CITI - ARC Joint Project Status Report

Center for Information Technology Integration
October 26, 2006

This memorandum reports on the status of a joint project between IBM, Almaden and CITI, University of Michigan under Agreement No. A0550295. The Statement of Work specifies three tasks:

- Locking, Delegations, and Shares for Cluster File Systems
- Replication and Migration
- pNFS

Task descriptions are shown below in gray text. Reporting is in-line with the tasks, shown in black text.

Project status is maintained in a wiki at **`wiki.linux-nfs.org`**. This report largely summarizes and refers to the wiki. Wiki contents as of this writing—lightly touched-up for spelling, grammar, and appearance—are attached as appendices.

Locking, Delegations, and Shares for Cluster File Systems

Correct operation of an NFSv4 server and a cluster file system with multiple servers requires some file operations to be synchronized with the cluster file system. Specifically, byte range locks, open share modes, and delegations must be passed to the file system through extensions to the VFS interface (or other means).

This task involves the definition and implementation of the following:

- Uniform interfaces for different lock managers (POSIX, LOCKD, NFSD) that invoke file system-provided VFS lock(). (This may require a new lock() interface that returns the conflicting lock). This involves defining a fair-queue scheme for handling blocking requests and providing a callback interface for reporting lock availability.

We coalesced the locking requirements of NFSD, LOCKD, and the Linux local lock manager into a single interface and a single queue. This alone is not enough to provide for fair queuing, due to differences in the way local processes, LOCKD and NFSv4 manage blocked requests, so we extended the lock interface to provide for a “provisional” lock on behalf of waiting lock contenders.

Now that GFS has been added to the mainline Linux kernel, we anticipate progress in gathering consensus among Linux maintainers in accepting these extensions.

The “Cluster Coherent NFS and Byte Range Locking” wiki reports in depth on this task. Appendix I shows the wiki as of this writing.

- New interface for open share modes that need to be supplied with NFSv4 OPEN call.

Representing ACCEPT READ, ACCEPT WRITE, ACCEPT BOTH, and DENY NONE in the POSIX interface is straightforward, but DENY READ, DENY WRITE, and DENY BOTH present atomicity problems in the Linux POSIX interface. We are wrestling with a range of options.

The “Cluster Coherent NFSv4 and Share Reservations” wiki reports in depth on this task. Appendix II shows the wiki as of this writing.

- New interface for acquiring file delegations before they are granted to the clients including callbacks for revocation.

Issues in managing delegations for cluster file systems have been identified, and an interface has been proposed.

The “Cluster Coherent NFSv4 and Delegations” wiki reports in depth on this task. Appendix III shows the wiki as of this writing.

Task 2: Replication and Migration

NFSv4 has a means of providing file system replication and migration services through the recommended `fs_locations` attribute. Allowing clients to move seamlessly from one NFSv4 server to another provides fault tolerance through replication, load balancing, migration in a global namespace, and flexible resource allocation. While replication is intended for read-only data, certain file systems like GPFS that provide a cluster solution can exploit this feature for read-write data as well. This task involves complete implementation of the optional features of RFC 3530 that relate to replication and migration.

This includes the following:

- Detection and handling of replication and migration events originating from an NFSv4 server at the client. This includes support for volatile file handles, file handle recovery on expiration, following of referrals and state recovery or expiration on migration.

NFSv4 uses the NFS4ERR_MOVED indicator for referrals, replication, and migration. The Linux NFSv4 client supports referrals and (read-only) replication by selecting a server on the referral list and reissuing the request. Migration is complicated by the need to reestablish client state on the target server.

Progress has been made in client recovery for migration, but state recovery depends on a state migration solution. Conventional client recovery can be improved in a cluster file system, so this task depends on Task 4.

- Server support for `fs_locations` and generation of migration and referral events on a trigger, possibly the exports file. This trigger can help a performance monitor to do load balancing, and an administrator to move NFSv4 clients between NFSv4 servers that have a common backend.

The Linux NFSv4 client can service referrals, but as above, we have made only minimal progress on triggering migration while we await progress on Task 4.

- Definition of a server interface for obtaining `fs_locations` information for a given file system, either from an external source (e.g., LDAP database) or through the file system.

The Linux NFSv4 client uses `/etc/exports` directly or LDAP for referrals and (read-only) replication. We will use the same mechanism for migration. We made good progress in organizing client state associated with an FSID to help state management.

In addition to these work items, CITI researchers working in parallel to the project developed and tested an NFSv4 read/write replication prototype with strong consistency guarantees and good performance in read-mostly environments.

- “Naming, Migration, and Replication for NFSv4,” Jiaying Zhang and Peter Honeyman, in *Proc. 5th System Administration and Network Engineering Conf.*, Delft (May 2006).
- “Replication Control in Distributed File Systems,” Jiaying Zhang and Peter Honeyman, *CITI Technical Report 04-01* (April 2004).

Recent work adapts the consistency and failure models to provide superior performance for scientific computation across high latency networks.

- “Hierarchical Replication Control in a Global File System” by Jiaying Zhang and Peter Honeyman.
  Submitted for publication; a preliminary version is available as *CITI Technical Report 06-07* (September 2006).
- “Consistent Replication for Grid Computing” by Jiaying Zhang and Peter Honeyman.
  To appear in *Proc. 4th International Workshop on Middleware for Grid Computing*, Melbourne

(November 2006); a preliminary version is available as *CITI Technical Report 06-3* (May 2006).

The CITI Technical Report series is located at **`www.citi.umich.edu/techreports`**.

Task 3: pNFS

pNFS, a new feature for a future NFSv4 minor version, can improve performance through parallel access to storage servers. We would like to experiment with an implementation and by doing so, help in the definition of this emerging protocol. Although the plan is for pNFS to support object servers and block devices, we would like to focus on the multiple NFS servers version. Parallel file I/O involves the client operating on one file by concurrently doing I/O to multiple NFSv4 servers.

This task involves the following:

- Extending the NFSv4 client to operate on a single file in parallel using specifications from the pNFS protocol extension.

A group that includes CITI and IBM developers has implemented pNFS layouts for file access. This task is rich with progress. We plan to demonstrate the work at SC06.

In addition to its partnership with IBM, CITI is developing pNFS under the sponsored of the Dept. of Energy's Advanced Simulation and Computing Program. The "CITI ASC status" wiki reports in depth on this partnership. Appendix IV shows the portion of that wiki pertaining to pNFS development as of this writing.

- Experimentation with **`fs_locations`** as file attributes for obtaining multiple locations for a file.

This subtask is obviated by development of the draft specification for pNFS, which allows device multipathing in layouts.

Task 4: State Migration

The goal of this task is to explore options in state migration. Section 8.14 of RFC 3530 states that servers involved in migration SHOULD transfer all server state from the original to the new server in a way that is transparent to the client. For cluster file systems that support persistent file handles, this case applies when the client has accessed a file system that is now moved to another server. (In a pure referral, state does not have to be transferred, but can expire). For cluster file systems to do effective load balancing using the migration capabilities in NFSv4 (as opposed to referrals), state MUST be migrated. It would be useful to support state migration in the server; whether the file system should be involved in the transfer is open to debate.

When a file system moves from a source server to a target server, the source server notifies clients with NFS4ERR_MOVED. Clients then reclaim state held on the source server by engaging in reboot recovery with the target server. For cluster file systems, server-to-server state transfer lets clients avoid the reclaim.

We redesigned state bookkeeping to ensure that state created on NFSv4 servers exporting the same cluster file system will not collide.

RFC 3530 recommends that servers save the ClientID of active clients in stable storage for reboot recovery. The Linux NFSv4 server does this by writing directly to a file system through the VFS layer. After a reboot, the new server instance reads the state from stable storage, also through the VFS.

We are rewriting this implementation to use a helper program instead of direct access to the VFS layer. The kernel communicates with this helper program using the pipefs interface, which was developed for communication between the NFS client, rpc.idmapd, and rpc.gssd.

We are also expanding the pipefs interface to allow client in-memory state to be transferred to or

from a user space daemon. The user space daemon can then support server-to-server state transfer to the corresponding daemon on a target server. We implemented a prototype of this interface, but have yet to prototype the server-to-server state transfer.

It remains to inform clients that state established with the source server is valid on the target server. The IETF NFSv4 working group is considering solutions for the NFSv4.1 protocol, but NFSv4.0 clients will not have support for this feature. We therefore need to provide Linux specific implementation support, perhaps using a mount option or a `/proc` flag. In a simpler approach, the client may continue to use the ClientID given by the source server at the risk of a ClientID collision on the target server after a migration.

The “NFS Recovery and Client Migration” discusses these issues in greater depth. Appendix V shows the wiki as of this writing.

Appendix I: Cluster Coherent NFS and Byte Range Locking

Background

For some time, exporting byte-range locks to NFS has been a challenge in Linux. Support for file system locks was designed with a process model and a local file system in mind. This suggested a synchronous interface in which a process that requests a lock is either granted the lock or suspended and placed on a queue. When the lock becomes available, a suspended process is granted the lock and allowed to proceed.

This synchronous approach breaks down when the request is made by a server, e.g., LOCKD or NFSD, where threads are a scarce resource. The synchronous approach threatens to block the server process, which constitutes a disaster.

Hence, an asynchronous lock request interface has emerged.

One of the complexities in making that transformation is a mechanism for queuing contending requests. The queue should be fair, not giving preference to one source of lock requests over another. Ideally, contending lock requests should be granted in the order in which they are issued. This argues for a single queue of pending requests, no matter their source of issue.

Cluster file systems exported with NFS introduce another layer of complexity: often they need to coordinate their locks with a lock manager in the back end. But back end coordination can be delayed, e.g., by inter-node communication, which poses another threat to a threaded server.

Finally, NFSv4 introduces one more layer of complexity: unlike NLM locks, which block, NFSv4 byte-range locks are non-blocking, so clients contending for a lock must poll. This raises the stakes for fair queuing, as a local process waiting for a lock will almost always acquire the contended lock before an NFSv4 client can.

NFSv4 Blocking Locks

Addressing fair queuing, the NFSv4 spec suggests that the server should maintain an ordered list of pending blocking locks. More broadly, queue fairness suggests that all lock requestors (local processes, LOCKD, and the NFSv4 server) should share such an ordered list.

Tasks

- Implement a shared blocking lock fair queue
- Implement the NFSv4 server fl_notify and use the fair queue

Progress

We have written patches that change the semantics of the existing file_lock->fl_block queue to integrate it with the NFSv4 server and to make it more ‘fair.’ This queue holds all blocking locks in the order in which they were requested. New blockers are added to the tail.

These patches have not been reviewed by the wider kernel community. However, the effort exposed a number of spec and implementation problems for which fixes were incorporated into the Linux kernel and the new NFSv4.1 draft.

The existing fl_block semantics

When a lock becomes available, local blocked processes are awakened and contending NLM clients are issued a “grant” callback. Contending NFSv4 clients, which do not block in anticipation of a server callback, receive no notification. Instead, they repeatedly poll the server to discover whether the blocked lock is available.

In more detail, when a lock is released, the kernel traverses the lock's fl_block list and wakes each blocked requester, resulting in a 'scrum' to get the lock. The winner then places all losers on its fl_block list.

This queue is fair to contending processes in that that blockers wake in order, and it is likely that a process awakened late will find the lock already claimed. But it's not fair to LOCKD, which has to perform some bookkeeping tasks before requesting the lock, which gives local processes an unfair advantage. And it is especially unfair to the NFSv4 server, which must wait for a contending client question to poll again before it can attempt to acquire the lock.

The new 'fair' fl_block semantics

We tried modifying the VFS lock code so that it grants locks to queued contenders, wakes the lucky ones whose locks succeed, and returns the others to the fl_block list. We used a kernel lock to protect the fl_block list during processing. We immediately ran into a few problems:

- Claiming the lock means calling posix_lock_file which calls kmalloc which can sleep, not possible when under a spinlock; so we'd have to use a semaphore or mutex; but
- For the purposes of mandatory lock checking, this new lock must be obtained in the read/write path to check for lock compliance, and adding a semaphore or mutex to the performance-critical read/write path is thought to be inefficient.

We investigated alternative locking schemes, however we soon identified a critical problem: an NFSv4 client that has been polling for a lock may stop polling at any time without notice. (For example, a user might grow weary of waiting for an application polling for a lock to make progress and issue an interrupt.)

Granting a lock to a client that does not want it is benign if the lock grant can be revoked. However, in some cases it may be difficult for the server to revoke the errantly granted lock, e.g., if the lock has been downgraded or coalesced with other locks. In these cases, the incorrect behavior cannot be reversed with a simple unlock.

This suggests the ability to request a new kind of byte-range lock from the back end file system—a provisional lock—that supersedes contending lock requests, but that does not downgrade or coalesce existing posix locks. This lets us remove the lock safely and easily if the client does not return.

Our patches add this provisional lock type to the VFS lock code. After these patches, the VFS lock code again walks through the fl_block list, now applying provisional locks as it can, and waking these queued contenders. We do not upgrade the lock to a real posix byte-range lock until the contender wakes up and requests (or, optionally, cancels) the lock.

To address the concern that the contender may never return, we consider the three cases: process, NLM client, and NFSv4 client.

First, the structure of the Linux kernel guarantees that a contending process on the queue must return: a process can lose interest in a lock only through an external signal, and the kernel signal handling code removes the process from the lock queue. Similarly, an NLM client that loses interest in a lock cancels its request when it wakes up, giving LOCKD the opportunity to revoke the lock request. Finally, if an NFSv4 client loses interest in a lock, NFSD revokes the lock request after a timeout.

The provisional lock is simple enough to be applied without requiring memory allocations, which sidesteps the kernel spinlock problems described earlier.

Along the way, we identified and fixed some problems with the NFSv4 protocol:

- The NFSv4 protocol has no equivalent to the NLM “cancel” call. This means that when a client process stops blocking on a lock, the server may wait up to a lease period (typically about a minute) before giving up and allowing another waiting client to take the lock. We found a solution that is backwards compatible (and thus implementable by current NFSv4.0 clients and servers), and also added language describing this solution to the new NFSv4.1 draft
- The NFSv4 protocol has no equivalent to the “grant” call; clients must thus poll very frequently if they wish to acquire contended locks in a timely manner. However, the traditional NLM grant call is known to have problems, e.g., numerous race conditions. We therefore proposed an alternate mechanism that allows a server to notify a client of lock availability without committing the server to granting the lock to that client. Specific language for NFSv4.1 has been proposed and met with interest, but is awaiting working group consensus.

Cluster File System lock() Interface
A Linux file system is allowed to export its own lock() method, but only a few file systems bother to do so. In particular, none of the file systems exported with NFS export a private lock() method. Consequently, neither LOCKD nor NFSD attempt to use private lock() methods.

Cluster file systems, on the other hand, do want to export a lock() method that is called by LOCKD and NFSD so that the back end can maintain a consistent view across servers. However, the current private lock() interface is unsuitable for cluster file systems.

- As before, we can't afford to block the NFSv4 server or LOCKD threads, which argues for an asynchronous interface. This is especially helpful for non-blocking locks, which do not offer the option of returning a temporary “blocked” response followed by a callback that grants the request.
- From the earlier discussion, even if the request is for a blocking lock, the file system must anticipate a return from the lock() method without having fully acquired the lock. We also need to anticipate cases where a process on the client is interrupted and the client cancels the lock.

Tasks
- Design and implement an asynchronous interface to the private lock() method
- Have LOCKD and NFSD test for the presence of a private lock() method and invoke the method when it is present

Progress
Acquiring a cluster file system lock may require communication with remote hosts. To avoid blocking lock manager threads during such communication, we allow the results to be returned asynchronously.

If a file system specific lock() invocation decides that it must block, e.g., because of a delay incurred in the course of granting a non-blocking lock request, the file system returns -EINPROGRESS. Later, the file system returns the result of the lock request through a callback registered in the lock_manager_operations struct.

An FL_CANCEL flag is added to the struct file_lock to indicate to the file system that the caller wants to cancel the provided lock.

New routines vfs_lock_file, vfs_test_lock, and vfs_cancel_lock replace posix_lock_file, posix_test_file, and posix_cancel_lock in LOCKD and the NFSv4 server. They invoke the private lock() method if it exists, otherwise they invoke the posix lock() method.

Status
Our solution has been tested with the GPFS file system. Patches have been submitted to the Linux community, and we are responding to comments.

The lack of a consumer in the Linux kernel, e.g., a cluster file system with byte-range locking, has impeded acceptance, but with GFS2 now included in the Linux 2.6.19-rc1 kernel, we have reason for optimism.

Appendix II: Cluster Coherent NFSv4 and Share Reservations

Background
NFSv4 share reservations control the concurrent sharing of files at the time they are opened. Share reservations come in two flavors, ACCESS and DENY. There are three types of ACCESS reservations: READ, WRITE, and BOTH; and four types of DENY reservations: NONE, READ, WRITE, and BOTH.

ACCESS reservations are familiar to Linux users, as they map directly to posix open() flags. NFSv4 ACCESS shares of READ, WRITE, and BOTH map directly to O_RDONLY, O_WRONLY and O_RDWR, respectively.

NFSv4 DENY reservations act as a type of whole file lock applied when a file is opened. NFSv4 DENY shares of READ, WRITE, and BOTH prevent other opens with read, write, or any access from succeeding. DENY NONE allows other opens to proceed.

The Linux system call interface for open() follows the posix standard, which does not include support for share reservations. In particular, there is no direct analog in posix for an application to request DENY READ, WRITE, or BOTH shares. Consequently, Linux NFSv4 clients always use DENY NONE.

The mismatch between posix and NFSv4 shares is also reflected on an NFSv4 server. The Linux NFSv4 server that receives DENY reservations from clients that can express them, which in practice means Windows clients, does the appropriate bookkeeping and enforcement, but the local file system is unable to enforce DENY shares for local access on the server.

When a cluster file system is exported with NFSv4, multiple NFSv4 servers export a common back-end file system, so ACCESS and DENY reservations must be distributed to take into account shares from other NFSv4 servers. In other words, the NFSv4 server has to ask the cluster file system if an incoming OPEN share can be granted.

DENY Share Support in Linux
Adding DENY share support to the Linux kernel faces several obstacles:

- DENY shares are alien to posix, the Linux model for file systems.
- There are currently no open Linux file systems that support DENY shares.
- Linux and all other UNIX-like NFSv4 clients currently work correctly because they never request DENY access.
- DENY shares do not meet the NFSv4 access needs of Linux clients, just Windows clients.
- Not even off-the-shelf Windows clients benefit as NFSv4 for Windows is a third-party add-on (from Hummingbird).
- The user level SAMBA server implements DENY shares with open and flock (albeit with the obvious race conditions), which obviates kernel support.

Implementation Issues
To enforce open share DENY access across the cluster back end is complicated, since an open with DENY must atomically lookup, (possibly) create, open, and lock the target file.

The Linux client atomically joins lookup, create, and open with lookup intents; the back end may have to do the same thing. The Linux client must also make the open and lock an atomic operation, but there is a problem: you can't lock that doesn't exist, so you must first create it. But as soon as the file is created, some other application might find it and lock it. Returning an error to an open that succeeding in creating a file is unexpected behavior.

Applying restrictive mode bits to the create won't always work, either, because another application might relax the mode restrictions and open the file.

This suggests that we add the share lock to the open call instead of making it a separate operation.

One approach: new flags for open()

- Use existing O_RDONLY, O_WRONLY and O_RDWR open flags to implement O_ACCESS_READ, O_ACCESS_WRITE, and O_ACCESS_BOTH, respectively.
- Add two open flags: O_DENY_READ and O_DENY_WRITE.
- Propagate O_DENY flags to the intent structure.
- Add operation adjust_share(file, flags). The file system should be allowed to refuse operations that could not result from open or close. (So, anything that doesn't only turn bits on or only turn them off.)

Is there a race here? E.g., say we open+create with a share lock. How do we decide whether to treat it as an upgrade or an open?

Another approach: best attempt

- Issue a lookup. If the file exists, then upgrade.
- Otherwise open with implicit create. If we get an error indicating a share conflict, retry the lookup.

This is obviously not ideal.

- Would it help to get a reference on the dentry before trying the open?
- Is there currently a lookup/open race if the backend is a distributed file system? One way of looking at it is “that's up to them.” The client just needs to look at how we implement open and make sure it does the intent stuff right.

An alternative might be to expose something along the lines of the open owner to the VFS and let it decide (by comparing open owners) whether a given open is an upgrade or a new open.

Status

Implementation awaits resolution of these issues.

Appendix III: Cluster Coherent NFSv4 and Delegations

Background
NFSv4 adds a new protocol feature: delegations. RFC 3530 explains:

> The major addition to NFS version 4 in the area of caching is the ability of the server to delegate certain responsibilities to the client. When the server grants a delegation for a file to a client, the client is guaranteed certain semantics with respect to the sharing of that file with other clients. At OPEN, the server may either provide the client a read or write delegation for the file. If the client is granted a read delegation, it is assured that no other client has the ability to write to the file for the duration of the delegation. If the client is granted a write delegation, the client is assured that no other client has read or write access to the file.
>
> Delegations can be recalled by the server. If another client requests access to the file in such a way that the access conflicts with the granted delegation, the server is able to notify the initial client and recall the delegation. This requires that a callback path exist between the server and client. If this callback path does not exist, then delegations cannot be granted. The essence of a delegation is that it allows the client to locally service operations such as OPEN, CLOSE, LOCK, LOCKU, READ, WRITE without immediate interaction with the server.

Linux NFSv4 Delegation Support for Cluster File Systems
To coordinate NFSv4 delegations with local access, we implement delegations with the lease extension to the VFS lock subsystem. The VFS lock subsystem uses fcntl() to set and get a lease. To allow a lease to be recalled, e.g., because of a conflicting open, the VFS layer has a break_lease() function

When the NFS server's open() method is invoked, it may issue or recall a delegation. A delegation can be issued if it does not conflict with an existing delegation. Issuing a delegation is optional. A delegation can be recalled at any time. Recalling a delegation is mandatory if a conflicting open is received.

A conflicting open can come from a variety of sources: local access, NFS access, Samba access, etc. Every invocation of the VFS open method must check for conflict with an existing delegation and recall it if necessary. NFSD may wait for the delegation recall to complete, or may respond to

If an OPEN request forces a delegation recall, NFSD issues a CB_RECALL request to all clients holding the conflicting delegation. This is implemented on the client with the VFS layer break_lease() call, which notifies lease holders that a conflicting OPEN has occurred. The VFS layer makes this determination without consulting the underlying file system.

Once the recall of conflicting delegations is complete, NFSD can proceed with its pending OPEN request. In order to determine whether it can issue a delegation for the request, NFSD needs information that lives on the other side of the VFS layer. The VFS lease subsystem can make the determination by examining the entry for the file in the open inode table: if there are no writers, then a READ delegation can be issued; if there are no readers or writers, then a WRITE delegation can be issued. NFSD must obtain the result of this determination from the VFS layer.

If NFSD elects to grant a delegation, it must inform the underlying file system.

Tasks
- VFS OPEN must ask the file system to check for delegation recall in progress prior to granting an OPEN, granting a delegation, or initiating a recall.
- If delegation is issued, the NFS client must set up a callback path for a potential CB_RECALL request from the server.
- NFSD must ask the file system if a delegation can be granted.
- The VFS must tell the file system of a lease conflict (rename, unlink, etc) and compel it to recall any delegations.

Proposed Implementation
Extend the set/get/breaklease interfaces to service cluster file systems. The extensions will resemble the posix locking extensions (callbacks, etc).

What we probably need is new inode operations:

- break_lease(inode, mode)
- setlease(filp, mode)
- getlease(filp, &mode)

Where mode can be one of read, write, or unlock. We'd also allow the mode to be or'ed with a nonblocking flag?

The VFS lease subsystem includes a series of lock manager callbacks. Will these be sufficient for the cluster file system case?

Actually current setlease and getlease functions use a struct file_lock instead of (or in addition to) the mode. Do we need that?

Also, setlease and getlease could be file operations instead of inode operations. This is probably a fairly arbitrary choice.

To handle the possibility that break_lease, setlease, getlease, etc. might block, even in the absence of contention, we might want to allow an -EINPROGRESS return to be followed by a callback e.g. break_lease_result(inode, stat); where stat might be -EAGAIN (we're waiting for the lease to be broken) or OK (it was immediately broken, or there never was one).

Status
Implementation awaits progress in resolving the above issues.

Appendix IV: pNFS development

Development
We updated the Linux pNFS client and server to the 2.6.17 kernel level, and are preparing to rebase again for 2.6.19.

We updated the pNFS code base to draft-ietf-nfsv4-minorversion1-05. Testing identified multiple bugs, which we fixed.

The linux client separates common NFS code from NFSv2/3/4 code by using version specific operations. We rewrote the Linux pNFS client to use its own set of version specific operations. This provides a controlled interface to the pNFS code, and eases updating the code to new kernel versions.

- Four client layout modules are in development.
- File layout driver (CITI, Network Appliance, and IBM Almaden).
- PVFS2 layout driver (CITI).
- Object layout driver (Panasas).
- Block layout driver (CITI).

To accommodate the requirements of the multiple layout drivers, we expanded the policy interface between the layout driver and generic pNFS client. This interface allows layout drivers to set the following policies:

- stripe size
- writeback cache gathering policies
- blocksize
- read and write threshold
- timing of layoutget invocation
- choice of pagecache or direct method for I/O

We are designing and coding a pNFS client layout cache to replace the current implementation, which supports only a single layout per inode.

We improved the interface to the underlying file system on the Linux pNFS server. The new interface is being used by the Panasas object layout server, the IBM GPFS server, and the PVFS2 server.

We are coding the pNFS layout management service and file system interfaces on the Linux pNFS server to do a better job of bookkeeping so that we can extend the layout recall implementation, which is limited to a single layout.

We have continued to develop the PVFS2 layout driver and PVFS2 support in the pNFS server. The layout driver I/O interface supports direct access, page cache access with NFSv4 readahead and writeback, and the O_DIRECT access method. In addition, PVFS2 now supports the pNFS file-based layout, which lets pNFS clients choose how they access the file system.

We developed prototype implementations of pNFS operations:

- OP_GETDEVICELIST,
- OP_GETDEVICEINFO,
- OP_LAYOUTGET,
- OP_LAYOUTCOMMIT,
- OP_LAYOUTRETURN and
- OP_CB_LAYOUTRECALL

We continue to test the ability of our prototype to send direct I/O data to data servers.

- “Large Files, Small Writes, and pNFS” Dean Hildebrand, Lee Ward, and Peter Honeyman. In *Proc. of the 20th ACM International Conf. on Supercomputing*, Cairns (July 2006).

  We demonstrated how pNFS can improve the overall write performance of parallel file systems by using direct, parallel I/O for large write requests and the NFSv4 storage protocol

for small write requests. To switch between them, we added a write threshold to the layout driver. Write requests smaller than the threshold follow the slower NFSv4 data path. Write requests larger than the threshold follow the faster layout driver data path.

- "Direct-pNFS: Simple, Transparent, and Versatile Access to Parallel File Systems" by Dean Hildebrand and Peter Honeyman. A preliminary version is available as *CITI Technical Report 06-8* (October 2006).

  We improved the performance and scalability of pNFS file-based access with parallel file systems. Our design, named Direct-pNFS, augmented the file-based architecture to enable file-based pNFS clients to bypass intermediate data servers and access heterogeneous data stores directly. Direct access is possible by ensuring file-based layouts match the data layout in the underlying file system and giving pNFS clients the tools to effectively interpret and utilize this information. Experiments with Direct-pNFS demonstrate I/O throughput that equals or outperforms the exported parallel file system across a range of workloads.

Milestones
At the September 2006 NFSv4 Bake-a-thon, hosted by CITI, we continued to test the ability of CITI's Linux pNFS client to operate with multiple layouts, and the ability of CITI's Linux pNFS server to export pNFS capable underlying file systems.

We demonstrated the Linux pNFS client support for multiple layouts by copying files between multiple pNFS back ends.

The following pNFS implementations were tested.

File Layout
- Clients: Linux, Solaris
- Servers: Network Appliance, Linux IBM GPFS, DESY dCache, Solaris, PVFS2

Object layout
- Client: Linux
- Servers: Linux, Panasas

Block layout
- Client: Linux
- Server: EMC

PVFS2 layout
- Client: Linux
- Server: Linux

Activities
Our current Linux pNFS implementation uses a single whole file layout. We are extending the layout cache on the client and layout management on the server to support multiple layouts and small byte ranges.

In cooperation with EMC, we continue to develop a block layout driver module for the generic pNFS client.

We continue to measure I/O performance.

We joined the Ultralight project and are testing pNFS I/O using pNFS clients on 10 GbE against pNFS clusters on 1 GbE. The Linux pNFS client is included in the Ultralight kernel and distributed to Ultralight sites, providing opportunities for future long haul WAN testing.

We are demonstrating pNFS file layout at SC06. The demonstration will include multiple 10G nic clients from the Sc06 demonstration floor accessing data across the Ultralight network to an IBM GPFS cluster at CITI.

Appendix V: NFS Recovery and Client Migration

Background

By exporting a shared cluster file system using multiple NFS servers, we can provide increased performance and availability through load balancing and failover. NFSv4 provides some minimal protocol features to allow migration and failover, but there are some implementation challenges, and some small protocol extensions required.

Consider a few scenarios, in order of increasing complexity:

1. Server 1 and server 2 share the same cluster file system. Server 1 runs an NFS server, but server 2 doesn't. When server 1 fails or is shut down, an NFS service is started on server 2, which also takes over server 1's IP address.
2. Server 1, 2, and 3 share the same cluster file system. Everything is as in the previous scenario, except that server 3 is also running a live NFS server throughout; so we need to ensure that server 3 is not allowed to acquire any locks it shouldn't during the period that server 2 is taking over.
3. Server 1, 2, and 3 share the same cluster file system. Everything is as in the previous scenario, except that server 2 is already running a live nfs server. We must handle the failover with minimal interruption to the preexisting clients of server number 2.
4. As in the previous scenario, except that we expect to keep server 1 running throughout, and migrate possibly only some of its clients. This allows us to do load balancing.

The implementation may choose to block essentially all locking activity during the transition, possibly as long as a grace period, which is on the order of a minute. This may be simpler to implement and may be adequate for some applications. However, we prefer to implement the transition period in such a way that applications see no significant delay. A variety of behaviors in between (e.g. that limit delays only to certain files) are also possible.

Finally, the implementation may allow the client to continue to use all its existing state on the new server, or may require the client to go through the same state recovery process it would go through on server reboot. The latter approach requires less intrusive modifications to the nfs server, and can be done without requiring ceasing locking activity, but there are still optimizations possible using the former method that may reduce latency to the migrating client.

We are exploring most of these possibilities as part of our Linux NFSv4 server implementation effort.

Protocol issues

In the process of designing the migration implementation for Linux, we have identified two small deficiencies in the NFSv4 protocol that limit the migration scenarios that an NFSv4 implementation can reliably support.

Migration to a live server

Scenarios 3 and 4 above involve migrating clients to an NFSv4 server that is already serving clients of its own. This causes some problems, which we need a little background to explain.

To manage client state, we first need a reliable way to identify individual clients; ideally, it should allow us to identify clients even across client reboots. Thanks to NAT, DHCP, and user space NFS clients, the IP address is not a reliable way to identify clients. Therefore the NFSv4 protocol uses a client-generated “client identifier” for this purpose.

However, to avoid some potential problems caused by servers that have multiple IP addresses, the NFSv4 spec requires the client to calculate the client identifier in such a way that it is always different for different server IP addresses.

This creates some confusion during migration—the client identifier that the client will present to the new server will differ from the one it presented to the old server, so we do not have a way to track the client across the migration.

The problem can be avoided in scenarios 1 and 2 by allowing the new server to take over the old server IP address.

The problem has been discussed in the NFSv4 IETF working group, but a solution has not yet been agreed on. Nevertheless, we expect the problem to be solved in NFSv4.1.

Transparent state migration
We have identified a small protocol change necessary to support transparent migration of state: migration that doesn't require the client to perform lock reclaims as it would on server reboot.

The problem is that a client, when migrating, does not know whether the server to which it is migrating will wants it to continue to use the state it acquired on the previous server, or wants it to reacquire its state. The client could attempt to find out by trying to use its current state with the new server and seeing what kind of error it gets back. However, there's no guarantee this will work—accidental collisions in the StateIDs handed out by the two servers may mean, for example, that the server cannot return a helpful error in this case.

The current NFSv4.1 draft partially solves this problem by defining a new error (NFS4ERR_MOVED_DATA_AND_STATE) that the server can return to simultaneously trigger a client migration and to indicate to the client that the new server is prepared to accept state handed out by the old server. (This solution is only partial because it doesn't help with failover—in that case, it's too late for the old server to return any errors to the client.) The final 4.1 specification will probably contain a more comprehensive solution, so at this point we're confident that the problem will be solved.

Linux implementation issues

Scenario 1 and NFSv4 reboot recovery
The current linux implementation currently supports the above scenario 1 with NFSv2/v3. (See, for example, Falko Timme's Setting up a high-availability NFS server, which actually shares data at the block level instead of a cluster file system.) It is possible to support the same scenario in NFSv4 as long as the directory where the NFSv4 server stores its reboot recovery information (/var/lib/nfs/v4recovery/ by default), is located on shared storage. However, there is a regression compared to v2/v3, because the v2/v3 also provides synchronous callouts to arbitrary scripts whenever that information changes, so that it could be shared using methods other than shared storage. The NFSv4 reboot recovery information is currently under redesign, and one of the side effects of the new design will be to allow such callouts. This work is not yet completed.

Scenario 2 and grace period control
The simplest way to support scenario 2 is to require clients of node 1 to recover their state on node 2 using the current server reboot recovery mechanism, by forcing both server 2 and 3 to observe a grace period.

We have patches to implement this approach available from the “server-state-recovery” branch of our public git repository; see a browsable version of the repository.

This approach is unsatisfactory because it forces the NFS server to observe a grace period for all exported file systems, even those that aren't affected (or even shared across nodes), and because it blocks all locking activity across the cluster for the duration of the grace period.

Therefore, we have a second design that allows us to limit the impact: instead of simply forcing servers 2 and 3 into grace, we remove all grace period checking from the nfs server itself, instead allowing the underlying file system to enforce locking operations when it is called to perform the locks. (This new behavior is enabled only for file systems such as cluster file system that define lock methods; behavior for other file systems is unchanged.) In order to enable the file system to make correct grace period decisions, we also need to distinguish between “normal” lock operations and reclaims; we accomplish by flagging reclaims locks when they are passed to the file system.

The cluster file system may then choose how to handle the migration; it may choose to continue to enforce a grace period globally across the whole file system, but it is now given the information

to enable it to make more sophisticated decisions if it prefers.

The patches in our git repository, referenced above, include incomplete support for this new design.

Due to the minimal support for cluster file systems in the mainstream Linux kernel, these patches (like the byte-range locking patches) are unlikely to be accepted for the time being.

While implementing this we also noticed a problem with our current NFSv4 implementation, which is that its grace period is not necessarily synchronized with the grace period used by lockd, which can cause problems in a mixed NFSv2/v3/v4 environment; those patches are available from our git repository and we expect them to proceed upstream normally.

Scenario 3 and 4, and reboot recovery interfaces
Like statd, the NFSv4 server is required to maintain information in stable storage in order to keep track of which clients have successfully established state with it. This solves certain problems identified in RFC 3530, where combinations of server reboots and network partitions can lead to situations where neither client nor server could otherwise determine whether the client should still be allowed to reclaim state after a reboot.

We are in the process of redesigning the linux implementation of this system, partly for reasons given under “Scenario 1 and NFSv4 reboot recovery” above.

As part of this new design, we need a way for a user space program to tell the NFS on startup which clients have valid state with the server (after that program has retried this information from stable storage).

Scenarios 3 and 4 require a kernel interface allowing an administrator to migrate particular clients from and to NFS servers. To this end, we plan to use the same reboot recovery interface.

The interface will consist of a call that takes a client identifier and a status (as an integer).

For normal NFSD startup, one call will be made for each known client, with a status of 0.

Similarly, to inform a server that a new client is migrating to it (and hence that it should allow lock reclaims from that client), we will again make one call for that client with status 0.

To inform a client that it should initiate a migration event, by returning NFS4ERR_MOVED to the client, we'll make a call for that client with status NFS4ERR_MOVED.

This interface may also be extended in the future to allow for, for example, administrative revocation of locks.