in enabling data to be shared among multiple clients and across heterogeneous computing platforms. The key to this success is that the protocols present data objects at a level of abstraction at which there is a broad common view among the multiple competing computing platforms: files and directories make sense in many operating systems. The flip side of this success is that by strictly enforcing this abstract view, the network file server becomes a road block to effectively scaling the level of shared access to the data. Even when the data reside on a shared storage medium (e.g. a high-performance SAN), only the file server knows how to map an element in the abstract namespace to a series of locations on the storage fabric. The protocol thus requires that the server be involved in every I/O operation of every client which needs to use the data.

The solution is for the network file system protocol to present data at multiple levels of abstraction, allowing clients to choose the level that is best suited to their needs. Some clients may choose to continue using the high level view of files and directories, but other clients may choose to view files as, for instance, lists of physical disk blocks. These lower level abstractions allow the clients that use them to assume, in parallel, I/O tasks that previously could only be performed by the network file server---thus eliminating the single-server bottleneck.

In recent years, several companies have leveraged this idea to build new parallel storage architectures (e.g. IBM's StorageTank [2], Panasas' ActiveScale [3], EMC's HighRoad [1]). Our own experience in designing and building one such product has led us to believe that a successful scalable network file system protocol should:

- Extend existing, proven network file system protocols (e.g. NFSv4, CIFS).
- Provide clients with a meta-data management interface for each file.
- Allow for an *extensible* notion of a block address space (or logical volume).

We will expand on each of these points in the following sections.

2. Extending NFS

Once the fact that existing network file system protocols are not sufficiently scalable has been recognized, and an additional meta-data management interface is required, it is logical to ask the question: should this problem be solved by developing a completely new protocol, or by designing a modular extension of existing protocols? The latter approach provides several advantages:

- Building a new protocol from scratch requires settling on a particular file system semantics. For instance, should the protocol provide a local-file-system semantics, or is it sufficient to support a more relaxed data consistency requirement? Perhaps the protocol should support both and leave the choice of semantics to the user? Such questions add unneeded complexity to the task at hand. By designing a protocol extension, we remain neutral on the question of file-system semantics.
- The extension approach allows us to leverage useful features of existing, stable protocols. For instance, NFSv4 has a lease-based mechanism for maintaining a client's lock state; it includes a callback mechanism of the kind we will require for breaking client block range locks (the callback path for read and write delegations); finally, it includes a sophisticated and flexible security framework. Re-using these established protocol elements makes the protocol design task simpler, and speeds the implementation of the new protocol extension.
- Having an existing network file system protocol available provides a simple fall-back solution for classes of files which do not derive much benefit from the parallel I/O approach (e.g. small files and directories), or in the case that a client's connection to the data storage fabric fails.

It was for these reasons that we chose the protocol extension approach for HighRoad, and for these reasons we prefer the extension approach in general.

3. A Meta-data Management Interface

To achieve superior scaling, the protocol extension must allow clients to read and write data to and from shared storage without requiring constant interaction with the file server. This requires each client to know the location of a file's data blocks on the shared storage; that is, the client must have access to a file's meta-data. A key feature of the protocol extension should be to define such an extended interface. This interface will certainly include:

- An operation to query the file's block mapping---that is, for each logical block of the file, the identity of the logical volume containing the block, and the location of the block within that logical volume's address space. This allows a client to read data directly from the storage fabric.
- An operation to allocate new blocks and (provisionally) add them to the file's block mapping. This allows a client to write data directly to the storage fabric.
- An operation to commit or release the provisionally allocated blocks. This allows a client to control when changes written by the client are visible to other clients.
- Mechanisms to maintain the consistency of file data when multiple clients are sharing access to a single file. One sufficient mechanism would be NFSv4's read and write delegations. Our view, however, is that for large files, it is useful to allow clients to read and write simultaneously to disjoint regions of the file, and thus we believe that it is better to provide block-level granularity delegations.
- A mechanism that enables consistent recovery from client or server failure. For instance, a lease mechanism similar to NFSv4's client lock-state lease.

At this point, a more concrete example of these proposed operations might be helpful. The following is a high-level description of the File Mapping Protocol, which we designed and implemented for the HighRoad project.

The client uses a *get_mapping* operation to query file block mappings. The client specifies a logically contiguous extent of file blocks as an argument. The server should return to the client, for each requested block, the identity of the logical volume containing the block, and the location of the block within the logical volume address space. Of course, if the file is sparse, then not every block of the file will map to an actual storage location. For such blocks, the server indicates a special state for the block, indicating that it is a file hole. In principle, the client could use the block mapping information to read or write data to the file blocks---but of course the client must first negotiate with the server for the proper delegation for each block. In our view it is simpler to combine the delegation negotiation with the mapping query. Thus the *get_mapping* operation succeeds only if the server can grant a *read* delegation for each requested block. If a conflict occurs, the server can tell the requesting client to retry the request later, and in the meantime, attempt to revoke the conflicting delegations held by other clients.

In order to write to a file, the client sends an *allocate_block* request to the server. Again, an extent of file blocks is the argument. For this operation, the server will attempt to grant a *write* delegation for each requested block. If the server succeeds then it processes each block in the requested extent. For those blocks that already exist in the file mapping, the server simply returns the block mapping information as above. But for those blocks which do not exist---either because they are file holes, or because they are beyond the current end of the file---the server provisionally allocates locations on the logical volume to store these new blocks, and it is these locations that it returns to the client.

Once the client has written data to the new locations, it can cause them to become permanently part of the file by sending the server a *commit_block* request. This request asks the server to update the file's meta-data to reflect that the new blocks are now a part of the file. If the client chooses not to write to the blocks after all, or simply decides to release the delegation on a particular set of blocks (perhaps at the request of the server), it can send a *release_block* request to the server.

As in NFSv4, a client's delegations are controlled by a lease, granted by the server. If the lease is not renewed within a certain time period, then the server is free to grant delegations for the blocks in question to other clients.

4. Logical Volume Discovery

A key requirement for making such a protocol work, is that the client and server must agree on the identities of the logical volumes which contain the file data. We have been purposely vague on the question of what, precisely, do we mean by the term *logical volume*? As we stated above, our view is that this should remain an extensible notion. For us, a logical volume is simply a shared data object which provides a block address space. Whether this object is realized by a physical disk, a hyper volume striped across multiple disks, or any other kind of storage object should not matter---as long as the client and server can agree on how to identify and access these objects.

Once a client has established contact with a server, before it starts performing I/O, it sends a *volume-discovery* request to the server. The client should provide two arguments as part of this request: the namespace (file-system) it wishes to use, and the *discovery-flavor* it is capable of

supporting. A discovery-flavor is simply a reserved value indicating the choice of protocol by which the server and client will agree to identify shared storage objects. For instance, a simple discovery-flavor might be termed *physical-discovery*. Under this choice of protocol, the server provides a mapping between the disk signature that the server has written on each physical device of a shared SAN, and a logical volume id. The client can then query the SAN's physical devices for their signatures, and match the results with its own local path information (chain, target, lun). As we stated above, within each block mapping returned by the server to the client is a logical volume identifier. Thus the client can use this identifier to determine the correct path to the file's data. It is worth repeating that logical volume identifier need not map to physical disks. For example, it is equally possible to define an alternative discovery flavor that maps logical volume ids to an Object Based Storage (OSD)[7] device and object identifier.

The discovery-flavor is meant to allow new discovery methods to be added to the protocol quickly and easily by server and client vendors alike. We can imagine that rather than a client specifying a single discovery flavor that it will use, it could instead negotiate with the server to find the "best" flavor supported by both parties. Generally speaking, protocol efficiency improves as the client takes on more of the volume management tasks. When the server can specify block mappings for a flat, abstract logical address space, the mappings contain fewer non-contiguous extents, and therefore require fewer bits of information to specify. Since the client takes on the processing task of mapping these logical locations to their underlying physical storage objects, load is further decreased on the server. Because the discovery flavor is an extensible notion, our protocol easily accommodates new logical volume management technologies.

5. Summary

New mechanisms are needed to break the single server bottleneck for network file system protocols. In order to allow clients to take on more of the I/O processing tasks that were previously only performed by the server, network file system protocols must allow clients to look past the high level abstractions that have traditionally defined such protocols. In this position paper, we have advocated an extension to standard network file system protocols (such as NFSv4) which defines a meta-data file interface as well as an extensible notion of a logical volume. Clients can use the protocol to achieve direct, safe, shared read/write access to file data residing on shared storage. The file server is only involved in serving and updating file meta-data, not file data, and thus is no longer a bottleneck limiting the scaling of data sharing. This protocol extension leverages useful features of existing network file system protocols to provide consistency mechanisms, high-availability guarantees, and recovery. Finally, by adopting an extensible notion of logical volumes, the protocol extension remains adaptable in the presence of evolving paradigms for high-performance shared storage.

6. References

[1] EMC white paper, *EMC Celerra HighRoad*, http://www.emc.com/pdf/products/celerra_file_server/HighRoad_wp.pdf

[2] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM Storage Tank---a heterogeneous scalable SAN file system", *IBM Systems Journal*, 42:2, p. 250.

[3] Panasas white paper, *Shared Storage Cluster Computing*, http://www.panasas.com/docs/Shared_Storage_WP.pdf

[4] RFC 1813, *NFS Version 3 Protocol*, http://www.ietf.org/rfc/rfc1813.txt

[5] RFC 3530, *NFS Version 4 Protocol*, http://www.ietf.org/rfc/rfc3530.txt

[6] SNIA CIFS Technical Work Group, *Common Internet File System (CIFS) Technical Reference*, http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf

[7] SNIA OSD Working Group, *SCSI Object Based Storage Device Commands*, ftp://ftp.t10.org/t10/drafts/osd/osd-r05.pdf