Standard Scalable Asynchronous Access to Data

NEPS Position Paper, 4 December 2003
Craig Everhart, IBM

We strive to build a client-server file system with many of the desirable properties of local and cluster file systems, which we see as more than simply high performance for parallel access. We want a file system with good performance for small as well as large files, and we want the same kind of coherency guarantees that exist in local file systems. We want to be able to use existing, unmodified storage systems, and we want clients on a variety of host operating systems, with excellent support for Windows. We believe that NFS version 4 is a good basis for building this file system.

Many client-server file systems have been built in which the client accesses the file data in the same way that it accesses file metadata (attributes and directory names) by consulting a file server. File systems have more recently been built to exploit a storage fabric (SAN) shared not only between server and storage but also with the client, separating the client's data access from its metadata access. This allows clients to access data *asynchronously*, using metadata provided by the server, rather than requiring the server to access storage on behalf of clients. The first such file systems included EMC's HighRoad and IBM's SANergy. These systems generally provided for high performance to large files, but with limitations imposed by the protocol atop which they were built. Later systems, including IBM's SAN File System, Panasas' ActiveScale, and Lustre, replace the base protocol and can thus provide better coherency guarantees. It is this second breed of file systems, with greater cache control, around which we particularly wish to extend the base NFS protocol.

The delegation facility in NFSv4 allows it to be a reasonable and attractive substrate for this work. Loosely, the model would be that the server provides the metadata service as usual, but that the client has the ability to read and write files not only by using the NFSv4 READ and WRITE operations, but also by learning what it needs from the file server so that it can do I/O over the SAN if it is connected appropriately. Since, in this architecture, the data access is performed asynchronously from server operations, there needs to be synchronization between the data access and such server operations that can rearrange the data storage, such as file truncation. The NFSv4 delegation mechanism is roughly suitable for this synchronization: the client is to perform its asynchronous data access only while it is holding an appropriate delegation, and the server is to recall the client's delegation before invalidating the SAN access information.

SAN access to block storage is not the only kind of asynchronous data access that clients might wish to perform. For example, if object disks are being used for data storage, the server could give the client the appropriate addressing information for that variety of storage—e.g., an object identifier and a capability. Alternatively, a subsidiary NFS server might be set up to house only data, so that the main server would tell the client what handle on which subsidiary NFS server is to be used to access data. Additionally, file data could be striped across sets of block devices, sets of object devices, or sets of subsidiary NFS servers. Furthermore, the protocol should support the use of multiple kinds of storage devices by a single client. In all these cases, though, synchronization of the access to the object is mediated by the primary file server, which then gives to clients the information they need in order to read and write file data, under delegation from the primary file server.

It may be a good idea to make specific versions of the existing idea of NFSv4 delegations to cover asynchronous access capability. The existing READ and WRITE delegation types are not specified with such access in mind. It may be more appropriate to define READ_ASYNCH and WRITE_ASYNCH variants that provide the guarantees of the existing delegation types, plus the guarantee that the access metadata will not change.

A new DIRECT delegation would be useful to describe a shared-write environment, in which multiple clients can both read and write asynchronously, without caching data between application and data storage. Multiple clients can hold the DIRECT delegation simultaneously. It is incompatible with both READ and WRITE delegations, since those delegations permit cached data. This delegation is useful for applications that use a shared-data model between multiple instances. Operations that change the block map in any way, e.g. by changing the file size, need to revoke the DIRECT delegation.

Additional Extensions

Our position is that NFS needs some key improvements that will allow it to scale to larger environments:

- coherency promises on directories with directory delegation (read delegations; write delegations not needed);
- the ability to cache byte-range locks on clients without a write delegation, meaning that they can be recalled and that the recall response can be YES or NO;
- the ability to cache share/deny modes on clients without a delegation, also meaning that they can be recalled and that the recall response can be YES or NO;
- a clear ability to regain a delegation once it has been recalled.

The current model for caching under delegations is loosely that clients are responsible for establishing all cached opens and locks with the server before releasing a delegation. In the explicit caching model, clients obtain opens and locks with the server but do not necessarily release them when the application does. The server, presented with a conflicting demand, needs to request that the client release the conflicting opens or locks. The client needs to be able to communicate that some opens and locks may be released, but that others are in use by current applications and therefore may not be released. The server might indicate that the requester intends to wait for one of these resources, and therefore would the client please relinquish it promptly once the application is finished. At the same time, the requestor (a second client) needs to be told promptly that the resource is available. These server-to-client communications can be done with the callback interface, but appropriate callback requests need to be defined.

Directory change notification should be supported in some manner in order to enrich support for Windows. Other such enrichments would include support for the DELETE operation in share/deny modes and the idea of a reparse point (to this level of abstraction, an uninterpreted blob of bytes associated with a new file type).

Data Access for Block Devices

For data stored on block devices, we would look for some generality in the mechanism used to describe the location and validity of this data. The generality we seek includes a way to describe the logical unit and physical address of successive extents in a file, allows allocation holes in the file, and provides a translation mechanism to describe striping. We prefer a mechanism in which the client can be responsible for the initialization of freshly-allocated storage, and we suggest a "valid" annotation on each block descriptor to capture this information. In addition, we expect copy-on-write processing to be done by the client, so that for any given address in a file, there might be block descriptors for both read and write operations. To elaborate on this last point, the representation of file data on block devices should include block addresses for both reading and writing, so that if the client is accessing copy-on-write data, it can read the old data and write new data at the same logical address within a file.

One realization of these requirements is a regime in which block addresses are annotated with a notion of whether the data to which they refer is *valid*, and in which separate Read and Write maps exist in principle for a file. When a client writes a new file, the server allocates new *invalid* blocks to the Write map for the file, and it gives their addresses to the client. Once the client writes data to the blocks and zeroes out unused partial blocks, it tells the server which blocks should now be considered *valid* as part of committing the write. No blocks have yet been allocated for the Read map. If a copy-on-write image is now made of the file, the blocks that had been allocated and marked *valid* in the Write map are now moved to the Read map for the file, and the Write map is made *invalid*. If a client now reads the copy-on-write file, it reads data indicated by *valid* addresses in the Read map and treats the file as having holes otherwise. If the client wishes to write to a copy-on-write extent, it requests the server to allocate blocks in the Write map, which will initially be marked *invalid*. It is the client's responsibility to copy partial data from the blocks described in the Read map, update it according to the application writes, and store the data into the blocks described in the Write map, then committing the write by telling the server to mark the blocks *valid*. Thus, in this realization, blocks in the Read map may be in either of the two states "unallocated, invalid" and "allocated, valid". Blocks in the Write map may be in any of the three states "unallocated, invalid", "allocated, invalid", and "allocated, valid". The "unallocated, invalid" state represents a hole in the block map. The "allocated, invalid" state represents an allocated, unwritten block. The "allocated, valid" state represents an allocated, written block suitable for reading.