CHAPTER 14

To Protect and Serve: Providing Data to Your Cluster

Chapter Objectives

- Introduce the concept of a cluster file system
- Explore available file system options
- Present examples of cluster file system configurations

The compute slices in a cluster work on pieces of a larger problem. Without the ability to read data and write results, the cluster's computation is of little value. Whether the cluster is a database cluster or a scientific cluster, the compute slices need coordinated access to the shared data that is "behind" their computations. This chapter introduces some of the design issues associated with providing data to multiple compute slices in a cluster and presents some of the currently available solutions.

14.1 Introduction to Cluster File Systems

When talking about cluster file systems we have to be careful about how we define the phrase. One simple definition might be: "a file system that is capable of serving data to a cluster." Another more complex definition might be: "a file system that allows multiple systems to access shared file system data independently, while maintaining a consistent view of that data between clients."

There is a subtle difference in the two definitions to which we need to pay close attention. The first definition might include network-based file systems like the NFS, which has only advisory locking and can therefore allow applications to interfere with each other destructively. The second definition tends to imply a higher level of parallelism coupled with consistency checks and locking to prevent collisions between multiple application components accessing the same data. Although the first definition is somewhat vague, the second definition might be said to refer to a "parallel file system."

14.1.1 Cluster File System Requirements

Because the I/O to a cluster's file system is being performed by multiple clients at the same time, there is a primary requirement for performance. Systems that run large parallel jobs often perform data "checkpointing" to allow recovery in the event that the job fails. There may be written requirements for a "full cluster checkpoint' within a given period of time. This has very direct requirements in terms of file system throughput.

If one assumes that the compute slices in the cluster must simultaneously write all memory to the disk for a checkpoint in a given length of time, you can compute the upper boundary requirement for the cluster file system's write performance. For example, 128 compute slices, each with 4 GB of RAM, would generate 512 GB of checkpoint data. If the period of time required for a clusterwide checkpoint is ten minutes, this yields 51.2 GB/min or 853 MB/s.

This is an extraordinarily high number, given the I/O capabilities of back-end storage systems. For example, a single 2-Gbps Fibre-Channel interface is capable of a theoretical maximum throughput of 250 MB/s. Getting 80% of this theoretical maximum is probably wishful thinking, but provides 200 MB/s of write capability. To "absorb" the level of I/O we are contemplating would take (at least) five independent 2 Gbps Fibre-Channel connections on the server. This ignores the back-end disk array performance requirements. Large, sequential writes are a serious performance problem, particularly for RAID 5 arrays, which must calculate parity information.

This example assumes a compute cluster performing sequential writes for a checkpoint operation. The requirements for database applications have different I/O characteristics, but can be just as demanding in their own right. Before we get too much ahead of ourselves, let's consider the general list of requirements for a cluster's file system. Some of the attributes we seek are

- High levels of performance
- Support for large data files or database tables
- Multiuser and multiclient security
- Data consistency
- Fault tolerance

One might also add "consistent name space" to this list. This phrase refers to the ability for all clients to access files using the same file system paths. Some file systems provide this ability automatically (AFS, the DCE distributed file system [DFS], and Microsoft's distributed file system [dfs]), and others require the system administrator to create a consistent access structure for the clients (NFS, for example). We will need to watch for situations that might affect the client's view of the file system hierarchy.

There are some standard questions that we can ask when we need to analyze the architecture of a given cluster file system solution.

How is the file system storage shared?
How many systems can access the file system simultaneously?
How is the storage presented to client systems?
Is data kept consistent across multiple systems and their accesses, and if so, how?
How easy is the file system to administer?
What is the relative cost of the hardware and software to support the file system?
How many copies of the file system code are there and where does it run?
How well does the architecture scale?

These questions, and others, can help us to make a choice for our cluster's file system. Two primary types of file system architectures are available to us: network-attached storage (NAS) and parallel access. As we will see in the upcoming sections, each has their own fans and detractors, along with advantages and disadvantages.

14.1.2 Networked File System Access

Both NFS and CIFS are client–server implementations of "remote" network-attached file systems. These implementations are based on client-to-server RPC calls that allow the client's applications to make normal UNIX file system calls (open, close, read, write, stat, and so on) that are redirected to a remote server by the operating system. The server's local, physical file systems are exported via an Ethernet network to the server's clients.

The server's physical file systems, such as `ext3`, `jfs`, `reiser`, or `xfs`, which are exported via NFS, exist on physical storage attached to the server that exports them. The I/O between the network, system page cache, and the file blocks and meta-data on the storage is performed only by a single server that has the file system mounted and exported. In this configuration, the physical disks or disk array LUNs, and the data structures they contain, are modified by a single server system only, so coordination between multiple systems is not necessary. This architecture is shown in Figure 14–1.

Figure 14–1 shows two disk arrays that present a single RAID 5 LUN each. Because each array has dual controllers for redundancy, the NFS servers "see" both LUNs through two 2-Gbps (full-duplex) Fibre-Channel paths. (It is important that the two systems "see" the same view of the storage, in terms of the devices. There are "multipath I/O" modules and capabilities available in Linux that can help with this task.) The primary server system uses the LUNs to create a single 2.8-TB file system that it exports to the NFS clients through the network. The secondary server system is a standby replacement for the primary and can replace the primary system in the event of a server failure. To keep the diagram from becoming too busy, the more realistic example of one LUN per array controller (four total) is not shown.

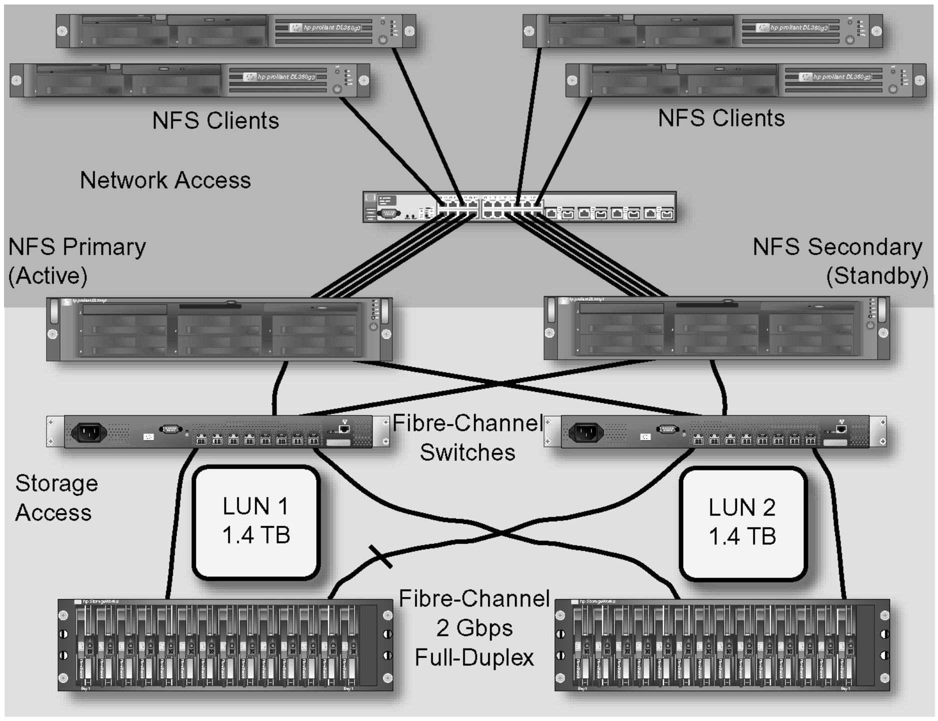

Figure 14–1 A network-attached file system configuration

The NAS configuration can scale only to a point. The I/O and network interface capabilities of the server chassis determine how many client systems the NFS server can accommodate. If the physical I/O or storage attachment abilities of the server are exceeded, then the only easy way to scale the configuration is to split the data set between *multiple* servers, or to move to a larger and more expensive server chassis configuration. The single-server system serializes access to file system data and meta-data on the attached storage while maintaining consistency, but it also becomes a potential performance choke point.

Splitting data between multiple equivalent servers is inconvenient, mainly because it generates extra system administration work and can inhibit data sharing. Distributing the data between multiple servers produces multiple back-up points and multiple SPOFs, and it makes it difficult to share data between clients from multiple servers without explicit knowledge of which server “has” the data. If read-only data is replicated between multiple servers, the client access must somehow be balanced between the multiple instances of the data, and the system administrator must decide how to split and replicate the data.

In the case of extremely large data files, or files that must be both read and written by multiple groups of clients (or all clients) in a cluster, it is impossible to split either the data or the access between multiple NFS servers without a convoluted management process. To avoid splitting data, it is necessary to provide very large, high-performance NFS servers that can accommodate the number of clients accessing the data set. This can be a very expensive approach and once again introduces performance scaling issues.

Another point to make about this configuration is that although a SAN may appear to allow multiple storage LUNs containing a traditional physical file system to be simultaneously accessible to multiple server systems, this is not the case. With the types of local file systems that we are discussing, only one physical system may have the LUN and its associated file system data mounted for read *and* write at the same time. So only one server system may be exporting the file system to the network clients at a time. (This configuration is used in active standby high-availability NFS configurations. Two [or more] LUNs in a shared SAN may be mounted by a server in read–write mode, whereas a second server mounts the LUNs in read-only mode, awaiting a failure. When the primary server fails, the secondary server may remount the LUNs in read–write mode, replay any journal information, and assume the IP address and host name of the primary server. The clients, as long as they have the NFS file systems mounted with the `hard` option, will retry failed operations until the secondary server takes over.)

The SAN's storage LUNs may indeed be "visible" to any system attached to the SAN (provided they are not "zoned" out by the switches), but the ability to read and write to them is not available. This is a limitation of the physical file system code (not an inherent issue with the SAN), which was not written to allow multiple systems to access the same physical file system's data structures simultaneously. (There are, however, restrictions on the number of systems that can perform "fabric logins" to the SAN at any one time. I have seen one large cluster design fall into this trap, so check the SAN capabilities before you commit to a particular design.) This type of multiple-system, read–write access requires a parallel file system.

14.1.3 Parallel File System Access

If you imagine several to several *thousand* compute slices all reading and writing to separate files in the same file system, you can immediately see that the performance of the file system is important. If you imagine these same compute slices all reading and writing to the *same file* in a common file system, you should recognize that there are other attributes, in addition to performance, that the file system must provide. One of the primary requirements for this type of file system is to keep data consistent in the face of multiple client accesses, much as semaphores and spin locks are used to keep shared-memory resources consistent between multiple threads or processes.

Along with keeping the file data blocks consistent during access by multiple clients, a parallel cluster file system must also contend with multiple client access to the file system meta-data. Updating file and directory access times, creating or removing file data in directories, allocating inodes for file extensions, and other common meta-data activities all become more complex when the parallel coordination of multiple client systems is considered. The use of SAN-based storage makes shared *access* to the file system data from multiple clients easier, but the

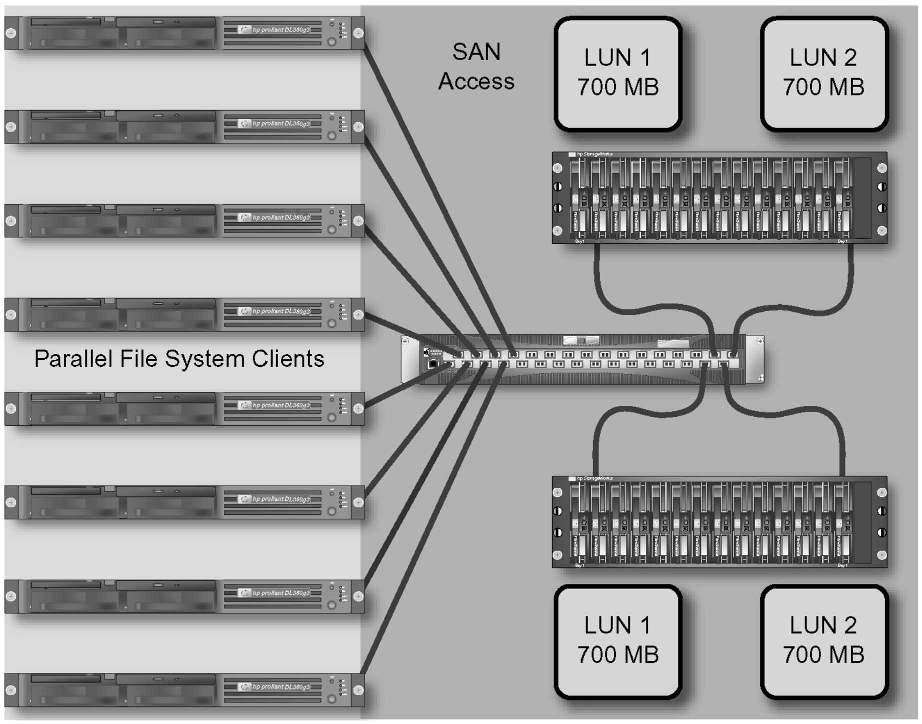

Figure 14–2 A parallel file system configuration

expense of Fibre-Channel disk arrays, storage devices, and Fibre-Channel hub or switch interconnections must be weighed against the other requirements for the cluster's shared storage.

Figure 14–2 shows a parallel file system configuration with multiple LUNs presented to the client systems. Whether the LUNs may be striped together into a single file system depends on which parallel file system implementation is being examined, but the important aspect of this architecture is that each client has shared access to the data in the SAN, and the client's *local* parallel file system code maintains the data consistency across all accesses.

A primary issue with this type of "direct attach" parallel architecture involves the number of clients that can access the shared storage without performance and data contention issues. This number depends on the individual parallel file system implementation, but a frequently encountered number is in the 16 to 64 system range. It is becoming increasingly common to find a cluster of parallel file system clients in a configuration such as this acting as a file server to a larger group of cluster clients.

Such an arrangement allows scaling the file system I/O across multiple systems, while still providing a common view of the shared data "behind" the parallel file server members. Using another technology to "distribute" the file system access also reduces the number of direct attach SAN ports that are required, and therefore the associated cost. SAN interface cards and switches are still quite expensive, and a per-attachment cost, including the switch port and interface card, can run in the $2,000- to $4,000-per-system range.

There are a variety of interconnect technologies and protocols that can be used to distribute the parallel file system access to clients. Ethernet, using TCP/IP protocols like Internet SCSI (iSCSI), can "fan out" or distribute the parallel file server data to cluster clients. It is also possible to use NFS to provide file access to the larger group of compute slices in the cluster. An example is shown in Figure 14–3.

The advantage to this architecture is that the I/O bandwidth may be scaled across a larger number of clients without the expense of direct SAN attachments to each client, or the requirement for a large, single-chassis NFS server. The disadvantage is that by passing the parallel file system "through" an NFS server layer, the clients lose the fine-grained locking and other advantages of the parallel file system. Where and how the trade-off is made in your cluster depends on the I/O bandwidth and scaling requirements for your file system.

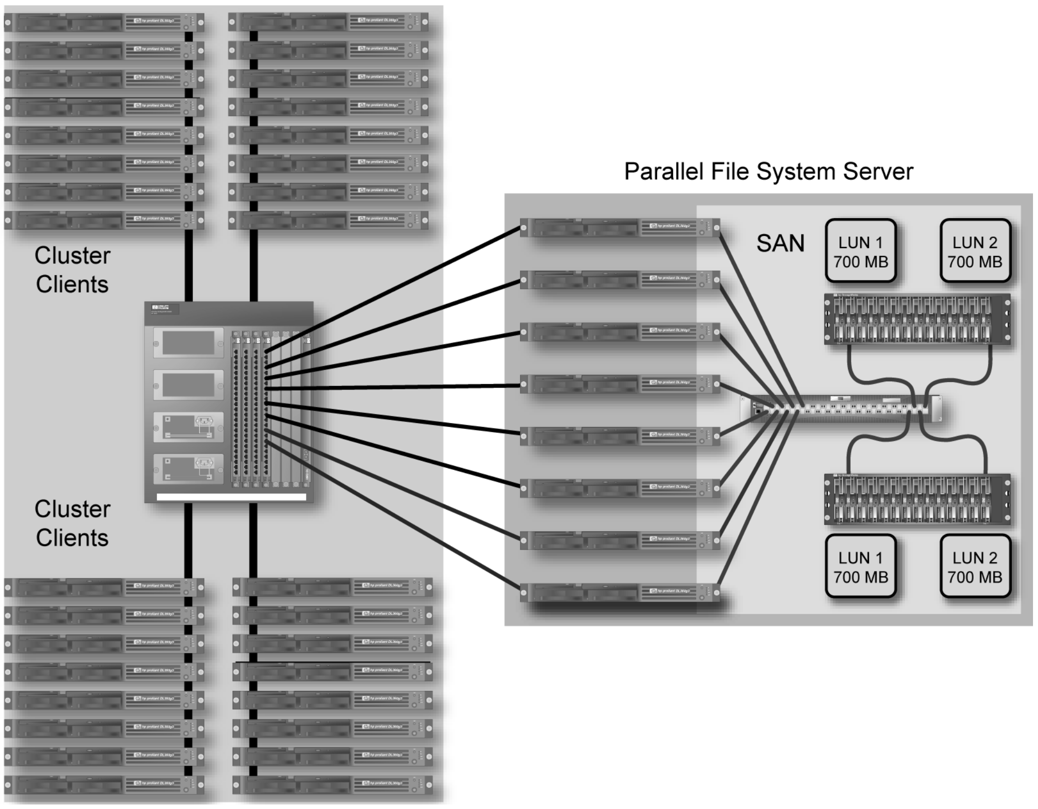

Figure 14–3 A parallel file system server

The example in Figure 14–3 shows an eight-system parallel file server distributing data from two disk arrays. Each disk array is dual controller, with a separate 2-Gbps Fibre-Channel interface to each controller. If a file system were striped across all four LUNs, we could expect the aggregate write I/O to be roughly four times the real bandwidth of a 2 Gbps FC link, or 640 MB/s (figuring 80% of 200 MB/s).

Dividing this bandwidth across eight server systems requires each system to deliver 80 MB/s, which is the approximate bandwidth of one half of a full-duplex GbE link (80% of 125 MB/s). Further dividing this bandwidth between four client systems in the 32-client cluster leaves 20 MB/s per client. At this rate, a client with 8 GB of RAM would require 6.67 minutes to save its RAM during a checkpoint.

Hopefully you can begin to see the I/O performance issues that clusters raise. It is likely that the selection of disk arrays, Fibre-Channel switches, network interfaces, and the core switch will influence the actual bandwidth available to the cluster's client systems. Scaling up this file server example involves increasing the number of Fibre-Channel arrays, LUNs, file server nodes, and GbE links.

The examples shown to this point illustrate two basic approaches to providing data to cluster compute slices. There are advantages and disadvantages to both types of file systems in terms of cost, administration overhead, scaling, and underlying features. In the upcoming sections I cover some of the many specific options that are available.

14.2 The NFS

A primary advantage of using NFS in your cluster is that it is readily available and part of most Linux distributions. Its familiarity also is an advantage to most UNIX or Linux system administrators, who are able to deploy NFS almost in their sleep. The knowledge needed to tune and scale an NFS server and the associated clients, however, is not as readily available or applied as you might think.

The implementation of features in NFS also varies between Linux distributions, the version of a particular Linux distribution, and between the NFS client and server software components. Some of the features that we may be used to having on proprietary UNIX systems are not yet available on Linux, or are in a partially completed state. NFS on Linux can be a moving target in some respects.

One example of this type of incomplete functionality lies in the area of NFS read and write block sizes, which may default to 1 KB on some versions and may not be set larger than 8 KB on others—limiting potential performance. Another area is the support of NFS over TCP, which is enabled for Fedora Core 1, but not for some earlier versions of Red Hat, like 8.0.

Because NFS is a client–server implementation of a networked file system, setup and tuning is necessary on both the client and the server to obtain optimum performance. Understanding the major NFS software components and the complete NFS data path can help select the best operating parameters for your installation. I will cover these topics in upcoming sections.

14.2.1 Enabling NFS on the Server

Once the local physical file systems are built on your NFS server, there are several steps necessary to make them available to NFS clients in the network. Before starting the NFS server subsystem, there are a number of options that we need to consider for proper operation. We also need to pay attention to potential security issues.

The NFS server processes are started from the `/etc/init.d/nfs` and `/etc/init.d/nfslock` files. Both of these files may be enabled with the customer `chkconfig` commands:

```
# chkconfig nfs on
# chkconfig nfslock on
```

The behavior of NFS on your server may be controlled by the `/etc/sysconfig/nfs` file. This file contains the options used by the `/etc/init.d/nfs` file, and usually does not exist by default.

The `/etc/init.d/nfs` file uses three programs that are essential to the NFS server subsystem: `/usr/sbin/rpc.mountd`, `/usr/sbin/rpc.nfsd`, and `/usr/sbin/exportfs`. The `rpc.mountd` daemon accepts the remote mount requests for the server's exported file systems. The `rpc.nfsd` process is a user space program that starts the `nfsd` kernel threads from the `nfsd.o` module that handles the local file system I/O on behalf of the client systems.

The `exportfs` command is responsible for reading the `/etc/exports` file and making the exported file system information available to both the kernel threads and the mount daemon. Issuing the `exportfs -a` command will take the information in `/etc/exports` and write it to `/var/lib/nfs/xtab`.

The number of `nfsd` threads that are started in the kernel determine the simultaneous number of requests that can be handled. The default number of threads started is eight, which is almost never enough for a heavily used NFS server. To increase the number of `nfsd` threads started, you can set a variable in the `/etc/sysconfig/nfs` file:

```
RPCNFSDCOUNT=128
```

The exact number of threads to start depends on a lot of factors, including client load, hardware speed, and the I/O capabilities of the server.

The `nfsd` threads receive client requests on socket number 2049 for either UDP/IP or TCP/IP requests. Which transport is used as a default depends on the Linux distribution and version. Some versions do not have NFS over TCP available. Performance is better over UDP, provided the network is robust.

One way to tell if you need more `nfsd` threads is to check for NFS socket queue overflows. Unfortunately, this method will tell you only if some socket on the system has overflowed, not the specific NFS socket. If the system is primarily an NFS server, you can infer that the most likely socket to overflow is the NFS socket. Check for this situation with

```
# netstat -s | grep overflow
    145 times the listen queue of a socket overflowed
```

If you are seeing socket overflows on the server, then try increasing the number of nfsd threads you are starting on your server. If the NFS server's input socket queue overflows, then a client request packet is dropped. This will force a client retry, which should show up in client NFS statistics available with the nfsstat command (on a client system):

```
# nfsstat -rc
Client rpc stats:
calls      retrans    authrefrsh
20         0          0
```

The number of hits in the retrans counter would potentially reflect any server issues with socket overflows, although there are other reasons for this number to increase, such as the end-to-end latency between the client and server or the server being too busy to reply within the time-out period set by the client. I discuss setting the client request time-out values when I talk about mount parameters.

You can easily tell how many nfsd threads are currently running on the system with the ps command. For example,

```
# ps -ef | grep nfs| grep -v grep
root       855     1  0 Mar18 ?        00:00:00 [nfsd]
root       856     1  0 Mar18 ?        00:00:00 [nfsd]
root       857     1  0 Mar18 ?        00:00:00 [nfsd]
root       858     1  0 Mar18 ?        00:00:00 [nfsd]
root       859     1  0 Mar18 ?        00:00:00 [nfsd]
root       860     1  0 Mar18 ?        00:00:00 [nfsd]
root       861     1  0 Mar18 ?        00:00:00 [nfsd]
root       862     1  0 Mar18 ?        00:00:00 [nfsd]
```

This shows the eight *default* nfsd threads executing on a server system. Oops. The fact that daemons are kernel threads is indicated by the brackets that surround the "process" name in the ps output.

14.2.2 Adjusting NFS Mount Daemon Protocol Behavior

The rpc.mountd process provides the ability to talk to clients in multiple NFS protocols. The most recent protocol, NFS protocol version 3 (NFS PV3) is supported, along with the older protocol version 2 (NFS PV2). Which protocol is used is selected based on the NFS client's mount request, or on the options supplied to the nfsd threads when they are started. The rpc.nfsd command is used to pass the protocol options to the kernel threads when they are started.

If you examine the man page for rpc.nfsd, you will see two options that control the available NFS protocols: --no-nfs-version and --nfs-version. Either of these options may be followed by the value 2 or 3 to disallow or force a particular NFS protocol version respectively. The /etc/sysconfig/nfs file provides two variables that translate into these options.

The MOUNTD_NFS_V2 variable and the MOUNTD_NFS_V3 variable may be set to the values no, yes, or auto. A value of no disallows the associated protocol version, a value of yes enables the associated protocol version, and a value of auto allows the start-up script to offer whichever versions are compiled into the kernel. An example /etc/sysconfig/nfs file that disallows NFS PV2 and enables only NFS PV3 would contain

```
MOUNTD_NFS_V2=no
MOUNTD_NFS_V3=yes
```

It is better to offer the available mount types, and control which one is used with the client mount parameters, unless there are issues between the client and server.

It is possible to determine which version of the NFS protocol a client system is using by looking at statistics from the nfsstat command. This command, in addition to providing retry information, will show the number of NFS PV3 and PV2 RPC requests that have been made to the server. An example output from this command is

```
# nfsstat -c
Client rpc stats:
calls      retrans    authrefrsh
20         0          0
Client nfs v2:
null   getattr setattr root lookup readlink
0 0%   0 0%   0 0%   0 0%   0 0%   0 0%
read       wrcache    write      create     remove     rename
0 0%      0 0%      0 0%      0 0%      0 0%      0 0%
link       symlink    mkdir      rmdir      readdir    fsstat
0 0%      0 0%      0 0%      0 0%      0 0%      0 0%
Client nfs v3:
null       getattr    setattr    lookup     access readlink
0 0%      14 70%    0 0%      0 0%      0 0%  0 0%
read       write      create     mkdir      symlink    mknod
0 0%      0 0%      0 0%      0 0%      0 0%      0 0%
remove     rmdir      rename     link readdir readdirplus
0 0%      0 0%      0 0%      0 0% 0 0%  0 0%
fsstat     fsinfo     pathconf   commit
0 0%      6 30%     0 0%      0 0%
```

The statistics kept by nfsstat are reset at each reboot or when the command is run with the zero-the-counters option. This command is

```
# nfsstat -z
```

It is also possible to use the `nfsstat` command to examine the server statistics for all clients being serviced. If there are clients using one or the other or both protocol versions, this is visible in the command's output, because the RPC calls are divided into groups according to protocol version. An example of the server statistics from `nfsstat` follows:

```
# nfsstat -s
Server rpc stats:
calls      badcalls   badauth    badclnt    xdrcall
1228       0          0          0          0
Server nfs v2:
null       getattr    setattr    root lookup     readlink
1 100%    0 0%       0 0%      0 0% 0 0%      0 0%
read       wrcache    write      create     remove     rename
0 0%      0 0%       0 0%      0 0%      0 0%      0 0%
link       symlink    mkdir      rmdir      readdir    fsstat
0 0%      0 0%      0 0%      0 0%      0 0%      0 0%

Server nfs v3:
null       getattr    setattr    lookup access      readlink
1 0%      212 17%   0 0%      83 6%  141 11%   0 0%
read       write      create     mkdir      symlink    mknod
709 57%   0 0%      0 0%      0 0%      0 0%      0 0%
remove     rmdir      rename     link readdir    readdirplus
0 0%      0 0%      0 0%      0 0% 22 1%    38 3%
fsstat     fsinfo     pathconf   commit
3 0%      18 1%     0 0%      0 0%
```

As you can see from the output of `nfsstat`, the server and client are both using NFS PV3 to communicate. Furthermore, by looking at the server, you can see that all clients are using NFS PV3, because there are no RPC calls listed for the PV2 categories. I am getting just a little ahead, because the `nfsstat` statistics are available only after NFS has been started. We aren't quite ready for this situation yet.

On an NFS server with a considerable number of clients, the maximum number of open file descriptors allowed by the NFS mount daemon may need to be adjusted. The `rpc.mountd` process uses the `--descriptors` option to set this, and the default number is 256. This value has to be passed with the `RPCMOUNTD_OPTS` variable in `/etc/sysconfig/nfs`:

```
RPCMOUNTD_OPTS='--descriptors=371'
```

This number represents the number of file handles allowed by the server; one is maintained per NFS mount. It is not the number of files that may be opened by the clients.

14.2.3 Tuning the NFS Server Network Parameters

The `/etc/sysconfig/nfs` file allows tuning the input queue size for the NFS sockets. This is done by temporarily altering the system default parameters for the networking subsystem, specifically the TCP parameters. To understand how this is done, we need to look at the general parameter tuning facility provided by the /sbin/sysctl command.

This command reads the contents of the /etc/sysctl.conf file and applies the parameters listed there to the appropriate system parameters. The application of kernel parameter changes is usually done at start-up time by the init scripts, but may also be done on a "live" system to alter the default kernel parameter values.

Using the sysctl command, temporary changes may be applied to the system's kernel parameters, and these "tunes" may be permanently applied by adding them to the /etc/sysctl.conf configuration file. The NFS start-up script allows us to tune only the NFS socket parameter, instead of making global changes. The commands to do this are listed from the /etc/init.d/nfs start-up file:

```
# Get the initial values for the input sock queues
# at the time of running the script.
if [ "$TUNE_QUEUE" = "yes" ]; then
    RMEM_DEFAULT='/sbin/sysctl -n net.core.rmem_default'
    RMEM_MAX='/sbin/sysctl -n net.core.rmem_max'
    # 256kb recommended minimum size based on SPECsfs
    # NFS benchmarks
    [ -z "$NFS_QS" ] && NFS_QS=262144
fi
[... intermediate commands deleted ...]
case "$1" in
  start)
# Start daemons.
        # Apply input queue increase for nfs server
        if [ "$TUNE_QUEUE" = "yes" ]; then
            /sbin/sysctl -w net.core.rmem_default=$NFSD_QS \
               >/dev/null 2>&1
            /sbin/sysctl -w net.core.rmem_max=$NFSD_QS     \
               >/dev/null 2>&1
        fi
[... intermediate commands deleted ...]
# reset input queue for rest of network services
        if [ "$TUNE_QUEUE" = "yes" ]; then
            /sbin/sysctl -w                         \
               net.core.rmem_default=$RMEM_DEFAULT  \
               >/dev/null 2>&1
            /sbin/sysctl -w                         \
               net.core.rmem_max=$RMEM_MAX          \
               >/dev/null 2>&1
        fi
```

The script temporarily modifies the values of the net.core.rmem_max and net.core.rmem_default parameters, starts the nfsd threads, then replaces the old values of the parameters. This has the effect of modifying the receive memory parameters for only the socket that gets created by nfsd threads for receiving NFS requests, socket 2049.

These network parameter values are also available in the files /proc/sys/net/core/rmem_max and /proc/sys/net/core/rmem_default. Note the similarity between the file path name in /proc and the parameter names used with the sysctl command. They are indeed the same data; there are just several methods of modifying the parameter value. We could alter the values (globally!) for these parameters by executing

```
# echo '262144' > /proc/sys/net/core/rmem_default
# echo '262144' > /proc/sys/net/core/rmem_max
```

Any changes we made this way would alter the defaults for all new sockets that are created on the system, until we replace the values.

You can crash or hang your system by making the wrong adjustments. Exercise caution when tuning parameters this way. It is, however, best to try values temporarily before committing them to the /etc/sysctl.conf file or to /etc/sysconfig/nfs. To enter parameters prematurely in this file can render your system unbootable (except in single-user mode). To perform these tuning operations when NFS is started, we can set the following values in the file /etc/sysconfig/nfs:

```
TUNE_QUEUE=yes
NFS_QS=350000
```

14.2.4 NFS and TCP Wrappers

If you use TCP wrappers (/etc/hosts.deny and /etc/hosts.allow) to tighten security in the cluster's networks, you need to make special provisions for NFS. There are a number of services that must be allowed access for proper NFS operation. One of the standard ways to use TCP wrappers is to deny any service that is not explicitly enabled, so if you use this approach, you need to modify the /etc/hosts.allow file on the NFS server to contain

```
mountd:     .cluster.local
portmap:    .cluster.local
statd:      .cluster.local
```

The client systems must be able to access these services on the server to mount the server's exported file systems and to maintain information about the advisory locks that have been created. Even though the binary executables for the daemons are named rpc.mountd and rpc.statd, the service names must be used in this file.

14.2.5 Exporting File Systems on the NFS Server

The format of the `/etc/exports` file on Linux systems is a little different than that of other UNIX systems. The file provides the necessary access control information to the NFS mount daemon, determining which file systems are available and which clients may access them. The `exportfs` command takes the information from the `/etc/exports` file and creates the information in the `/var/lib/nfs/xtab` and `/var/lib/nfs/rmtab` files, which are used by `mountd` and the kernel to enable client access to the exported directories.

The `/etc/exports` file contains lines with a directory to export, and a whitespace-separated list of clients that can access it, and each client may have a list of export options enclosed in parentheses. The client specifications may contain a variety of information, including the specific host name, wild cards in a fully qualified host name, a network address and net mask, and combinations of these elements. An example file for our cluster might be

```
/scratch        cs*.cluster.local(rw,async)
/admin          10.3.0.0/21(rw)
/logs           ms*.cluster.local(rw) cs*.cluster.local(rw)
```

An entry in the `/var/lib/nfs/xtab` file for an exported directory with no options specified is listed (for each system with a mount) as

```
/kickstart      cs01.cluster.local(ro,sync,wdelay,hide,      \
secure,root_squash,no_all_squash,subtree_check,secure_locks,\
mapping=identity,anonuid=-2,anongid=-2)
```

This shows the default values for the options in the `/etc/exports` entry:

- Read-only access.
- Synchronous updates (don't acknowledge NFS operations until data is committed to disk).
- Delay writes to see if related sequential writes may be "gathered."
- Require file systems mounted within other exported directories to be individually mounted by the client.
- Require requests to originate from network ports less than 1024.
- Map `root` (UID or GID 0) requests to the `anonymous` UID and GID.
- Do not map UIDs or GIDs other than `root` to the `anonymous` UID and GID.
- Check to see whether accessed files are in the appropriate file system *and* an exported system tree.
- Require authentication of locking requests.
- User access permission mapping is based on the identity specified by the user's UID and GID.
- Specify the `anonymous` UID and GID as the value `-2`.

14.2.6 Starting the NFS Server Subsystem

Now that we have all the various files and configuration information properly specified, tuning options set, and file systems built and ready for export, we can actually start the NFS server subsystem and hope that the clients can access the proper file systems. This is done in the normal manner:

```
# service nfs start
# service nfslock start
```

You should be able to verify that the `[nfsd]` and `[lockd]` kernel threads are shown by the `ps` command. Entering the command

```
# exportfs
```

should show the exported file systems on the server with the proper export options. You should also be able to locate processes for `rpc.mountd`, `rpc.statd`. `portmap`, and `rpc.rquotad` in the `ps` output. With the configuration parameters and required processes in place, we may proceed to mounting the server directories from a client system.

14.2.7 NFS Client Mount Parameters

The client mount parameters for NFS can control the behavior of the client and server interaction to a large degree. These parameters are shown in Table 14–1.

Several important points need to be made about NFS mount parameters and their default values.

- The default size for `rsize` and `wsize` (1,024 bytes) will almost certainly have an adverse impact on NFS performance. There are a number of points where these settings can impact the performance of NFS servers and clients: file system read and write bandwidth, network utilization, and physical read/modify/write behavior (if the NFS block size matches file system fragment size). In general, the 8-KB maximum for NFS PV2 is a minimum size for efficiency, with a 32,768-byte block being a better option if supported by the NFS PV3 implementation.
- The values for `timeo` and `retrans` implement an exponential back-off algorithm that can drastically affect client performance. This is especially true if the NFS server is busy or the network is unreliable or heavily loaded. With the default values, `timeo=7` and `retrans=3`, the first sequence will be 0.70 second (minor time-out), 1.40 second (minor time-out), 2.80 seconds (minor time-out), and 5.60 seconds (major time-out). The second sequence doubles the initial value of `timeo` and continues.
- The default behavior for a mount is *hard* and should be left that way unless the remote file system is read-only. NFS is meant to substitute for a local file system, and most applications are unaware that the data they are accessing is over a network. When *soft* mounts return errors on reads or writes, most applications behave badly (at best) and

Table 14–1 Important NFS Client Mount Parameters

NFS Mount Parameter	Default	Description
`rsize`	`rsize=1024`	The number of bytes to request in an NFS read operation (maximum for NFS PV2 is 8,192 bytes)
`wsize`	`wsize=1024`	The number of bytes to write in an NFS write operation (maximum for NFS PV2 is 8,192 bytes)
`timeo`	`timeo=7`	Tenths of a second between RPC retries (NFS minor time-outs); value is doubled each time-out until either 60 seconds or the number of minor time-outs specified in `retrans` is reached
`retrans`	`retrans=3`	The number of minor time-outs that can occur before a major NFS time-out
`soft`		If a major time-out occurs, report an error and cease retrying the operation (*This can result in data corruption! Soft is bad, bad, bad!*)
`hard`		If a major NFS time-out occurs, report an error and continue to retry; this is the default and the only setting that preserves data integrity
`intr`		Similar behavior to *soft*, do not use unless you have good reasons!
`nfsver=<2|3>`	`nfsver=2`	Allow mounting with either NFS PV2 or PV3; PV2 is the default
`tcp`		Mount the file system using TCP as the transport; support may or may not be present for the TCP transport
`udp`		Mount the file system using UDP as the transport, which is the default

can lose or corrupt data. The *hard* setting causes the client to retry the operation forever, so that a “missing” NFS server resulting from a crash or reboot will pick up the operation where it left off on returning. This is extremely important NFS behavior that is little understood.

- The default NFS transport is UDP. This transport is nonconnection oriented and will perform better than TCP in stable, reliable networks. NFS implements its own error recovery mechanism on top of UDP (see the `retrans` and `timeo` parameters), so unless the mount is over a WAN connection, the preferred transport is UDP. Some proprietary UNIX operating systems, like Solaris, default to using TCP as the NFS transport.

- The default mount protocol is NFS PV2, which is less efficient than NFS PV3. The NFS PV3 protocol supports large files (more than 2 GB), larger block sizes (up to 32 KB), and safe asynchronous writes (acknowledgment of physical commit to disk not required on every write operation). Using NFS PV3 is extremely important to overall NFS performance.

The mount parameters for NFS file systems, whether in `/etc/fstab` or specified in automounter maps, need to have the following options specified for best performance:

```
rsize=32768,wsize=32768,nfsvers=3
```

14.2.8 Using `autofs` on NFS Clients

Maintaining the NFS mounts in a `/etc/fstab` file on multiple systems can become an exercise in frustration. A more centralized method of maintaining this information is to use the NIS subsystem to distribute the `auto.master` map and other submaps to the client systems. The NFS client system may then be configured to use the `autofs` subsystem to mount file systems automatically when required and unmount them when idle.

Unmounting the NFS file system when idle removes the direct dependency between the client and server that can cause retries if the remote file system's server becomes unavailable. This behavior is supported by "indirect" maps, unlike the direct map entries that simulate an NFS mount created by an entry in the `/etc/fstab` file. A side benefit is that indirect mounts can be changed without rebooting an NFS client using them. This level of indirection allows the system manager to change the location (server, directory path, and so on) of the data without affecting the client's use of the data.

An `auto.master` map from either NIS or a local file provides the names of pseudo directories that are managed by a specific map and the `autofs` subsystem on the client. The settings in the `/etc/nsswitch.conf` file for the `automount` service define the order of precedence for locating the map information—for example,

```
automount: nis files
```

Thus, `autofs` will examine the NIS information for the maps first, then local files located in the `/etc` directory. A simple example of a master map would be

```
/data       auto.data        nfsvers=3,rsize=32768,wsize=32768
/home       auto.home        nfsvers=3,rsize=32768,wsize=32768
```

Notice that the mount options in the master map, which override the submap options, are in the third field in the master map data. This is different from the options in the submaps, which are the second field in the data. I use the convention `auto.<directory>` to name the map that instructs `autofs` how to mount the information under the pseudo directory named *<directory>*. An example `auto.data` map might contain the entries:

```
scratch          fs01:/raid0/scratch
proj             fs02:/raid5/projects
bins             fs02:/raid5/executables
```

Any time a client system references a path, such as `/data/proj`, the `autofs` subsystem will mount the associated server directory at the appropriate location. When the directory is no longer in use, it is dismounted after a time-out.

Once the `autofs` configuration in `/etc/nsswitch.conf` is complete, and the NIS or local map files are present, the `autofs` subsystem may be started with

```
# chkconfig autofs on
# service autofs start
```

You should be able to `cd` to one of the directories controlled by the indirect maps to verify their proper operation. I do not cover direct maps here, because I consider them to be evil.

14.2.9 NFS Summary

Although NFS is not a parallel file system, its ease of implementation can make it a choice for smaller clusters. With proper tuning and attention to the NFS data path between client and server, NFS can offer adequate performance. Certainly, most UNIX or Linux system administrators have experience with implementing NFS, but tuning this network file system is not always trivial or obvious. ("You can tune a file system, but you can't tune a fish," as stated by the UNIX `man` page for `tunefs`.)

There are a number of areas that can be tuned on an NFS server, and the examples that we covered in this section illustrate a general method for setting Linux kernel parameters. The use of the `sysctl` command and its configuration file `/etc/sysctl.conf` allow any parameters to be modified at boot time. A cautious approach to tuning is to set the parameter values temporarily with either `/proc` or the `sysctl` command before permanently committing them.

In general, NFS mounts in the `/etc/fstab` file, and direct mounts using `autofs` are a bad idea because they create direct dependencies between the client systems and the server that is mounted. If you decide to service the NFS server, the client RPC retry behavior for hard mounts will preserve data integrity, but will cause infinite retries until the server returns. It is better to let `autofs` unmount unused file systems and avoid the direct dependencies.

14.3 A Survey of Some Open-Source Parallel File Systems

This section provides an overview of some of the available parallel file systems. This is not an exhaustive list, just some of the options that I have come across in my research and experience. I cover several open-source parallel file systems, and one commercially available one in the overviews.

A common issue with open-source parallel file systems is that they may require kernel reconfiguration expertise to install and use. Some of the functionality may be available in the form of premodified kernels and modules, precompiled kernel modules for a particular kernel version, or a do-it-yourself kit in the form of patches. If you are not running a production environment, or are open to experimentation and have the necessary kernel hacking skills, several of the file systems covered here might be options for you. Otherwise, a commercially available and supported solution is a better choice.

Many of the possible choices will require you to work with a minimum amount of documentation, other than the source code. This is a normal occurrence in the Linux and open-source world. The projects that we are examining are still works in progress, although some of the functionality is being included in the next release of Linux version 2.6 of the kernel.

You should note that most of the parallel file systems we are examining use TCP/IP as a transport. This means that the parallel file systems may be run over Ethernet, but there are other choices. Recall that the HSIs like Myrinet and Quadrics allow transport of TCP/IP along with the other message formats needed by MPI or OpenMP. This means that very fast transport is available for the parallel file system, provided that your cluster has an HSI in place.

14.3.1 The Parallel Virtual File System (PVFS)

The first cluster file system we will examine is the parallel virtual file system, or PVFS. There are currently two versions of this file system, PVFS1 and PVFS2. The newest version, PVFS2, requires the Linux 2.6 kernel, so for the purposes of this introduction, we will concentrate on PVFS1. Both versions are currently being maintained.

The PVFS1 software is available from `http://www.parl.clemson.edu/pvfs` and the PVFS2 software is available from `http://www.pvfs.org/pvfs2`. I chose the PVFS1 software as the first example, because it requires no modifications to the kernel to install, just compilation of the sources. The architecture of PVFS is shown in Figure 14–4.

Figure 14–4 shows an application talking to PVFS through a dynamically loaded kernel module that is supplied as part of the PVFS software. This module is provided to allow normal interaction between the application, the Linux virtual file system (VFS) layer, and the PVFS daemon, which performs I/O on behalf of the application. Applications may be recompiled to use a PVFS library that communicates directly with the meta-data server and the I/O daemons on the server nodes, thus eliminating the kernel communication (and one data copy).

The server nodes store data in an existing physical file system, such as `ext3`, to take advantage of existing journaling and recovery mechanisms. The PVFS `iod` process, or I/O daemon, performs file operations on behalf of the PVFS client system. These operations take place on files striped across the PVFS storage servers in a user-defined manner.

You may download the source for PVFS1 from `ftp://ftp.parl.clemson.edu/pub/pvfs` and perform the following steps:

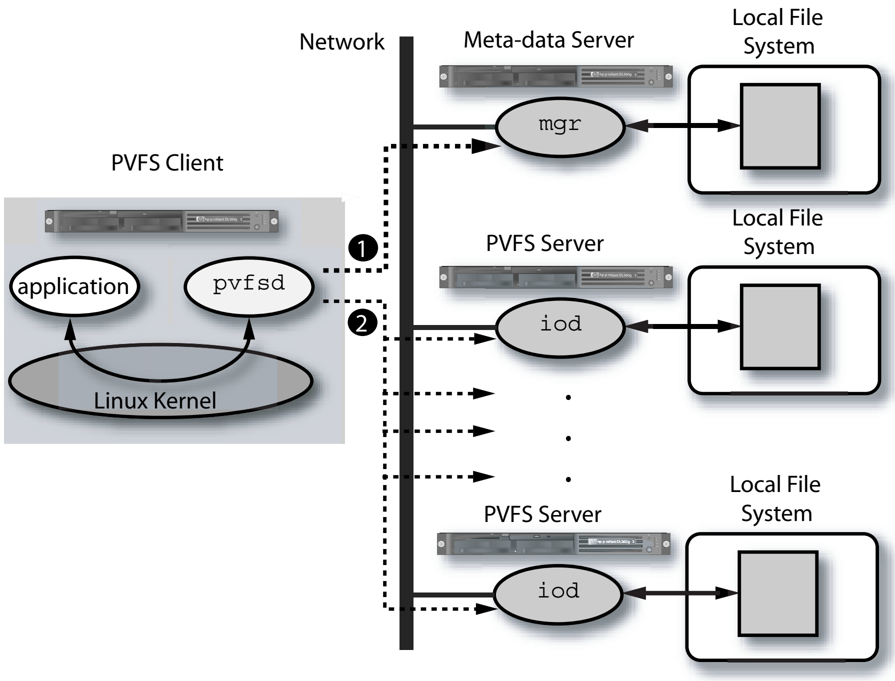

Figure 14–4 Architecture of the PVFS

```
# tar xvzf pvfs-1.6.2.tgz
# cd pvfs-1.6.2
# ./configure
# make
# make install
```

The installation unpacks `/usr/local/sbin/iod`, `/usr/local/sbin/mgr`, the application library `/usr/local/lib/pvfs.a`, and a host of utilities in `/usr/local/bin`, such as `pvfs-ping` and `pvfs-mkdir`. The default installation includes auxilliary information, like the `man` pages and header files for the development library under `/usr/include`. The entire operation takes approximately three minutes.

If you wish to use the kernel module to allow access to the file system for existing programs, you may download the `pvfs-kernel-1.6.2-linux-2.4.tgz` file and build the contents:

```
# tar xvzf pvfs-kernel-1.6.2-linux-2.4.tgz
# cd pvfs-kernel-1.6.2-linux-2.4
# ./configure
# make
# make install
# cp mount.pvfs /sbin/mount.pvfs
# mkdir /lib/modules/2.4.18-24.8.0/kernel/fs/pvfs
# cp pvfs.o /lib/modules/2.4.18-24.8.0/kernel/fs/pvfs
# depmod -a
```

Let's install the software package on five nodes: one meta-data server and four data servers. For this example, let's install the kernel package on a single node that is to be the PVFS client. For the meta-data server, install and configure the software, then started the meta-data manager daemon, `mgr`:

```
# mkdir /pvfs-meta
# cd /pvfs-meta
# mkmgrconf
This script will make the .iodtab and .pvfsdir files
in the metadata directory of a PVFS file system.
Enter the root directory (metadata directory):
/pvfs-meta
Enter the user id of directory:
root
Enter the group id of directory:
root
Enter the mode of the root directory:
777
Enter the hostname that will run the manager:
cs01
Searching for host...success
Enter the port number on the host for manager:
(Port number 3000 is the default) <CR>
Enter the I/O nodes: (can use form node1, node2, ... or
nodename{#-#,#,#})
cs02, cs03, cs04, cs05
Searching for hosts...success
I/O nodes: cs02 cs03 cs04 cs05
Enter the port number for the iods:
(Port number 7000 is the default) <CR>
Done!
# /usr/local/sbin/mgr
```

On each of the data servers, configure and install the I/O server daemon, `iod`:

```
# cp /tmp/pvfs-1.6.2/system/iod.conf /etc/iod.conf
# mkdir /pvfs-data
# chmod 700 /pvfs-data
# chown nobody.nobody /pvfs-data
# /usr/local/sbin/iod
```

Then test to make sure that all the pieces are functioning. A command that is very useful is the `pvfs-ping` command, which contacts all the participating systems:

```
# /usr/local/bin/pvfs-ping -h hpepc1 -f /pvfs-meta
mgr (cs01:3000) is responding.
iod 0 (192.168.0.102:7000) is responding.
iod 1 (192.168.0.103:7000) is responding.
iod 2 (192.168.0.104:7000) is responding.
iod 3 (192.168.0.105:7000) is responding.
pvfs file system /pvfs-meta is fully operational.
```

Wahoo! We can now go and mount the file system on a client. There are a few minor steps before we can mount the file system on the client. First we need to create the /etc/pvfstab file, which contains mounting information:

```
cs01:/pvfs-meta     /mnt/pvfs  pvfs       port=3000       0 0
```

Next, we can actually mount the parallel file system, which involves installing the kernel module and starting the pvfsd to handle communication with "normal" programs and commands:

```
# insmod pvfs
# /usr/local/sbin/pvfsd
# mkdir /mnt/pvfs
# /sbin/mount.pvfs cs01:/pvfs-meta /mnt/pvfs
# /usr/local/bin/pvfs-statfs /mnt/pvfs
blksz = 65536, total (bytes) = 388307550208,                \
       free (bytes) = 11486298112
```

We are now ready to start using the file system. Let's load a copy of iozone, which is a handy file system performance tool, available from http://www.iozone.org, and start running tests. Things should work just fine within the limits of your test network and systems. For performance and scaling information, see the PVFS Web site.

The version of PVFS that we are testing, PVFS1, although it is still maintained, is likely to be superseded by the PVFS2 version when the Linux 2.6 kernel becomes generally available. There have been a number of noticeable improvements in the tools for configuring and managing the file system components, along with more documentation. Download and try PVFS1 for your cluster, but keep your eyes on PVFS2.

14.3.2 The Open Global File System (OpenGFS)

The Open Global File System, also known as OpenGFS and OGFS, is another file system that provides direct client access to shared storage resources. The systems that use OpenGFS access the storage devices through a Fibre-Channel SAN (Figure 14–5). Some of the features of OpenGFS are

- Individual journals for separate systems
- Pluggable locking services based on LAN connections

- Management of cluster "participation" to manage journaling and locking actions
- "Fencing" of storage access to prevent corruption by failed nodes[1]
- Support for volume manager aggregation of multiple devices, with volume managers like the enterprise volume management system (EVMS; available at `http://evms.sourceforge.net/docs.php`)

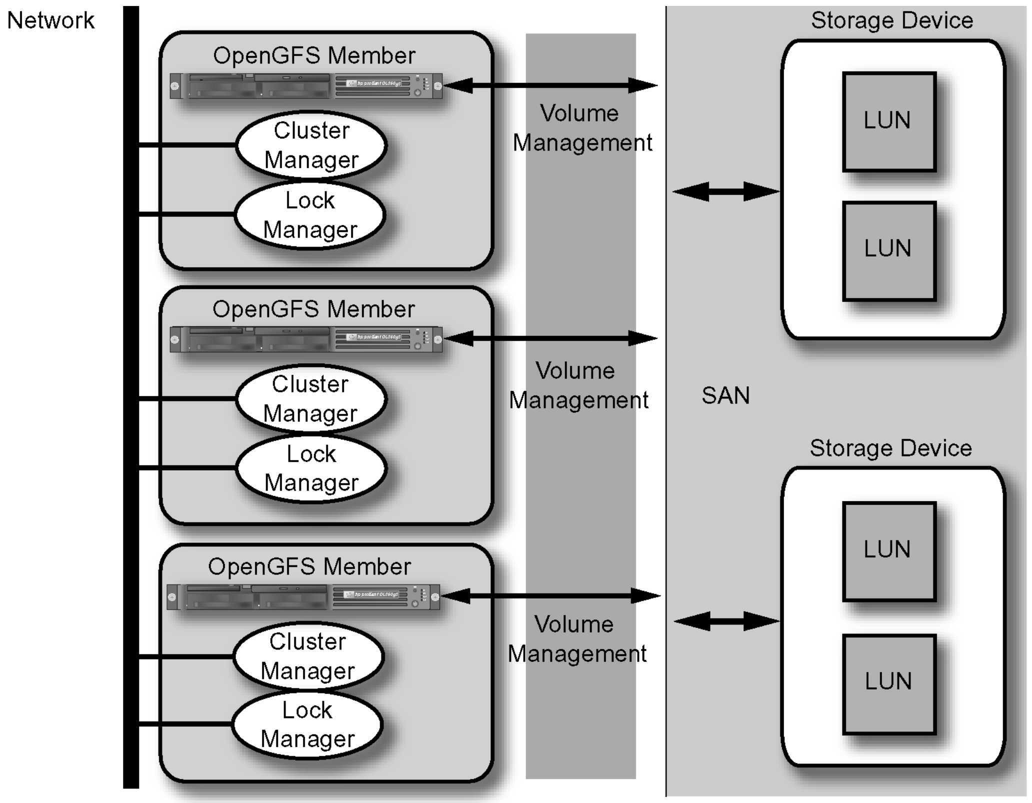

Figure 14–5 OpenGFS architecture

The OpenGFS software requires compilation and patching of the kernel as part of the installation process. This means that only certain versions of the kernel are acceptable for use

1. If you have not heard of STOMITH, then this is a good time to learn that it stands for "shoot the other machine in the head." This is a common, somewhat humorous, term in parallel file systems, and it refers to the collective cluster's ability to modify access for a given system to the SAN devices. A misbehaving cluster member is voted "off the island" by its partners.

with OpenGFS. If you choose to use EVMS in conjunction with OpenGFS, additional patching is required. The current release version, as of this writing, is 0.3.0.

Because the level of modification to the kernel requires specific software development and kernel-patching expertise, it is beyond the scope of this discussion. If you are interested in experimenting with OpenGFS, there is plenty of documentation available for obtaining the software, compiling it, patching the kernel, and installing it (go to `http://opengfs.sourceforge.net/docs.php`). Among the HOW-TO information on this site are documents on using OpenGFS with iSCSI, IEEE-1394 (Firewire), and Fibre-Channel storage devices.

14.3.3 The Lustre File System

The name *Lustre* is a melding of the words *Linux* and *cluster*, with a tweak of the last two letters. Although the vision of the final Lustre file system is far from realized, version 1.0 (actually version 1.04 as of this writing) is now available for download and demonstration purposes. Information on Lustre is available from `http://www.lustre.org`. Before we actually try Lustre, let's enumerate some of the target design attributes:

- Object-oriented storage access (hundreds of object storage servers [OSSs])
- Scalable to large numbers of clients (tens of thousands)
- Scalable to huge amounts of storage (petabytes)
- Scalable to large amounts of aggregated I/O (hundreds of gigabytes per second)
- Manageable and immune to failures

As you can tell from the descriptive items, the word *scalable* comes up a lot with reference to Lustre. So do the words *huge* and *large*. Although the end goal is a parallel file system that scales for use in very large clusters, there is nothing to prevent us from using Lustre for our own, albeit, smaller clusters.

Storage in Lustre is allocated at a higher level than the physical hardware blocks that are associated with disk sectors. File objects may be spread over multiple "object storage servers," with a variety of flexible allocation and access schemes. The internal architecture of Lustre is quite complex, and to start even a high-level description we should examine a diagram. Figure 14–6 shows a high-level collection of Lustre components.

Lustre clients access meta-data servers (MDSs) and OSSs using a message-passing library called "Portals" from Sandia National Laboratories (see `http://www.sandiaportals.org` and `http://sourceforge.net/projects/sandiaportals`.) This library, among other things, provides a network abstraction layer that allows message passing over a variety of physical transports, including TCP, Infiniband, SCI, Myrinet, and Quadrics ELAN3 and ELAN4. The specification includes provisions for RDMA and operating system bypass to improve efficiency.

Lustre clients create files from objects allocated on multiple OSS systems. The file contents are kept on the OSS systems, but the meta-data is stored and maintained by a separate

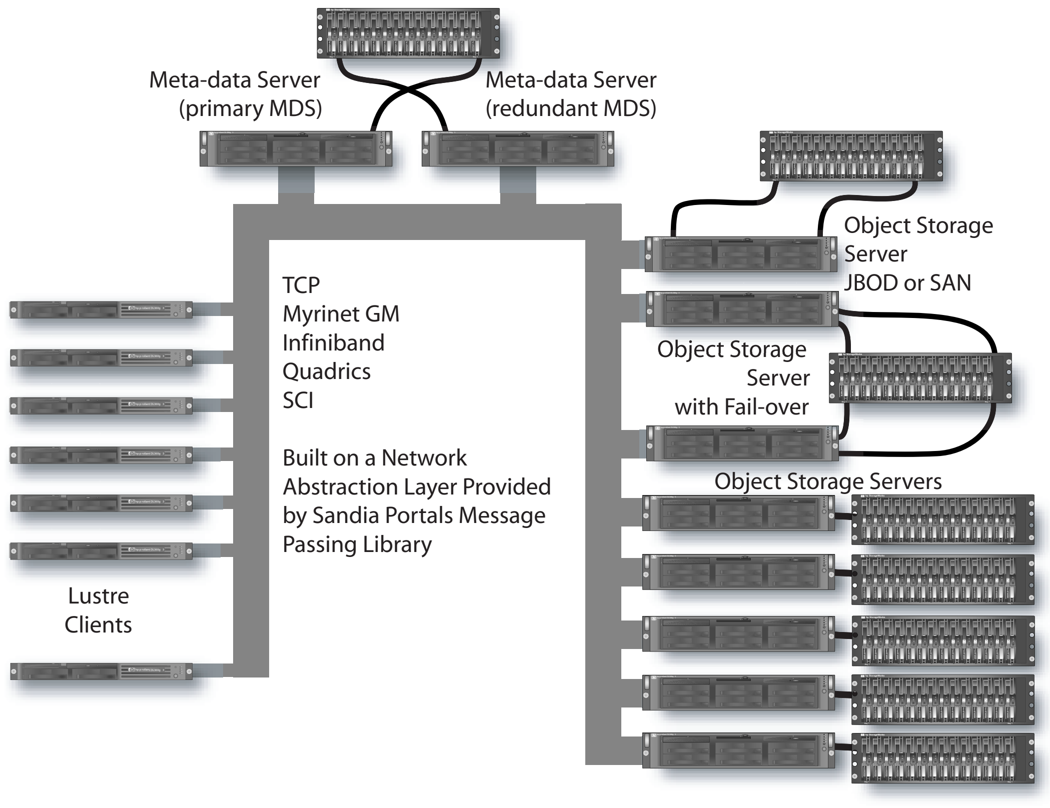

Figure 14–6 Lustre high-level architecture

meta-data server. The later versions of Lustre are to allow many MDS systems to exist, to scale the meta-data access, but the current implementation allows for only one MDS (and possibly a fail-over copy, but the documentation is unclear about this).

Lustre OSS systems use the physical file systems of a Linux host to store the file objects. This provides journaling and other recovery mechanisms that would need to be implemented "from scratch" if a raw file system is used. It is entirely possible that manufacturers will create special OSS hardware with the Lustre behavior captured in firmware, thus providing an OSS appliance. This does not appear to have happened yet.

To try Lustre, you can download two packages from the Web site:

```
kernel-smp-2.4.20-28.9_lustre.1.0.4.i586.rpm
lustre-lite-utils-2.4.20-28.9_lustre.1.0.4.i586.rpm
```

There is also a kernel source package and an all-inclusive package that contains the kernel *and* the utility sources. The precompiled packages are easy enough for a quick trial. As you can see by the names, the packages are based on a version of the kernel from Red Hat 9.0. The two packages and the HOW-TO at `https://wiki.clusterfs.com/lustre/lustrehowto` can help us get started.

First, install the two packages. Make sure that you use the `install` option to the `rpm` command, *not* the `update` option, or you might replace your current kernel. We want to install the software in addition to the current kernel:

```
# rpm -ivh kernel-smp-2.4.20-28.9_lustre.1.0.4.i586.rpm     \
        lustre-lite-utils-2.4.20-28.9_lustre.1.04.i586.rpm
```

This will place the kernel in `/boot`, update the bootloader menu, and install the required modules in `/lib/modules/2.4.20-28.9_lustre.1.0.4smp`. When you reboot your system, you should see a menu item for `2.4.20-28.9_lustre.1.0.4smp`. Select it and continue booting the system.

The configuration information for Lustre is kept in a single file that is used by all members of the cluster. This file is in XML format and is produced and operated on by three main configuration utilities in `/usr/sbin`: `lmc`, `lconf`, and `lctl`. There is a rather large PDF document (422 pages) describing the architecture and setup of Lustre at `http://www.lustre.org/docs/lustre.pdf`. It is not necessary to understand the whole picture to try out Lustre on your system. There are example scripts that will create a client, OSS, and MDS on the same system, using the loop-back file system.

The demonstration scripts are located in `/usr/lib/lustre/examples`. There are two possible demonstrations. The one we are using activates `local.sh` as part of the configuration:

```
# NAME=local.sh; ./llmount.sh
loading module: portals srcdir None devdir libcfs
loading module: ksocknal srcdir None devdir knals/socknal
loading module: lvfs srcdir None devdir lvfs
loading module: obdclass srcdir None devdir obdclass
loading module: ptlrpc srcdir None devdir ptlrpc
loading module: ost srcdir None devdir ost
loading module: fsfilt_ext3 srcdir None devdir lvfs
loading module: obdfilter srcdir None devdir obdfilter
loading module: mdc srcdir None devdir mdc
loading module: osc srcdir None devdir osc
loading module: lov srcdir None devdir lov
loading module: mds srcdir None devdir mds
loading module: llite srcdir None devdir llite
NETWORK: NET_localhost_tcp NET_localhost_tcp_UUID tcp cs01 988
OSD: OST_localhost OST_localhost_UUID obdfilter              \
       /tmp/ost1-cs01 200000 ext3 no 0 0
MDSDEV: mds1 mds1_UUID /tmp/mds1-cs01 ext3 no
recording clients for filesystem: FS_fsname_UUID
Recording log mds1 on mds1
```

```
OSC: OSC_cs01_OST_localhost_mds1 2a3da_lov_mds1_7fe101b48f
OST_localhost_UUID
LOV: lov_mds1 2a3da_lov_mds1_7fe101b48f mds1_UUID 0 65536 0 0\
[u'OST_localhost_UUID'] mds1
End recording log mds1 on mds1
Recording log mds1-clean on mds1
LOV: lov_mds1 2a3da_lov_mds1_7fe101b48f
OSC: OSC_cs011_OST_localhost_mds1 2a3da_lov_mds1_7fe101b48f
End recording log mds1-clean on mds1
MDSDEV: mds1 mds1_UUID /tmp/mds1-cs01 ext3 100000 no
OSC: OSC_cs01_OST_localhost_MNT_localhost
986a7_lov1_d9a476ab41 OST_localhost_UUID
LOV: lov1 986a7_lov1_d9a476ab41 mds1_UUID 0 65536 0 0       \
       [u'OST_localhost_UUID'] mds1
MDC: MDC_cs01_mds1_MNT_localhost
91f4c_MNT_localhost_bb96ccc3fb mds1_UUID
MTPT: MNT_localhost MNT_localhost_UUID /mnt/lustre mds1_    \
       UUID lov1_UUID
```

There are now a whole lot more kernel modules loaded than there were before we started. A quick look with the `lsmod` command yields

```
Module                  Size  Used by    Not tainted
loop                   11888   6  (autoclean)
llite                 385000   1
mds                   389876   2
lov                   198472   2
osc                   137024   2
mdc                   110904   1  [llite]
obdfilter             180416   1
fsfilt_ext3            25924   2
ost                    90236   1
ptlrpc                782716   0                            \
     [llite mds lov osc mdc obdfilter ost]
obdclass              558688   0                            \
     [llite mds lov osc mdc obdfilter fsfilt_ext3 ost ptlrpc]
lvfs                   27300   1                            \
     [mds obdfilter fsfilt_ext3 ptlrpc obdclass]
ksocknal               81004   5
portals               144224   1                            \
     [llite mds lov osc mdc obdfilter fsfilt_ext3 ost       \
      ptlrpc obdclass lvfs ksocknal]
```

These are the Lustre modules that are loaded in addition to the normal system modules. If you examine some of the Lustre architecture documents, you will recognize some of these module names as being associated with Lustre components. Names like `ost` and `obdclass` become familiar after a while. Notice the number of modules that use the `portals` module.

The output of the `mount` command shows that we do indeed have a Lustre file system mounted:

```
local on /mnt/lustre type lustre_lite                       \
     (rw,osc=lov1,mdc=MDC_cs01_mds1_MNT_localhost)
```

There are also a lot of files under the `/proc/fs/lustre` directories that contain state and status information for the file system components. Taking a look at the `local.xml` configuration file created by the example, we can see lots of Lustre configuration information required to start up the components (I have used bold type for the major matching XML tags in the output):

```
<?xml version='1.0' encoding='UTF-8'?>
<lustre version='2003070801'>
  <ldlm name='ldlm' uuid='ldlm_UUID'/>
  <node uuid='localhost_UUID' name='localhost'>
    <profile_ref uuidref='PROFILE_localhost_UUID'/>
    <network uuid='NET_localhost_tcp_UUID' nettype='tcp'     \
       name='NET_localhost_tcp'>
      <nid>hpepc1</nid>
      <clusterid>0</clusterid>
      <port>988</port>
    </network>
  </node>
  <profile uuid='PROFILE_localhost_UUID'                     \
       name='PROFILE_localhost'>
    <ldlm_ref uuidref='ldlm_UUID'/>
    <network_ref uuidref='NET_localhost_tcp_UUID'/>
    <mdsdev_ref uuidref='MDD_mds1_localhost_UUID'/>
    <osd_ref uuidref='SD_OST_localhost_localhost_UUID'/>
    <mountpoint_ref uuidref='MNT_localhost_UUID'/>
  </profile>
  <mds uuid='mds1_UUID' name='mds1'>
    <active_ref uuidref='MDD_mds1_localhost_UUID'/>
    <lovconfig_ref uuidref='LVCFG_lov1_UUID'/>
    <filesystem_ref uuidref='FS_fsname_UUID'/>
  </mds>
  <mdsdev uuid='MDD_mds1_localhost_UUID'                     \
       name='MDD_mds1_localhost'>
    <fstype>ext3</fstype>
    <devpath>/tmp/mds1-cs01</devpath>
    <autoformat>no</autoformat>
    <devsize>100000</devsize>
    <journalsize>0</journalsize>
    <inodesize>0</inodesize>
    <nspath>/mnt/mds_ns</nspath>
    <mkfsoptions>-I 128</mkfsoptions>
    <node_ref uuidref='localhost_UUID'/>
```

```
      <target_ref uuidref='mds1_UUID'/>
    </mdsdev>
    <lov uuid='lov1_UUID' stripesize='65536' stripecount='0'  \
         stripepattern='0' name='lov1'>
      <mds_ref uuidref='mds1_UUID'/>
      <obd_ref uuidref='OST_localhost_UUID'/>
    </lov>
    <lovconfig uuid='LVCFG_lov1_UUID' name='LVCFG_lov1'>
      <lov_ref uuidref='lov1_UUID'/>
    </lovconfig>
    <ost uuid='OST_localhost_UUID' name='OST_localhost'>
      <active_ref uuidref='SD_OST_localhost_localhost_UUID'/>
    </ost>
    <osd osdtype='obdfilter'                                  \
        uuid='SD_OST_localhost_localhost_UUID'                \
        name='OSD_OST_localhost_localhost'>
      <target_ref uuidref='OST_localhost_UUID'/>
      <node_ref uuidref='localhost_UUID'/>
      <fstype>ext3</fstype>
      <devpath>/tmp/ost1-cs01</devpath>
      <autoformat>no</autoformat>
      <devsize>200000</devsize>
      <journalsize>0</journalsize>
      <inodesize>0</inodesize>
      <nspath>/mnt/ost_ns</nspath>
    </osd>
    <filesystem uuid='FS_fsname_UUID' name='FS_fsname'>
      <mds_ref uuidref='mds1_UUID'/>
      <obd_ref uuidref='lov1_UUID'/>
    </filesystem>
    <mountpoint uuid='MNT_localhost_UUID' name='MNT_localhost'>
      <filesystem_ref uuidref='FS_fsname_UUID'/>
      <path>/mnt/lustre</path>
    </mountpoint>
  </lustre>
```

I believe we are seeing the future of configuration information, for better or worse. Although XML might make a good, easy-to-parse, universal "minilanguage" for software configuration, it certainly lacks human readability. If this means we get nifty tools to create the configuration, and nothing ever goes wrong to force us to read this stuff directly, I guess I will be happy.

The example script calls `lmc` to operate on the XML configuration file, followed by `lconf` to load the modules and activate the components. As you can see from the configuration, an MDS and object storage target (OST) are created along with a logical volume (LOV) that is mounted by the client "node." Although this example all takes place on the local host, it is possible to experiment and actually distribute the components onto other systems in the network.

Cleaning up after ourselves, we perform the following:

```
# NAME=local.sh;./llmountcleanup.sh
MTPT: MNT_localhost MNT_localhost_UUID /mnt/lustre mds1_UUID
lov1_UUID
MDC: MDC_cs01_mds1_MNT_localhost
21457_MNT_localhost_4aafe7d139
LOV: lov1 d0db0_lov1_aa2c6e8c14
OSC: OSC_cs01_OST_localhost_MNT_localhost
d0db0_lov1_aa2c6e8c14
MDSDEV: mds1 mds1_UUID
OSD: OST_localhost OST_localhost_UUID
NETWORK: NET_localhost_tcp NET_localhost_tcp_UUID tcp cs01 988
killing process 3166
removing stale pidfile: /var/run/acceptor-988.pid
unloading module: llite
unloading module: mdc
unloading module: lov
unloading module: osc
unloading module: fsfilt_ext3
unloading module: mds
unloading module: obdfilter
unloading module: ost
unloading module: ptlrpc
unloading module: obdclass
unloading module: lvfs
unloading module: ksocknal
unloading module: portals
lconf DONE
```

To try the multinode example, you will need to install the Lustre kernel on multiple nodes, reboot, and feed a common XML configuration file to the `lconf` command on each node. Because this is relatively intrusive, we will not attempt the nonlocal demonstration here. The multinode example is left as an exercise for you, when you feel ambitious.

One of the interesting things about Lustre is the abstraction of the file access. There are what might be called "pseudo" drivers that implement access patterns (striping, RAID, and so on) across the multiple OSS devices. This architecture appears to be substantially different from some of the other parallel file systems that we will encounter in terms of the "back-end" storage and SAN configurations. The I/O requests (object accesses) may be spread across multiple SANs or physically attached devices on the OSS systems to produce the promised level of scaling.

Lustre development continues today, so many of the architectural promises have not yet been realized. The level of support for this project from major corporations like Hewlett-Packard, however, shows a potential for realizing the goals, and the project is on a stepwise approach to the full vision (what we have today is "Lustre Lite"). We need to keep our eyes on Lustre as it matures and extends its features.

14.4 Commercially Available Cluster File Systems

In addition to the open-source cluster file systems that are readily available for free, there are a number of commercial parallel file system implementations on the market. The primary advantage of the commercial packages is the available support, which can include bug fixes and updates to functionality. Although I do not have time or space to delve deeply into all these products, three will serve as a representative sample of what is available.

There is a set of common requirements that you will find in all parallel file systems used in Linux clusters. The first requirement is POSIX semantics for the file system, which allows UNIX and Linux applications to use the file system without code modifications. The set of operations, the behavior, and the data interfaces are specified in the POSIX standard. For database use, you will find that the file system needs to provide some form of direct-access mode, in which the database software can read and write data blocks without using the Linux system's buffer or page cache. The database software performs this caching itself and removes the associated overhead.

For use with Oracle 9i RAC installations, a parallel file system allows sharing of the Oracle home directory with the application binaries and associated log and configuration files. This means a single software installation (instance) instead of one install per node participating in the cluster. This is one of the general benefits of using a file system of this type for clusters—the shared data *and* shared client participation in the file system can mean less system administration overhead.

14.4.1 Red Hat Global File System (GFS)

Red Hat has recently completed the purchase of Sistina Software Inc. and their GFS technology (information at `http://www.sistina.com`). On a historical note, the OpenGFS software described earlier is built on the last open-source version of GFS before the developers founded Sistina Inc. Red Hat offers a number of cluster "applications" for their Linux offering, including high availability and IP load balancing, and GFS adds a parallel cluster file system to that capability. The Sistina Web site mentions that the GFS software will be open source soon.

The GFS product provides direct I/O capabilities for database applications as well as POSIX-compliant behavior for use with existing UNIX and Linux applications. Both Red Hat and SUSE Linux are supported. The Red Hat Linux versions supported as of April 2004 are RHEL 3.0, RHAS 2.1, and Red Hat version 9.0. The SUSE SLES 8 release is also supported. GFS supports up to 256 nodes in the file system cluster, all with equal (sometimes referred to as *symmetric*) access to the shared storage.

As with other parallel cluster file systems, GFS assumes the ability to share storage devices between the members of the cluster. This implies, but does not require, a SAN "behind" the cluster members to allow shared access to the storage. For example, it is quite possible to use point-to-point Fibre-Channel connections in conjunction with dual-controller RAID arrays and Linux multipath I/O to share storage devices without implementing a switched SAN fabric.

Another way to extend the “reach” of expensive SAN storage is to use a SAN switch with iSCSI or other network protocol ports. The Fibre-Channel switch translates IP-encapsulated SCSI commands, delivered via the iSCSI protocol, into the native Fibre-Channel frames. High-performance systems may access the SAN via the switch Fibre-Channel fabric, and other, potentially more numerous systems may access the storage via GbE.

In addition to shared access to storage in an attached SAN fabric, Red Hat provides GFS server systems the ability to access a storage device via a local GFS network block device (GNBD), which accesses devices provided by a GNBD server. This type of interface on Linux (there are several, including the network block device [nbd] interface) typically provides a kernel module that translates direct device accesses on the local system to the proper network-encapsulated protocol. The GNBD interface allows extending storage device access to the server systems via a gigabit (or other) low-cost Ethernet network transport. This arrangement is shown in Figure 14–7.

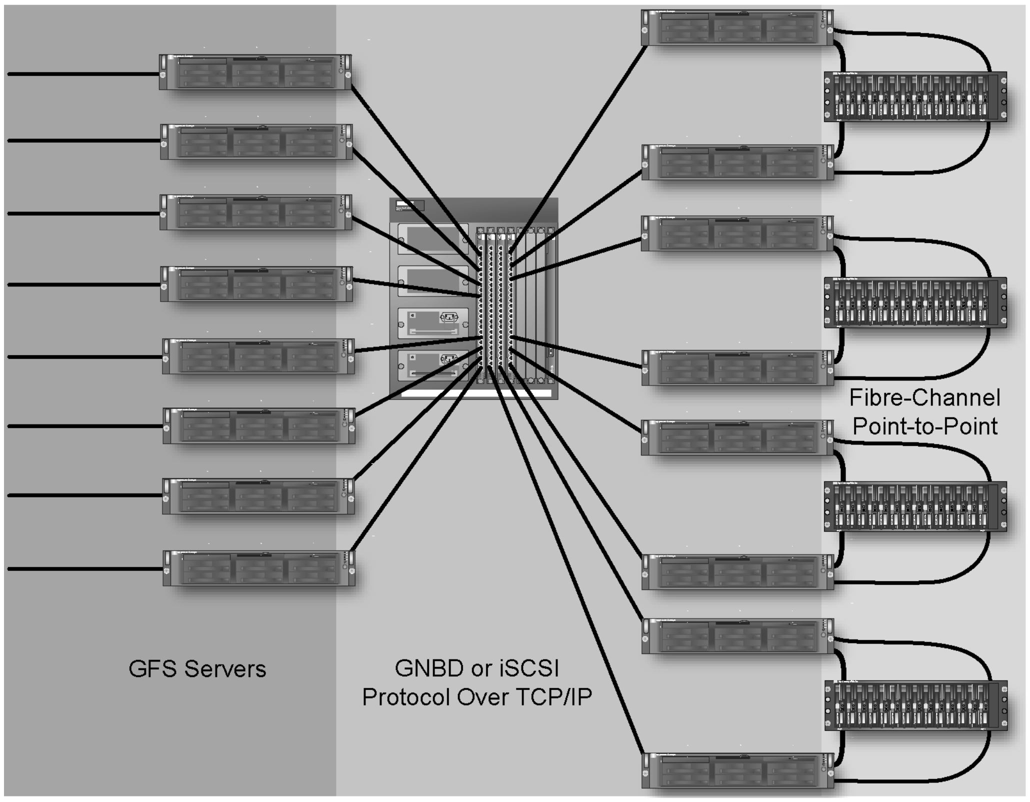

Figure 14–7 GNBD and iSCSI device access to Red Hat GFS

This approach provides more fan-out at a lower cost than a switched SAN, but at a potentially lower performance. A key factor is the ability to eliminate expensive back-end SAN switches by using direct attach Fibre-Channel storage and either GNBD or iSCSI over lower cost GbE, thus decreasing costs. An entire book could be written on the design issues and implementation approach for this type of solution—I am hoping that somebody will write it soon. We will stop here before we get any deeper.

14.4.2 The PolyServe Matrix File System

The Matrix Server product is a commercially available cluster file system from PolyServe Inc. The product runs on either Microsoft Windows or Linux systems and provides a high availability, no SPOF file-serving capability for a number of applications, including Oracle databases. Red Hat and SUSE Linux distributions are supported by the software. See `http://www.polyserve.com` for specific information.

Access to the file system is fully symmetric across all cluster members, and is managed by a distributed lock manager. Meta-data access is also distributed across the cluster members. The Matrix Server product assumes shared storage in a SAN environment. The high-level hardware architecture is shown in Figure 14–8.

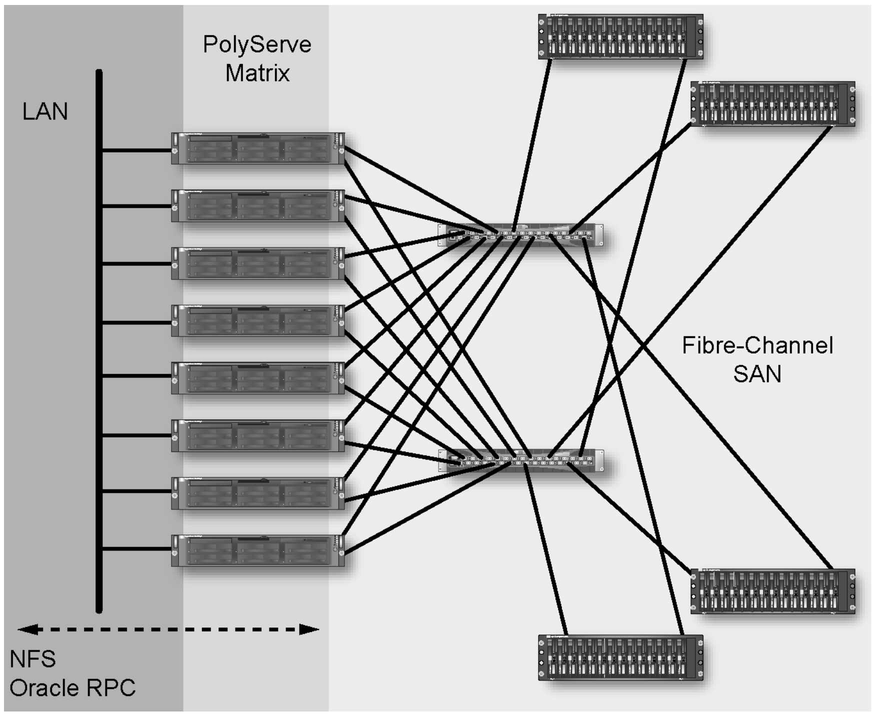

Figure 14–8 PolyServe Matrix architecture

The file system is fully journaled and provides POSIX semantics to Linux applications, so they do not need to be specially adapted to cluster environments. The high-availability package provides a number of features, including multiple fail-over modes, standby network interfaces, support for multipath I/O, and on-line addition or removal of cluster members.

An interesting management feature provided by the PolyServe file system is the concept of “context-dependent symbolic links” or CDSLs. A CDSL allows node-specific access to a common file name (for example, log files or configuration files like `sqlnet.ora`) in the file system. The CDSLs allow the control of which files have one instance and are shared by all members of the cluster, and which files are available on a node-specific basis. The PolyServe file system allows creating CDSLs based on the node’s host name.

The PolyServe file system may be exported via NFS or made available to database clients with Oracle RPC, via the network. In this situation, the PolyServe nodes also become NFS or Oracle 9i RAC servers to *their* clients, in addition to participating in the cluster file system. Their common, consistent view of the file system data is available in parallel to NFS or Oracle clients. Other applications, like Web serving, Java application serving, and so forth, also benefit from exporting a single, consistent instance of the data in the cluster file system.

14.4.3 Oracle Cluster File System (OCFS)

The Oracle cluster file system (OCFS) is designed to eliminate the need for raw file access in Linux implementations of Oracle 9i RAC. The OCFS software is available for download free, and it is open source. It is intended only for supporting the requirements of the Oracle RAC software and is not a general-purpose cluster file system.

Because the Oracle database software manages the required locking and concurrency issues associated with data consistency, this functionality is not provided by the OCFS implementation. Other restrictions exist with regard to performing I/O to the OCFS, so its use should be limited to Oracle 9i RAC implementations only. It can help reduce the cost of implementing an Oracle 9i RAC cluster by eliminating the need to purchase extra cluster file system software.

Although OCFS removes the requirement to use raw partitions to contain the database, there are other side effects of which you should be aware. For this reason, you should investigate the system administration consequences, such as no shared Oracle home directory (implying multiple instances of log files and other information), before committing to OCFS as the file system for your database cluster.

14.5 Cluster File System Summary

We examined a number of cluster file systems and their architectures in this chapter. Although there is a lot of promise in PVFS, OpenGFS, Lustre, and the plethora of other open-source cluster file systems, the current developmental state of these file systems may not be appropriate for production environments. There are many more parallel file systems available from the open-source and research communities, and more are becoming available every day.

An alternative to the experimental nature of some of the parallel open-source file systems is good old NFS. We are mostly familiar with NFS and its characteristics, and should have a fairly good instinct about whether it is up to the challenge presented by our cluster. Like the other open-source cluster file systems, the price is right, but NFS has the advantage of being thoroughly wrung out by a large number of users who are familiar with it.

The commercially available file systems, like the Red Hat GFS and the PolyServe Matrix file system, are potentially viable alternatives to the open-source file systems for production environments that require scaling, resiliency, and availability of professional support. The support and feature set from the commercial cluster file systems makes them attractive for database, Web-serving, and high-availability cluster file system applications. It is unclear whether these file systems will scale into the many hundreds or thousands of clients.

Which choice you make for providing data to your cluster will depend on your ability to deal with software development issues, scaling, implementation details, and the availability of support. I recommend a thorough investigation before selecting a cluster file system for your environment. The good thing is that we have a wide range of choices.