Building a Single Distributed File System from Many NFS Servers

-or-

The Poor-Man's Cluster Server

Dan Muntz

Hewlett-Packard Labs
1501 Page Mill Rd, Palo Alto CA 94304, USA

dmuntz@hpl.hp.com

Tel: +1-650-857-3561

Abstract

In this paper, we describe an architecture, *NFS^2*, for uniting several NFS servers under a single namespace. This architecture has some interesting properties. First, the physical file systems that make up an NFS^2 instance, i.e., the file systems on the individual NFS servers, may be heterogeneous. This, combined with the way the NFS^2 namespace is constructed, allows files of different types (text, video, etc.) to be served from file servers (potentially) optimized for each type. Second, NFS^2 storage is strictly partitioned—each NFS server is solely responsible for allocating the resources under its control. This eliminates resource contention and distributed lock management, commonly found in cluster file systems. Third, because the system may be constructed with standard NFS servers, it can benefit from existing high-availability solutions for individual nodes, and performance improves as NFS servers improve. Last, but not least, the system is extremely easy to manage—new resources may be added to a configuration by simply switching on a new server, which is then seamlessly integrated into the cluster. An extended version of this architecture is the basis for a completed prototype in Linux [5].

1 Introduction

NFS [1] servers are widely used to provide file service on the Internet. However, adding new servers to an existing namespace is management intensive, and in some ways inflexible. When a new server is brought online, all clients requiring access to the new server must be updated to mount any new file systems from the server, and access rights for the new file systems must be configured on the server. Additionally, the new file systems are bound to sub-trees of each client's namespace.

The NFS^2 architecture allows standard NFS servers to be combined into a single, scalable file system. Each NFS server is essentially treated as an object store. New servers added to an NFS^2 system merely add more object storage—they are not bound to a particular location in the namespace. Clients accessing the NFS^2 file system need not be aware when new NFS servers are added or removed from the system. The system takes its name from the fact that NFS is being used "on top of" NFS—the NFS protocol is being used to maintain object stores, and these object stores are combined into a single distributed file system that is exported via the NFS protocol.

2 Architecture

Figure 1 shows one possible configuration for an NFS^2 file system.

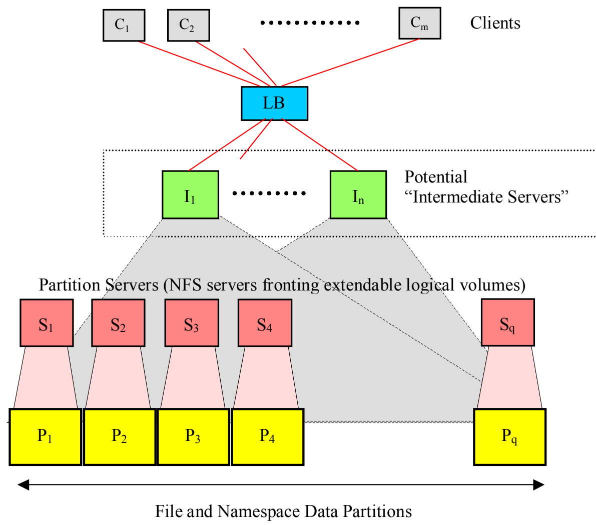

Figure 1: An NFS^2 File System

Storage partitions, P_i, are exported to the other parts of the system via standard NFS servers, S_i, also called *partition servers*. For scalability of the individual partitions/servers, *intermediate servers* can be introduced between the clients and servers. The intermediate servers accept NFS requests from the clients, and transform these requests into one or more NFS requests to the partition servers.

The intermediate servers perform another important, and powerful, function. Each partition server is used as an object store, but some entity must choose which partition is used for the creation of a new object. In the trivial case, this *placement policy* could simply be round-robin. A slightly more complex placement policy could choose a partition server based on resource balancing—choosing the partition server with the most storage available to balance storage resources, or choosing the partition server experiencing the least CPU load to do CPU load balancing. Even more complex placement policies are possible. For example, if one of the partition servers is a slow, legacy machine, and there is some knowledge of data access patterns, less-frequently-accessed data may be placed on the slow machine. This concept could be extended to integrate tertiary storage into the NFS^2 file system.

We describe the architecture under the assumption that the intermediate server translation functionality and placement policy are embedded in the partition servers and that clients issue requests directly to the partition servers. An implementation based on this assumption would retain most of the benefits of the complete system (possibly sacrificing some ability to scale with single-file "hot spots"), but would also have some beneficial simplifications (e.g., reduced leasing overhead, fewer network hops, etc.).

3 Design Considerations

The most important concept behind the construction of the NFS^2 namespace is the *cross-partition reference*. A directory residing in one partition may have children (files or directories) residing on another partition.

There are a couple of alternatives for implementing cross-partition references in NFS^2. Directories in most on-disk file systems are implemented as "files," however these directories have implementation-specific data and interfaces. If we are allowed to modify the NFS servers, directories can be implemented using regular files in the underlying physical file system. While this adds some overhead when compared to using the existing directory structures of the underlying file system, there are also benefits. Directory files may use a variety of data structures (e.g., hash tables, b-trees, etc.), and can surpass the performance of the typical linear list structure used in many systems [5]. More importantly for our purposes, with *directory files* we can extend directory entries to support cross-partition references, independently of the physical file systems. To achieve the goal of using unmodified NFS servers, symbolic links can be used to construct cross-partition references. We first describe the system in terms of directory files (for clarity), and follow this with a description of how the same functionality can be achieved with symbolic links. Fault tolerance and correctness for cross-partition references, without using distributed lock management (DLM), are addressed in another paper [7].

Another alternative is to store directories separately from files. Standard NFS servers are used to store files while separate servers are used for the namespace (either using directory files, or an alternative mechanism). Cross-partition references enable this separation of the namespace from the files. Servers for the namespace could be NFS servers modified to support directory files, a database, or some other construct.

The NFS^2 file system consists of user files and directory files. Both types of files exist as standard files in their respective partitions—a user file, `/usr/dict/words` might be represented as the file `/abc/123` on partition P_3, while the directory `/usr/dict` might be a file `/def/xxx` (containing *directory entries*) on partition P_4. An NFS^2 *directory entry* associates the user's notion of a file or directory name with the system's name for the file/directory, the partition where the file/directory is located, and any other relevant information. For example, some entries in `/def/xxx` could be represented as:

> .:/def/xxx:P_4
> ..:/yyy:P_6
> words:/abc/123:P_3

File handles passed to the client contain some representation of the system's name for the object (file or directory) and the partition where the object resides. This information is opaque to the client, but may be interpreted by the load balancer (LB) to direct requests to the correct partition server. Alternatively, the partition servers could be made the sole entities responsible for interpreting file handle information. A request could be sent to an arbitrary partition server that interprets the file handle and may then have to forward the request one "hop" (O(1)) to the server responsible for the object.

In the initial state, a well-known root partition server (say, P_1) contains a file, e.g., "`/root`", which corresponds to the user's view of the root of the NFS^2 distributed file system. The client mounts the file system by obtaining a file handle for the `/root` file as a special case of the lookup RPC.

Let us consider how some operations are handled in this file system. A `mkdir` request from a client will contain a file handle for the parent directory (pfh) and a name for the new directory (dname). A *switch function* is used by LB to direct the request to the partition server (P_x) where the new directory will reside. The switch function implements an arbitrary policy for where new file system objects are created (e.g., all video files might be placed on a "video server," or the switch function may chose the server with the most available capacity). P_x creates a new file representing dname that has the name dname′ in the physical file system served by P_x. P_x then issues a request to the partition server responsible for the parent directory, P_y (extracted from pfh), to add a directory entry: dname:dname′:P_x to the parent directory file (contained in pfh). If an entry for dname already exists, the operation is aborted and dname is removed from P_x. Otherwise, the new directory entry is added to the parent directory file and the operation completes.

The communication between P_x and P_y could be implemented using the standard NFS and lock manager protocols. P_x first locks the parent directory file, and checks for the existence of dname. If no entry for dname exists, it can issue an NFS write request to add the entry to the parent directory file. The directory file is subsequently unlocked. Alternatively, this communication could take place via a simple supplementary protocol that would allow the locking to be more efficient—a single RPC is sent to P_y, which then uses local file locking for the existence checking and update, and returns the completion status.

File creation is essentially identical to `mkdir`.

The `read` and `write` operations are trivial (referring to the definitions from Figure 1):

write(fh, data, offset, length):

C_i sends the request to LB.
LB looks into fh and directs the request to the appropriate S_j.
S_j issues a local write call to the file specified in fh.

`Read` is similar.

To construct cross-partition references with symbolic links, we can build an NFS^2 cluster as a proof-of-concept as follows. First, each partition is assigned a name (assume P_i, as in Figure 1). The NFS servers then mount all partitions into their local namespace at locations /P1, /P2, etc. using the standard mount protocol. Now, a cross-partition reference is created by making a symbolic link that references the physical file through one of these mount points.

For the example:

words:/abc/123:P_3

An underlying file, `/abc/123`, contains the data for the file, and resides on partition P_3. The namespace entry `words` is a symbolic link in its parent directory with the link contents: `/P3/abc/123`. It is important to note that we are talking about systems on the scale of a cluster file system, so the cross-mounting does not involve a "huge" number of servers. An extension to this work [5] looks at expanding the architecture to a global scale.

4 Future Work

There are several areas requiring further investigation. The performance of the architecture in its various possible incarnations (the symbolic link version, the directory file version, and others) must be studied.

We also want to investigate the potential uses and performance implications of directory files. Directory files were conceived for the NFS^2 architecture to address the problem of providing a single directory structure over diverse underlying file systems, and the need for an easily extensible directory structure. Such benefits may be useful for other file system research. Also, because directory files allow the directory structure to be flexible, they can be used to investigate alternative data structures for directories, alternative naming schemes, new access control mechanisms, and new types of information that might be associated with files.

Due to the structure of cross-partition references, object-level migration should be relatively straight-forward in NFS^2. Migration and replication are two more areas requiring further research.

5 Related work

There has been a significant amount of research and product development in the area of cluster file systems [2,4,8]. Most are based on principles established in the VAXclusters [2] design. These systems use distributed lock management to control access to shared resources, which can restrict their scalability. NFS^2 partitions resources to eliminate DLM [5,7].

Frangipani proposed one of the most scalable DLM solutions in the literature [8]. System resources are partitioned into logical volumes [3] and there is one DLM server dedicated to each volume. This requires using two levels of virtualization: virtual disk and file system. NFS^2 resembles Frangipani in its partitioning of the storage resources for improving contention control. However, NFS^2 uses one level of virtualization allowing decisions for resource utilization and file placement to be made at the file service level. Also, cluster file systems, including Frangipani, depend on their own, proprietary physical file system. NFS^2 is a protocol-level service and can leverage diverse file systems for optimal content placement and delivery. Nevertheless, NFS^2 is complementary to cluster file systems—a partition can be implemented as a cluster file system and can be integrated into a broader file space.

Slice [6] is a system that also uses a partitioning approach, similar to NFS^2. Slice's file placement policies (small versus large files and a deterministic distribution within each class of files) are implemented in µ*proxie*s—modules that forward client operations to the right partition, operating at the IP layer. To make placement decisions, µ*proxies* have to maintain a view of the server membership in the system. In case of reconfiguration, the new membership information is diffused among the (possibly thousands of) µ*proxies* in a lazy fashion. As a result, resource reconfiguration in Slice is coarse-grained; also, file allocation is static for the duration of an object's life. In comparison, NFS^2 can extend the traditional file system namespace metadata to achieve highly flexible and dynamic file placement and resource reconfiguration. However, this requires extensions (even if minor) to the client access protocol. Slice's µproxy idea could be used to transparently intercept client-service communication and redirect it to the appropriate partition server. In that case, µproxies will not need to maintain distribution tables; instead, they will interpret the contents of the (opaque to the client) file handles to retrieve the location of the server for each client request.

6 Conclusions

NFS^2 provides a mechanism for uniting NFS servers under a single namespace. It simplifies management of multiple NFS servers by providing access to all servers through a single namespace (no need for multiple client mount points), and by providing a transparent mechanism for the addition of new servers as the system grows.

This system avoids distributed lock management, which has been a limiting factor in the scalability of cluster file systems. NFS^2 supports heterogeneous physical file systems within the single namespace, whereas other systems have relied on their own proprietary physical file systems. Support for arbitrary *placement policies* to place files on certain servers allows a great deal of flexibility, including placement of files on servers optimized for a given file's content type, load balancing, storage balancing, and others.

7 References

1. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., *Design and Implementation of the Sun Network Filesystem*, in *Proc. of the Summer USENIX Technical Conference*, Portland, OR, USA, June 1985.
2. Kronenberg, N., H. Levy, and W. Stecker, *VAXClusters: A closely-coupled distributed system.* ACM Tansactions on Computer Systems, 1986, 4(2): pp. 130-146.
3. Lee, E. and C. Thekkath. *Petal: Distributed Virtual Disks*. In *ASPLOS VII*, MA, USA, 1996.
4. *Veritas Cluster File System (CFS)*, 2000, Veritas Corp., Mountain View, California. http://www.veritas.com.
5. C. Karamanolis, M. Mahalingam, L. Liu, D. Muntz and Z. Zheng. *An Architecture for Scalable and Manageable File Services*. HP Labs Technical Report No. HPL-2001-173 (7/12/2001).
6. Anderson, D., Chase, J., and Vadhat, A., *Interposed Request Routing for Scalable Network Storage*, in *Proc. of the USENIX OSD*I, San Diego, CA, USA, October, 2000.
7. Zhang, Z. and Karamanolis, C., *Designing a Robust Namespace for Distributed File Services*, in *Proc. of the 20th Symposium on Reliable Distributed System*s. New Orleans, USA. IEEE Computer Society, October 28-31, 2001.
8. Thekkath, C., T. Mann, and E. Lee. *Frangipani: A Scalable Distributed File System*. In *16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, 1997.