NFSv4 and High Performance File Systems: Positioning to Scale

Dean Hildebrand dhildebz@eecs.umich.edu
Peter Honeyman honey@citi.umich.edu
Center for Information Technology Integration
University of Michigan
Ann Arbor

1 Introduction

The *avant garde* of high performance computing is building petabyte and beyond storage systems [1–3]. I/O is quickly emerging as the main bottleneck limiting performance in these systems, making the need for scalable file access increasingly urgent.

Many of the techniques for improving scalability of a distributed file system, such as replication and client caching, are unsuitable. Many clients cannot cache a gigabyte of data and many servers lack the resources to replicate it. At CITI, we are engineering NFSv4 [4] for use as a universal standard for fast and secure access to data, whether across a WAN or within a (possibly massive) cluster. NFSv4 has tightly integrated mandatory security as well as support for aggressive client caching, anticipating efficient and secure WAN access. Support in NFSv4 for caching, locking, and delegation suggest the potential for superior performance both in a cluster and across a WAN.

An NFSv4 server manages much state information, which interferes with access to files (or portions of files) through multiple servers. The constraint of a single server becomes a bottleneck as load increases. In this white paper, we introduce Parallel NFSv4, which extends the NFSv4 protocol to support distributed state maintenance. This includes a new server-to-server protocol to manage the global state of the system and a new file description and location mechanism. The goal of our design is to enable access to data repositories for data collection and post-analysis with orders of magnitude improvements in capacity and bandwidth while enforcing and preserving consistent and secure shared access.

1.1 Runtime State in the NFSv4 Server

In addition to some basic information about clients, users, and files, an NFSv4 server keeps track of share reservations, byte-range locks, and delegations.

A share reservation is a mechanism to control access to a file, comparable to whole-file locking. When a client issues an `OPEN` request, it specifies the type of access required (read, write, or both) and the type of access to deny to others (deny none, read, write, or both). The server maintains this information to ensure that future OPEN requests do not conflict with the current share reservations. Each client determines the unit of share lock granularity–known to the client as the *open owner*–which may be a process id, an inode, a user credential, or any other grouping of client resources that access the files.

NFSv4 supports two styles of record locking: mandatory locks and advisory locks. Like share reservations, a lock owner is associated with each lock to identify the group of processes on a client accessing a file.

The server passes control of a file to the client, at the server's option, in response to an OPEN request. These delegations come in two flavors, read delegations and write delegations. To prevent inconsistent access, the NFSv4 server must remember all outstanding delegations on a file so that callbacks can be issued when needed.

1.2 Exported File System Models

The high performance community uses parallel and cluster file systems to access and store data. For NFSv4 to become successful in the HPC, it must be able to scale with these file systems.

Cluster file systems such as GPFS [5] provide a consistent view of a file system from all nodes. Scalability is in direct proportion to the number of nodes and disks in the system. By distributing servers over the nodes in the cluster file system, Parallel NFSv4 can utilize this scalability to overcome the bottleneck in the NFSv4 single server design.

Parallel file systems such as Lustre [6] provide clients with direct access to file system data. Replicated failover metadata servers maintain a transactional record of high-level file and file system changes. Data is striped across nodes that handle all of the interaction between client data requests and the underlying physical storage. Parallel NFSv4 extends the NFSv4 protocol also supports the parallel file system model, allowing direct access to its data.

2 Related Work

NFSv4 is a new protocol still in its initial implementation stage, so no scalability experiments have been published, but work based on NFSv3 has been reported. Juszczak [7] implemented *write gathering* on the server, which combined several metadata updates to the same file into a single disk operation. This helps reduce server CPU and disk usage.

Not Quite NFS [8] and Spritely NFS [9] added soft and hard state respectively, to the server to achieve full cache consistency. Spritely NFS influenced the design of NFSv4 delegations. While these techniques improve performance for a single client, this paper focuses on scaling the number of clients.

Extensions proposed by Peter Corbett and Dave Noveck from Network Appliance to the NFSv4 protocol intend to scale the number of clients in the system by enabling the NFSv4 server to stripe file data across multiple servers, transforming NFSv4 into a parallel file system. These extensions do not improve the read and write throughput for underlying cluster and parallel file systems, which is the primary goal of this paper.

3 Design

The design goals of Parallel NFSv4 are:

- No impact on the NFSv4 security model.
- Minimal impact on NFSv4 fault tolerance semantics.
- Minimal increase in network traffic.
- Extensibility.
- Agnostic support for underlying parallel file system.

To export a file from multiple NFSv4 servers, the servers need a common view of state. NFSv4 servers must therefore share copies of the state information, and must do so consistently, i.e., with single-copy semantics. We propose a *state server* to distribute the portions of state needed to serve READ and WRITE requests from data servers. The architecture of our design is shown in Figure 1. Throughout our design, we pay particular attention to the amount of additional communication needed to maintain consistency.

When an NFSv4 server receives an OPEN request from a client, it creates and maintains associated state. This state includes the open owner for the file, the level of access granted to the open owner via the share reservation, file handle, etc. We distribute this state among the data servers with a server-to-server protocol. The server then returns to the client the address of the data servers that manage the requested data. The client directs its READ and WRITE requests to these data servers. Once the client has completed all I/O requests on the open file, it sends a CLOSE request to the state server. The state server then reclaims the state from the data servers that previously received the state for this file and open owner.

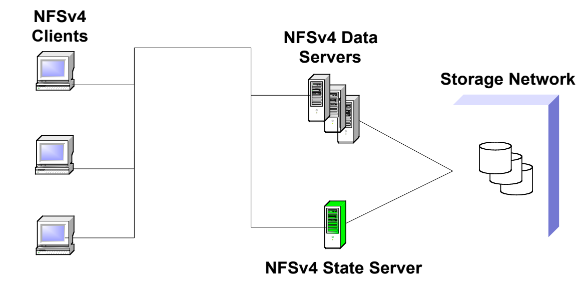

Figure 1: Parallel NSFv4 Architecture. Storage is accessed through the Linux VFS interface to a cluster or parallel file system, such as GPFS or Lustre. NFSv4 servers are divided into data servers, which handle all READ and WRITE requests, and a state server, which handles all other requests.

3.1 Configuration and Setup

The mechanics of a client connection to a server is the same as the standard NFSv4 protocol. However, here the client mounts a state server, not the data servers. Initially, we anticipate a single state server.

At start-up, data servers contact the state server and register as available data servers. Data servers are allowed to register with the state server at any time. The newly registered data servers are immediately available to NFSv4 clients for access. This allows easy incremental growth.

3.2 Server-to-Server Protocol

To process a READ or WRITE request, a data server must have an accurate picture of the current state of the system, such as the access rights of an open owner or the lock owner on a file. To provide this view, when the state serve receives an OPEN request, it first determines which data servers have registered to service the data request. It then creates the appropriate state for the request and transfers that state to the data servers it selected.

The following items constitute a unique identifier for share state:

- Client Name
- Client Verifier
- Client IP Address
- State Owner ID
- Access Bits
- Deny Bits
- File ID

File handle lock state has three additional items:

- Start byte
- End byte
- Lock type

On receiving this gift from the state server, the data server proceeds to recreate the NFSv4 state structures so that its view of the state for the file matches that of the state server. The data server can use the file handle to create export related data structures at this time or wait for a client to perform a PUTFH. Once the data server indicates that the state is successfully created, the state server refers the client to the data server.

On receiving a CLOSE request from the client, the state server contacts the data server to reclaim the state. Once the reclamation is completed, the standard NFSv4 close processing is executed.

3.3 Distribute State Operations

Each type of state requires a separate RPC call from the state server to data server. This section lists the arguments required for share, lock, and delegation state distribution.

```
const NFS4_FHSIZE = 128;
typedef opaque nfs_fh4<NFS4_FHSIZE>;
typedef uint64_t clientid4;
struct stateid4 {
   uint32_t seqid;
   opaque other[12];
};
enum nfs_lock_type4 {
   READ_LT = 1,
   WRITE_LT = 2,
   READW_LT = 3,
   WRITEW_LT = 4
};
struct deleg_type {
   READ_LT = 1,
   WRITE_LT = 2
};
struct DISTRIBUTE_SHARE_STATE_ARGS {
   clientid4 client;
   stateid4 state;
   uint32_t access_bits;
   uint32_t deny_bits;
   nfs_fh4 file_handle;
};
struct DISTRIBUTE_LOCK_STATE_ARGS {
   clientid4 client;
   stateid4 state;
   uint64_t offset;
   uint64_t length;
   nfs_lock_type4 type;
};
struct DISTRIBUTE_DELEGATION_STATE_ARGS {
   clientid4 client;
   deleg_type type;
   stateid4 state;
};
```

3.4 Recall State Operations

The state server invalidates state previously distributed to a data server using a single RPC operation.

```
struct RECALL_STATE_ARGS {
   clientid4 client;
   stateid4  state;
};
```

3.5 Redirection of Clients

The NFSv4 protocol recommends support for the attribute and associated error code NFS4ERR_FS_MOVED to allow migration or replication of an entire file system. To support client redirection, we extend FS_LOCATIONS and NFS4ERR_FS_MOVED by defining a new attribute and error code, FILE_LOCATIONS and NFS4ERR_FILE_MOVED, respectively. The new attribute allows redirection at the granularity of a file or portion of a file. As with FS_LOCATIONS, a client may request the FILE_LOCATIONS attribute at any time or may be directed to retrieve it when it receives the NFS4ERR_FILE_MOVED error code.

The FILE_LOCATIONS attribute includes these fields:

- List of data servers
- Time-to-live parameter
- Per data server root pathname
- Per data server supported operations
- Per data server lease maintenance indicator

Clients use this information to direct read and write commands (and others) to one of the data servers. The time-to-live parameter indicates the lifetime of the attribute. The root pathname allows each data server to have its own namespace. The supported operations mask declares which operations the data server will accept, e.g., read-only, etc. The lease maintenance indicator informs the client whether it must maintain leases on the state server, data server, or both.

3.6 Striped File Support

To support the export of files that are striped across multiple data servers, we extend the FILE_LOCATIONS attribute to include, for every data server, the byte ranges it exports. A more compact option is for FILE_LOCATIONS to include file layout properties (stripe size, layout pattern, etc), but this requires co-ordination with the underlying file system.

3.7 Load Balancing

The state server controls the lifetime of the FILE_LOCATIONS attribute through the time-time-live parameter. This allows the state server to load balance client requests among the data servers in replicated and cluster file system environments. Depending on the file access semantics of a system, a system administrator can customize the load-balancing algorithm to provide optimal utilization of resources.

The factors that directly affect the load balancing algorithm include:

- Number of data servers
- Proportion of read-only data servers
- Predicted average file size
- Predicted number of users
- Predicted file access pattern

4 Fault Tolerance

The addition of data servers in our design allows us to improve NFSv4's recovery strategy. Having multiple NFSv4 servers removes a single point of failure in the system. If the state server crashes, on recovery it can effectively use the data servers as stable storage to regenerate the system state and continue operation. A disadvantage of multiple NFSv4 servers is maintaining consistency of global state–the single point of failure is replaced with numerous points of failure. These failures are no longer fatal to system operation, but clients and servers now require extra processing to ensure consistency and to minimize access failures.

5 Security

NFSv4 mandates the use of RPCSEC_GSS [10] as its security mechanism to enable strong security. RPCSEC_GSS uses the GSS-API, allowing various security mechanisms to be used by the RPC layer without additional implementation overhead. The addition of data servers to the NFSv4 protocol does not require extra security mechanisms. The client continues to use the SECINFO command to negotiate the security protocol with the state server. Server implementations must also ensure the protection of the new server-to-server protocol.

6 Other Considerations

Determining when to reap a client's state is more complex when using multiple data servers because implicit renewals are distributed. One option is for data servers to inform the state server at regular intervals of all clients with which they have interacted. If any available data server does not report a client as active for some period, the state server invalidates the state.

7 Conclusion

This white paper presents a design that increases the scalability of NFSv4 by allowing the export of a single file via multiple NFSv4 servers. We make two contributions with this work. The first is a server-to-server protocol for NFSv4 that enables distributed state and file metadata maintenance. A state server manages all state information and distributes state to date servers as required. This reduces communication overhead by distributing state only to the necessary servers. The second contribution is the FILE_LOCATIONS attribute, which enables an NFSv4 server to migrate or replicate a single file or portions of a file.

References

[1] ASCI Project Team, *ASCI Purple RFP, Attachment 2, DRAFT STATEMENT OF WORK, Version 9*. Livermore, California, February 2002.

[2] "Earth Simulator Project," presented at Supercomputing 2002.

[3] "Fermi National Accelerator Laboratory http://www.fnal.gov/."

[4] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network File System (NFS) Version 4 Protocol," RFC 3530, April 2003.

[5] Frank Schmuck and Roger Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in Proc. USENIX Conf. on File and Storage Technologies, 2002.

[6] Peter J. Brahm, "Lustre: A Scalable, High-Performance File System," 2002.

[7] Chet Juszczak, "Improving the Write Performance of an NFS Server," in *Proc. Winter 1994 Technical Conf.*, 1994.

[8] Rick Macklem, "Not Quite NFS, Soft Cache Consistency for NFS," *in Proc. USENIX Winter Technical Conf.*, 1994.

[9] V. Srinivasan and Jeffrey C. Mogul, "Spritely NFS: Implementation and Performance of Cache-Consistency Protocols," Western Research Laboratory, Palo Alto, May 1989.

[10] M. Eisler, A. Chiu, and L. Ling, "RPCSEC_GSS Protocol Specification," *RFC 2203*, September 1997.