Standardizing Storage Clusters

II pNFS become the new standard for parallel data access?

GARTH GOODSON, SAI SUSARLA, and RAHUL IYER, NETWORK APPLIANCE

Data-intensive applications such as data mining, movie animation, oil and gas exploration, and weather modeling generate and process huge amounts of data. File-data access throughput is critical for good performance. To scale well, these HPC (high-performance computing) applications distribute their computation among numerous client machines. HPC clusters can range from hundreds to thousands of clients with aggregate I/O demands ranging into the tens of gigabytes per second.

To simplify management, data is typically hosted on a networked storage service and accessed via network protocols such as NFS (Network File System) and CIFS (Common Internet File System). For scalability, the storage service is often distributed among multiple nodes to leverage their aggregate compute, network, and I/O capacity. Traditional network file protocols, however, restrict clients to access all files in a file system through a single server node. This prevents a storage service from delivering its aggregate capacity to clients on a per-file basis and limits scalability. To circumvent the single-server bottleneck of traditional network file system protocols, designers of clustered file services are faced with three choices (these are illustrated in figure 1):

- To force administrators to partition the file namespace manually among servers via multiple mount points.
- To hide the data distribution from clients by employing a transparent request routing layer.
- To devise a new protocol that allows clients to directly access multiple servers in parallel.

The first approach imposes the burden and expense of manual data distribution on system administrators. It is error-prone, reduces availability, and quickly becomes unmanageable as data grows in size. Moreover, it cannot spread large files over multiple servers without application-level changes.

The second approach allows existing unmodified clients to access distributed storage and hence is simple to deploy and maintain on large client farms. It limits end-to-end scalability, however, by forcing a client's data always to flow through a single entry point.

The third approach eliminates this bottleneck and enables true data parallelism. As such, it has been adopted by several clustered storage solutions. Because of the lack of a standard protocol for parallel data access, however, the protocols and interfaces remain proprietary.

Although custom client access protocols provide the best performance and scalability, they have limitations: they inhibit interoperability across diverse client platforms and storage architectures; they also make it difficult to develop and maintain client software for the heterogeneous platforms that must operate in large compute farms over extended periods of time; and finally, clients using inflexible interfaces cannot be evolved rapidly to benefit from advances in distributed storage architectures and require constant maintenance. The lack of a standard parallel data access protocol remains a key hurdle to the widespread adoption of clustered storage for mission-critical HPC applications.

The pNFS (parallel NFS) protocol is being standardized as part of the NFSv4.1 specification to bridge the gap between current NFS protocols (versions 2, 3, and 4) and parallel cluster file system interfaces.

Current NFS protocols force clients to access all files on a given file-system volume from a single server node, which can become a bottleneck for scalable performance. As a standardized extension to NFSv4, however, pNFS provides clients with scalable end-to-end performance and the flexibility to interoperate with a variety of clustered storage service architectures.

The pNFS protocol enables clients to directly access file data spread over multiple storage servers in parallel. As a result, each client can leverage the full aggregate bandwidth of a clustered storage service at the granularity of an individual file. A standard protocol also improves manageability of storage client software and allows for interoperability across heterogeneous storage nodes. Finally, the pNFS protocol is backward-compatible with the base NFSv4 protocol. This allows interoperability between old and new clients and servers.

Using the pNFS protocol, clients gather metadata, called layouts, about how files are distributed across data servers. Layouts are maintained internally by the pNFS server. Once the client understands the file's layout, it is able to directly access the data servers in parallel. Unlike NFSv4 whereby a client accesses data via the NFS protocol from a single NFS server, a pNFS client communicates with the data servers using a variety of storage access protocols, including NFSv4 and iSCSI/Fibre Channel using the SCSI block command set or the new SCSI object command set. The pNFS specification allows for the addition of new layout distributions and storage access protocols. It also provides significant flexibility in the implementation of the back-end storage system.

Design principles and General architecture

The design of pNFS follows three main principles:

- **Familiar semantics.** The pNFS protocol must provide consistency and security semantics similar to the base NFSv4 protocol. This simplifies the adoption of pNFS by the diverse set of existing NFS applications.
- **Simplicity.** The pNFS extension must be kept simple, yet efficient. The pNFS interface must provide the most basic primitives needed to exploit data parallelism efficiently, while allowing the layering above of additional, more complex semantics.
- **Flexibility.** The pNFS protocol must be flexible enough to accommodate a variety of storage architectures. These architectures include a federation of stand-alone file servers, a mix of a NAS (network-attached storage) file server and RAID storage devices, and a fully transparent parallel clustered file system.

Architecturally, the components include one metadata server, some number of data servers, and some number of NFS clients. Figure 2 provides an overview of the components of a pNFS system.

The pNFS protocol logically separates metadata control operations from data accesses. For each file, the metadata server maintains a layout that encapsulates the location, striping parameters, and any other tokens required to identify that file on the remote data servers. Additionally, the metadata server stores all directory metadata and file attributes related to the striped files, while data servers respond only to I/O requests. On the other hand, file data is striped, typically in a round-robin fashion, across a set of data servers.

At a high level, a client accesses data as follows:

1. It opens a file and obtains the file's layout from the metadata server; the layout describes where the file's data is located.
2. The client then performs I/O in parallel to the data servers described by the layout.
3. Once the client completes its I/O, it commits modified metadata and closes the file at the metadata server.

Note that figure 2 represents only an abstract view of the pNFS architecture. In practice, an actual implementation may have components that assume multiple roles (e.g., a single node acting as both a metadata server and a data server). Furthermore, it is possible to split the architecture between a front end that exports pNFS through the NFSv4.1 protocol and a back end that supports synchronization between the metadata and data servers (e.g., through some type of cluster file system).

As shown in figure 2, the pNFS protocol works in conjunction with two others: a storage-access protocol

and a control protocol. The pNFS protocol operates between the client(s) and the metadata server and can be viewed as NFSv4 with a few additional operations for clients to query layouts and obtain data server locations. The storage-access protocol dictates how clients access data directly from the data servers after they have obtained the layout information (via NFS or SCSI). The control protocol is used by the storage system to synchronize state between the metadata server and data servers. This protocol is deliberately left unspecified to provide flexibility to server implementations by making the back end of the storage system free to choose when and how to synchronize file metadata between the metadata and data servers.

Layout types

A pNFS layout contains enough information for clients to determine where each stripe of a file is stored, how to access it directly, and the storage-access protocol to use. The layout delegates to the client the ability to do parallel I/O directly to the data servers. The metadata server can provide the client with a layout that applies to an entire file or to a specific byte range therein. Clients can cache layouts for performance. To keep the layout held by a client up-to-date, the metadata server must explicitly recall it before making any changes to the file's distribution (e.g., before restriping or migrating a file).

The pNFS protocol is designed to allow for the addition of new layout types and their associated storage-access protocols, which are determined by the interface exported by the data server. Three distinct layout types are currently being specified: file, object, and block. They correspond to different ways of storing file data on data servers and require different storage-access protocols to access the data. Each of these three layout types uses a different storage-access protocol.

The file layout is used when a pNFS file's data is striped across multiple NFSv4 file servers and uses NFSv4 as the storage-access protocol. Clients must use the NFSv4.1 read and write operations to communicate with data servers when using the file layout, which contains a list of data servers over which a file is striped. For each data server listed, the file layout contains the size of each stripe and the NFS file handles to be used. Since the file layout is compact and does not change with updates to the file, a file layout can be cached by many clients without incurring synchronization overheads even when the file is widely shared. The file-layout specification is the result of a joint collaboration between Network Appliance, Sun, IBM, and others.

In contrast, block layouts export LUNs (logical unit numbers) hosted on SAN (storage area network) devices, such as disks or RAID arrays. Block layouts use iSCSI or Fibre Channel employing the SCSI block command set to access SAN devices. Through the use of block extents, a block layout explicitly lists the physical storage blocks for each portion of the file. Thus, the layout grows as file size increases, and it needs to be updated as file blocks are allocated and freed by the metadata server. The block layout specification is heavily influenced by EMC's NAS front end called HighRoad (also known as MPFS, for Multi-path File System).

Object layouts share similarities with file layouts. They use the SCSI object command set to access data

stored on OSDs (object-based storage devices). Similar to file layouts, they have a compact representation and use object IDs similarly to file handles. There are some significant differences, however. For example, object layouts contain a security capability required for accessing the data on the OSDs. Object layouts also contain richer, more complex parameters to describe striping patterns, whereas file layouts are designed to be lighter weight. The object layout specification is heavily influenced by PanFS, a commercialization of the NASD (network-attached secure disks) project from Carnegie Mellon University by Panasas.

For a client to locate its data, the name of the data server must be encoded within the layout. Since data servers are accessed via the storage-access protocol, their names are specific to the addressing scheme used by that protocol. Data-server names may be long (for example, block-based layouts use volume labels as device names, which may be several kilobytes in size). Hence, the pNFS protocol virtualizes device names by mapping them to a shorter, layout-independent data server ID. Layouts reference these data-server IDs.

protocol operations

The pNFS protocol adds a total of six operations to NFSv4. Four of them support layouts (i.e., getting, returning, recalling, and committing changes to file metadata). The two other operations aid in the naming of data servers (i.e., translating a data server ID into an address and getting a list of data servers). All the new operations are designed to be independent of the type of layouts and data-server names used. This is key to pNFS's ability to support diverse back-end storage architectures.

These are the steps involved in accessing a file distributed by pNFS:

1. The client must obtain a layout that describes to the client where to find the file's data and how to access it.
2. The client must obtain the proper access rights by opening the file at the metadata server.
3. The client can now perform I/O directly to the data servers using the storage-access protocol associated with the file's layout type.
4. If the client writes data to the data servers, it must commit changes to the file's metadata (including modification times and size) to the metadata server. For example, if the client writes to a block device, the metadata server must track how the written disk blocks map to data blocks within the file.
5. The client may close the file and return the layout once it has completed its I/O and committed any metadata changes.

To reduce the state maintained by both the metadata server and the client, a client may return a layout that it is no longer using. Similarly, the metadata server can explicitly recall a layout. In response to a recall, the client must return the layout after flushing any dirty data. Once returned, the client must not attempt to access the file's data via the data server directly. Instead it should either obtain a new layout or access the data via the metadata server by falling back to NFSv4-style I/O.

File-sharing semantics

One of the goals of pNFS is backward compatibility with NFSv4 in terms of file-sharing semantics for clients (i.e., to provide close-to-open consistency). Unlike some other distributed file systems, supporting Posix-style single-system semantics to clients (i.e., immediate visibility of remote writes) is not a goal for pNFS. Close-to-open semantics guarantee that a client will observe changes made to a file by other clients only after they close the file. These semantics are adequate for the common case where concurrent write-sharing of files is rare. When write-sharing does occur, pNFS clients can acquire NFSv4 byte-range locks through the metadata server to serialize concurrent updates to the same file to ensure that it can be made linear.

The pNFS protocol requires that clients notify the metadata server after new data updates have been made, as these updates may not yet be visible to other clients. This operation makes the updates visible through changes to the file size and time. To ensure close-to-open consistency, a client must issue this operation before closing the file if it has written data. This operation allows the back-end control protocol to be lazy in synchronizing data-server writes with file metadata updates for improved parallelism.

Since pNFS clients communicate directly with individual data servers, they cannot issue a single I/O operation that spans multiple data servers. Thus, clients are now responsible for ensuring the atomicity of writes that span multiple data servers (cross-data server writes). If clients require atomic cross-write functionality, they must implement it themselves or fall back to using regular NFSv4.

Access control and security

Possessing a layout does not by itself give the client access rights to the file's data. Instead, access checks are performed and access rights are obtained by existing NFSv4 operations (e.g., OPEN, LOCK). Similar to NFSv4, access checks should be performed on every I/O operation, but this is not always possible because of the storage-access protocol in use. For example, per-I/O access checks on block devices (e.g., Fibre-Channel disks) are not possible because of their more constrained interfaces. In such situations, the per-I/O access checking normally performed by the NFS server is moved into the client. Storage system architects and consumers must recognize these tradeoffs when deploying such systems.

The different layout types require very different security and authentication models. For example, the only security and authentication mechanisms that exist for block devices typically operate at the granularity of the data server or LUN (e.g., the fencing of devices on a SAN). Device-level granularity presents a problem when a system is faced with malfunctioning or malicious clients. Clients are no longer restricted to reading/writing only the files to which the file system permits them access, but rather can access and corrupt any file on any device to which they have access.

Even though the difference between file- and object-based implementations is small, one still exists. OSD (object-based storage device) security is based on secure capabilities that are generated by the

metadata server and validated by the data servers. These capabilities provide fine-grained access to devices and objects. File-based security and authentication is provided through the RPCSEC_GSS framework at the RPC (remote procedure call) level, and access control is enforced at the file level by the data servers through the use of ACLs (access-control lists) and file permissions, similar to NFSv4. Changes made to ACLs and other file permissions through operations to the metadata server (e.g., a CHMOD that generates an ACL change) must generate the appropriate changes to the data servers.

Caching layouts

Layouts for overlapping byte ranges of the same file may be cached by multiple clients simultaneously. Generally, the type of I/O for which the layout is being used (read or write) does not affect the layout's cacheability by multiple clients, since a layout simply describes where and how to access the data.

A layout can change during the lifetime of a pNFS file. The metadata server must recall all affected cached layouts before they are modified, and, moreover, clients are guaranteed that explicit recalls will be made before the layout is modified—for example, if a file is being striped across a new set of data servers. After a layout has been recalled, the client can always fall back to accessing data via the metadata server using the NFS protocol, as in regular NFSv4. Indeed, the ability to fall back to performing I/O through the metadata server using regular NFS accesses is useful in a number of error cases.

System Architecture Considerations

The separation of the data and control paths allows data accesses to proceed simultaneously, in parallel from multiple clients to multiple storage endpoints. It also simplifies the design and implementation of the pNFS protocol and allows for the use of non-NFS data servers, such as SAN devices. Although the pNFS architecture assumes that a single server is responsible for all metadata, it can also be applied to systems where metadata, in addition to data, is distributed among multiple servers.

Although limited to mounting only a single metadata server at a time, clients can be distributed across a set of metadata servers if each one exports the same global namespace. In such systems, clients can use the pNFS protocol to communicate with any of the available metadata servers to obtain file layouts. The pNFS semantics require the metadata servers to enforce coherent, location-independent access to file layouts. The metadata servers are free to use any internal protocol to locate file layouts and keep them consistent among themselves and with data servers, as long as they provide coherent layouts.

Some clustered file systems provide one or more front-end servers that export a coherent file namespace to clients and provide a path to storage through any front-end server. In such cases, all front-end servers are equivalent from a client's perspective: the client sees and is able to access the same file system regardless of the front-end server mounted. These implementations can dynamically load-balance client accesses across the set of data servers by altering the list of front-end servers within the layouts issued to clients. Since all data servers access common underlying storage and provide coherent data access,

clients can be given layouts for the same file that list different sets of data servers, thus allowing the load to be spread more evenly.

Client/Server Implementation

Implementing a pNFS client adds some amount of complexity to a regular NFSv4 client. To start with, NFSv4.1 has a number of additional required features; for example, the mandatory use of the Sessions protocol adds complexity to an already stateful NFSv4 client. In addition to the complexity added by NFSv4.1, pNFS requires a client to understand both the generic pNFS components, such as layouts and device mappings, as well as the layout-specific storage-access protocol used to access the data servers. Figure 3 shows an example of how a client might be structured to incorporate the additional layers required by pNFS.

As shown in figure 3, a client implementation may separate the generic pNFS layout caching and device management from the layer that implements layout-specific device access, implemented via layout drivers that interface with the appropriate subsystem for that device type. For example, the block-layout driver would require access to devices exported via the SCSI layer, whereas the file-layout driver would access files via Sun RPC-based NFS. The Linux pNFS client is in development with this split architecture.

When designing a pNFS server, the most important consideration is the layout type(s) that the system will support; after all, the layout type dictates the client's interface to the data servers and has a considerable impact on the design of the control protocol that runs between the metadata and data servers and the security/trust model. For example, implementations that support the block layout can easily use off-the-shelf block devices as data servers; however, the metadata server implementation is complicated by the requirement to track all data-block allocations—which, otherwise, can be offloaded to object- or file-based data servers. Also, as discussed previously, clients are responsible for enforcing fine-grained access control since block data servers are typically unable to do so.

File-layout server implementations can be constructed from a wide range of data servers of varying complexity. The simplest of implementations may simply stitch together a number of NFSv4.1 servers with a basic control protocol responsible for maintaining consistent attributes and state.

Similar to the client, the pNFS metadata server can use multilayered architecture, which may be especially useful when adapting an existing cluster file system back end to pNFS. A front-end layer provides the pNFS interface to the client, while the back end interfaces with the file system to be exported, typically a cluster file system. At a high level, this is similar to how current operating systems export local file systems via NFS.

A major shift

As data-intensive HPC applications grow in their demand for scalable storage capacity and bandwidth, new file system protocols that are able to provide direct, parallel data access from the client become necessary. Many of today's systems restrict file access from a single client through a single front-end server. This prevents it from leveraging the cluster's aggregate bandwidth on a per-file basis.

The pNFS protocol introduces a major shift to the NFS standard. It provides a standard interface for clients to leverage the aggregate I/O capacity of these systems more effectively by enabling them to issue I/Os in parallel to multiple servers. Although the separation of metadata and data increases client complexity, it also provides flexibility to storage architecture implementations.

While parallel file systems fill a niche market today, pNFS has the opportunity to standardize this interface and potentially commoditize parallel file systems.

GARTH GOODSON has been a member of the engineering technical staff at Network Appliance since 2004, where he has been involved in the IETF pNFS draft specification. He has also worked on a pNFS server prototype using file layouts. He has a Ph.D. from Carnegie Mellon University.

SAI SUSARLA has been a researcher at Network Appliance for three years. He has a Ph.D. in computer science from the University of Utah. His research interests include scalable distributed data access, automated data management, and system analytics.

RAHUL IYER is a researcher at Network Appliance, working on pNFS. He also collaborates in the development of the open source Linux NFSv4.1 client. Prior to joining Network Appliance, he received his master's degree from Carnegie Mellon University.