The Global File System

Ken Preslan

kpreslan@sistina.com

Sistina Software

http://www.sistina.com/

Outline

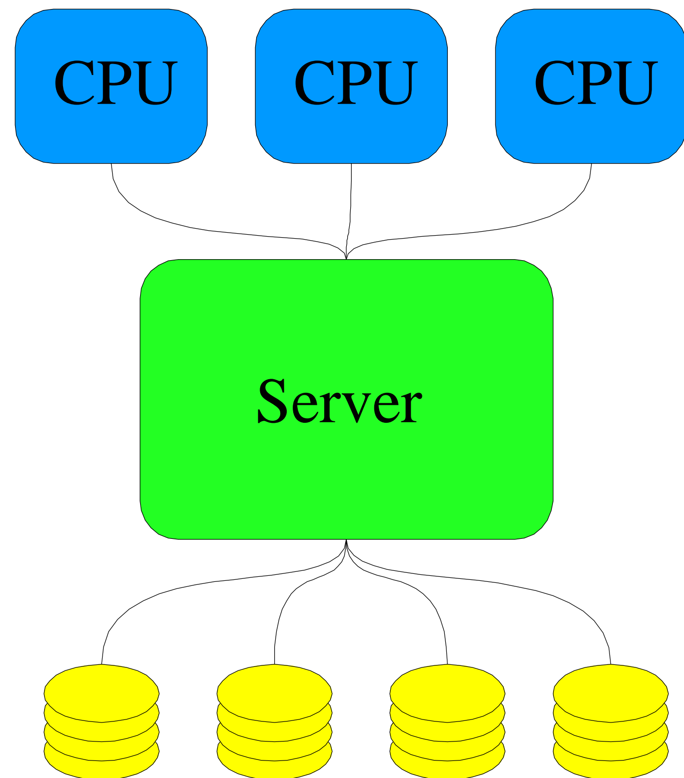
- Sistina
- SAN Background
- GFS Features
- Locking
- Journaling and Distributed Recovery
- Future Work

Sistina Software

- The GFS Group from University of Minnesota left to become Sistina Software
- Offices all over the US and Europe
- 20+ engineers working on GFS and related software
- 5+ years working on GFS and clustered storage

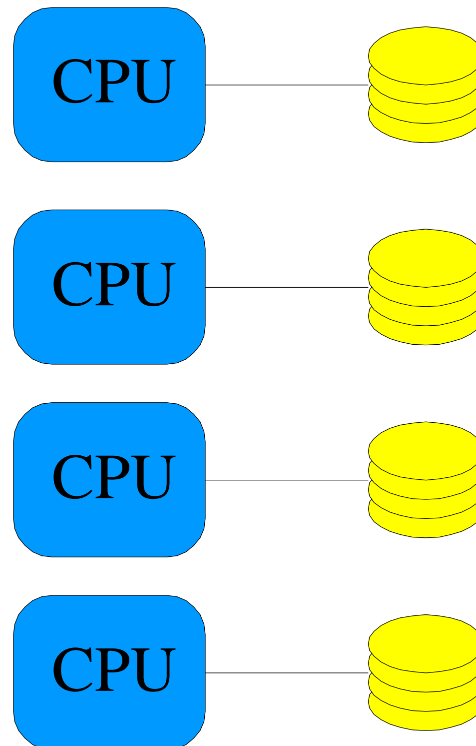
Centalized File Server

- Traditional network storage model
- NFS, CIFS
- Server is a bottleneck
- Server is a single point of failure



Replicated–Data Server Farm

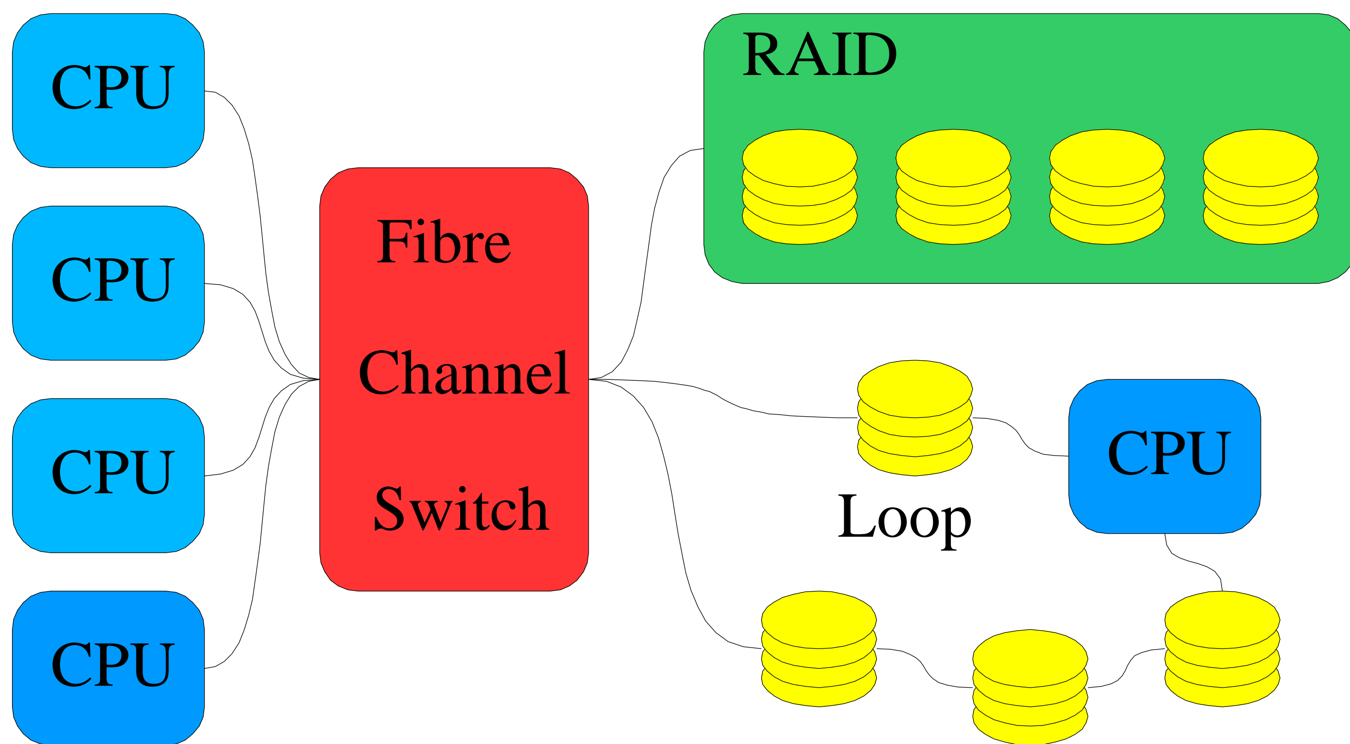
- Each machine has its own disks
- Data is kept coherent on each machine manually
- System configuration must be done on each machine



Storage Area Networks (SANs)

- New Storage Area Networks like Fibre Channel (FC) allow a different approach
- Disks and hosts connect to the storage network
- All the machines can talk to every disk
- Each machine accesses the shared disks as if they were local
  - Faster access
  - Greater availability

A Fiber Channel Network

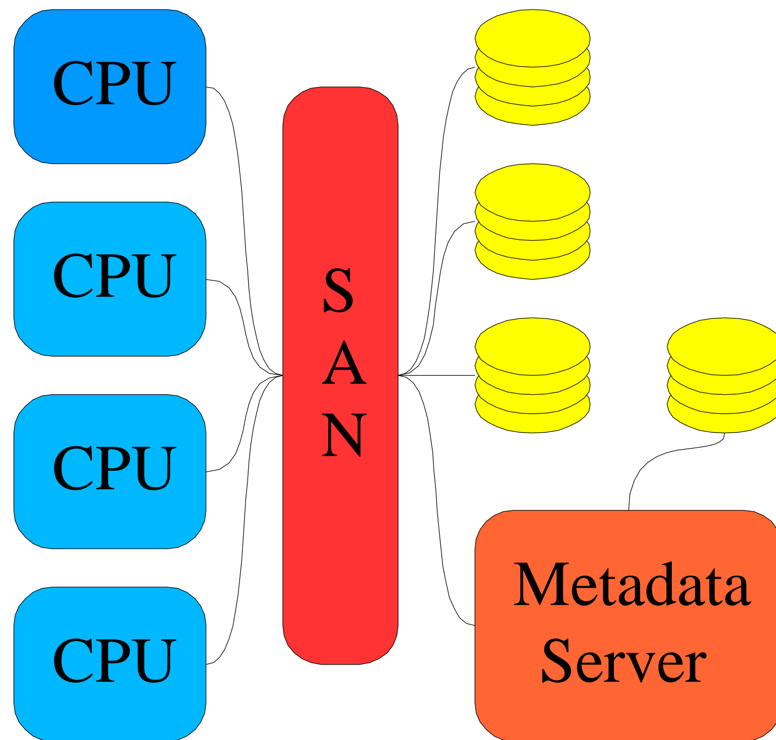


Shared Disk File Systems (SDFS)

- Sharing disks over a SAN requires a filesystem that understands shared storage
- The FS coordinates the accesses and disk caching of the machines
- Need a method of synchronization between the machines
- Two catagories of SDFS
  1) 3rd Party Transfer (Asymmetric)
  2) DMEP/GFS (Symmetric)

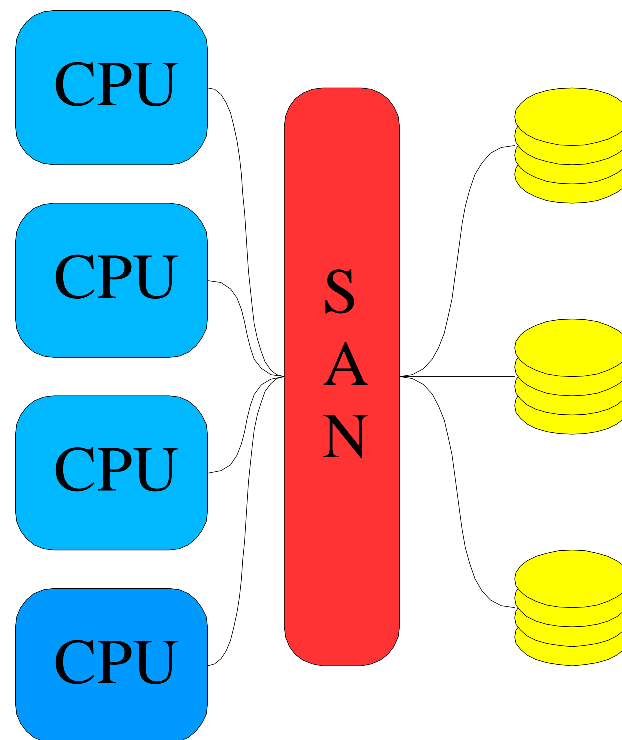
Asymmetric SDFS

- Machines share disks containing data, not metadata
- Metadata is controlled by a central server
- The server provides synchronization between clients
- Machines make metadata requests (create, unlink, bmap) to the server
- Machines read actual data from the disks
- Similar to a traditional DFS
- SGI’s CXFS, CentraVision, EMC’s MPFS



Symmetric SDFS

- Machines share disks containing data and metadata
- Metadata is managed by each machine as it is accessed
- Synchronization is achieved using global locks (DMEP or a Distributed Lock Manager (DLM))
- A local file system with inter–machine locking
- GFS, VaxCluster, Frangipani



The Global File System

- Symmetric Shared Disk File System
- Open Source (GNU GPL)
- 64–bit Files and File System
- Journaled
- High Performance
- Originally for Irix, now Linux, and FreeBSD next

Possible Applications/Uses

- Web Serving clusters
- NFS Serving clusters
- Shared–Root file systems for ease of administration
- I/O intensive Beowulf clusters

Many Parts to the Project

- Network Storage Pool manager (Pool)
- Logical Volume Manager (LVM)
- DMEP SCSI command
- Global Network Block Driver (GNBD)
- Lock Modules (nolock, MemExp)
- Failover IP MemExp server (MemExpd)
- STOMITH framework
- Global File System (GFS)

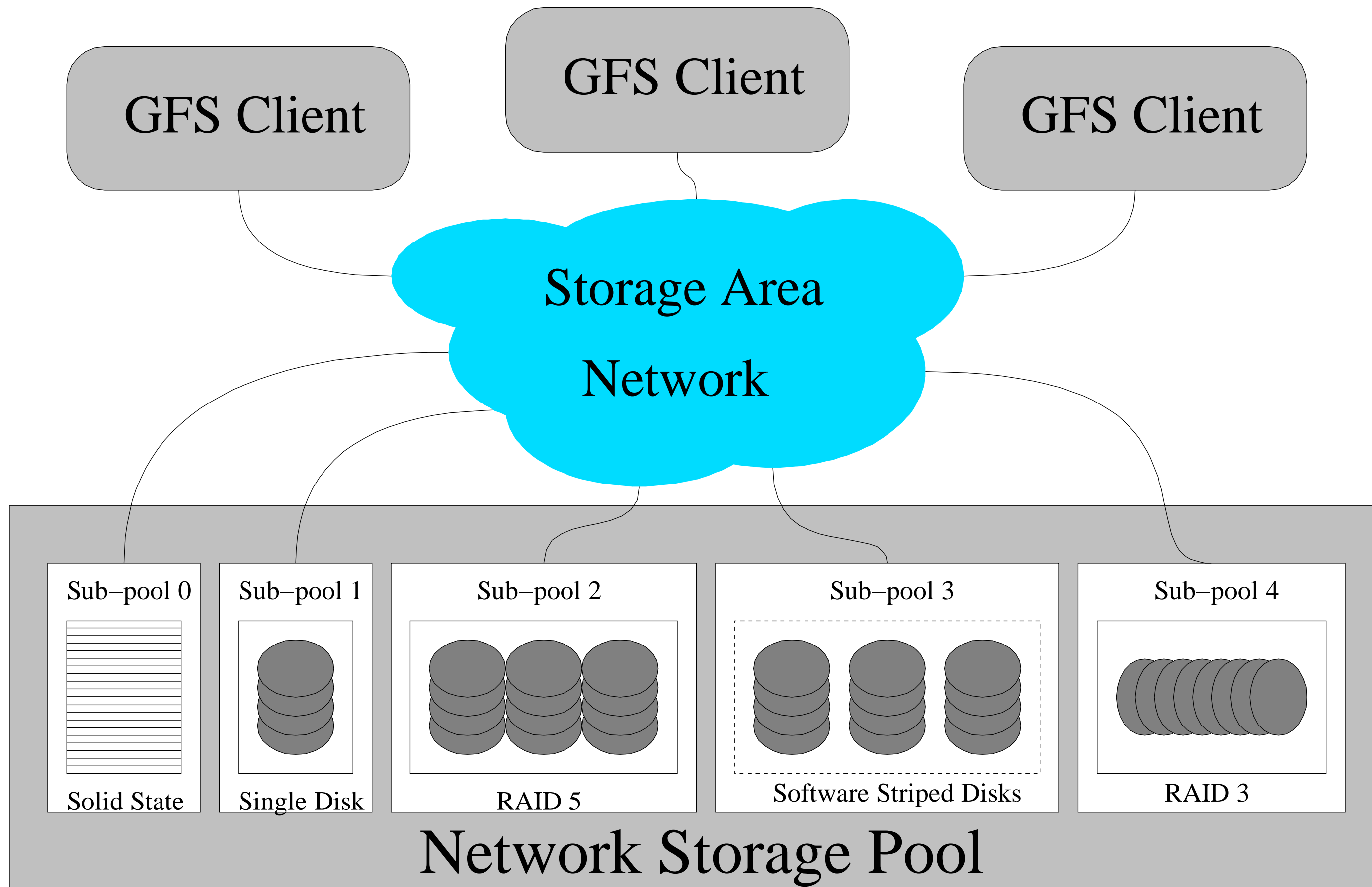
Memory Export Command

- SCSI command that exports memory buffers to the SAN
- 9–byte sparse namespace
- Buffer size is set–able
- Modified by a Load Locked, Store Conditional method
- Used to implement the old Dlock spec; the state machine is maintained by the clients

The Pool Driver

- A Logical Volume Driver for SANs
  - Combines multiple disks into one logical address space
  - Combines multiple DMEP devices into one logical lock space
- Handles disks that change IDs because of network rearrangement
- A Pool is made up of SubPools of devices with similar characteristics

A Network Storage Pool



The File System

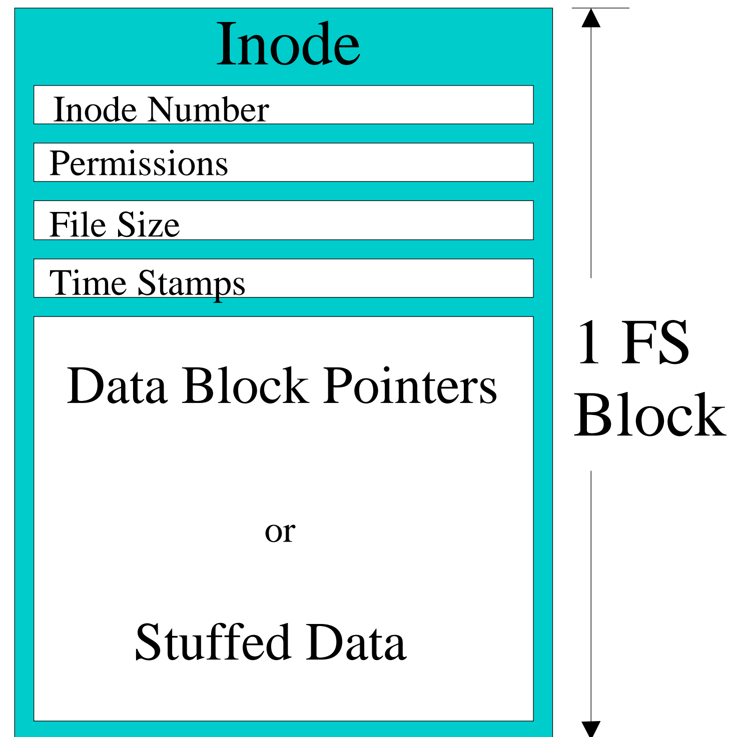
- A high performance local file system with inter–machine locking
- Optimized for Network Attached Storage
- When the locks are removed, GFS makes a decent local file system

GFS Features

- Dynamic inodes
- Flat/64–bit metadata structure
- Platform independent metadata
- Extendible Hashing Directories
- Full use of the buffer cache
  (full read and write–back caching)
- Cross Platform Support
- Online Growable

Dynamic Inodes

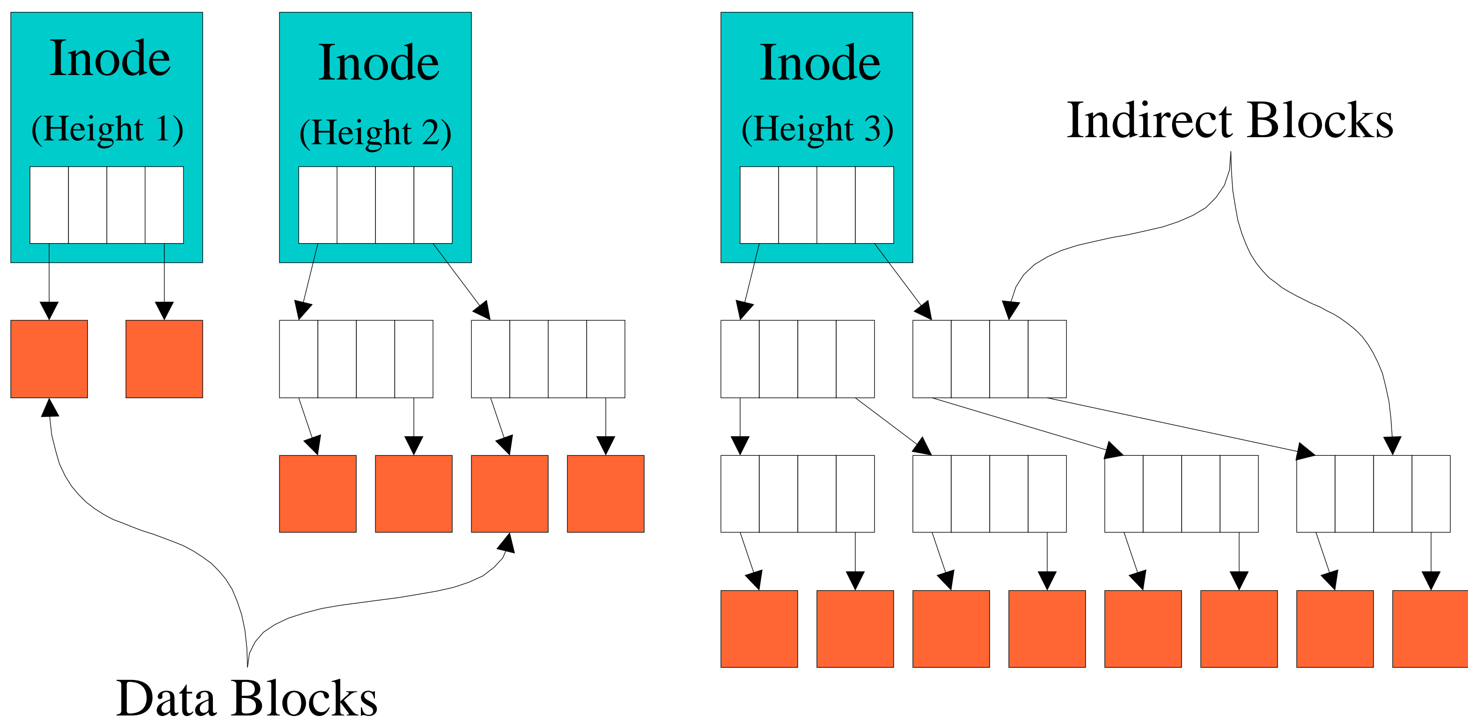
- No preallocated inode tables
- Each inode is just a file system block
- There can be as many inodes as there are file system blocks
- Inode numbers are just disk addresses
- Inodes identified in the allocation bitmaps
- Inodes can be *stuffed* for space efficiency



Flat/64–bit File Structure

- All file sizes, offsets, and block addresses are 64 bit
- File metadata trees are of uniform height
- All direct pointers, or all indirect pointers, or all double indirect pointers...
- Tree height grows to accommodate the size of the file
- No practical file size limit
- Simplifies the block mapping routines

Flat/64–bit File Structure



Platform Independent Metadata

- All on–disk structures are in a platform independent format
- Differences in structure packing are handled
- Differences in endianess are handled
- Very important for GFS because all clients must understand and manipulate the metadata

Fast Directories

- Small directories are stuffed in the inode
- Larger directories use a technique called *Extendible Hashing*
- File names are hashed into keys that are indices into a growable hash table
- Faster than B–Trees
- A bit more space hungry

Using the Buffer Cache

- The buffer cache is critical to the performance of a file system
- Linux's buffer cache is written with the assumption that only one machine is modifying the data on the disks
- GFS uses routines to keep track of the buffers in the buffer cache and invalidate them when necessary
- GFS can do both read and write caching

Architecture Abstraction

- Two layers
  - Top layer – VFS to GFS interface
    - Each OS (or even major OS release) gets its own directory of code
    - One (big) directory of common code
  - Bottom Layer – GFS to bufcache/memory/etc...
    - a include file and library of abstractions that GFS uses

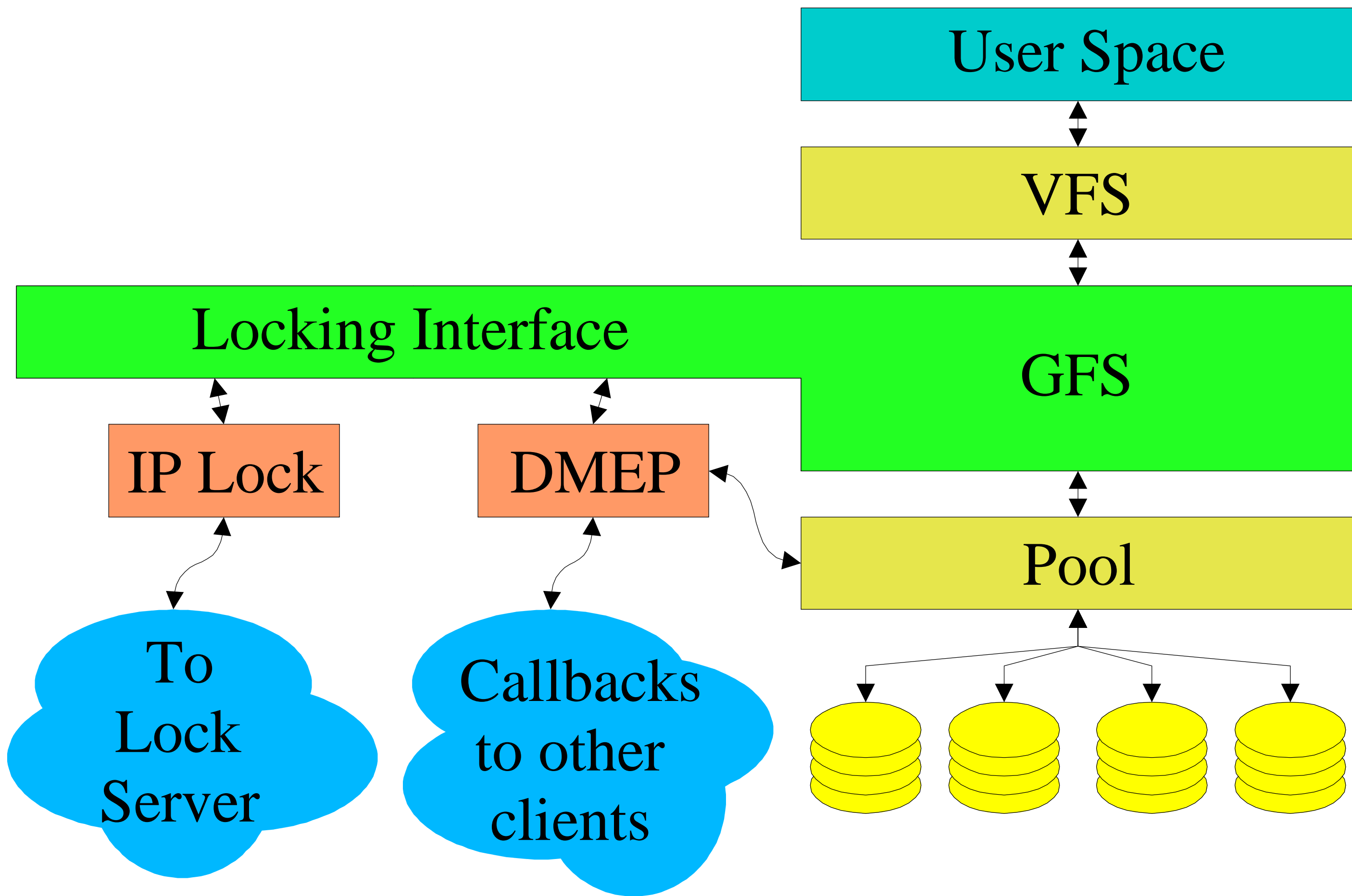
Online Growable

- Can add more filesystem space online
  - Resource Groups (data space)
  - Journals
- New space automatically detected by the rest of the cluster
- Pool supports online growing as well

Locking

- The locking code is broken into layers
  - Glock layer
    - Part of the filesystem
    - Handles lock caching
  - Locking Modules
    - Handles low–level mechanics of acquiring locks
    - Provides abstract locking interface that GFS uses
    - Interchangeable

Organizational Structure



Glock Layer Features

- Allow recursive locking
  - –Helpful when routines that acquire Glocks are reused
  - –If one process has a Glock, another process that wants the same lock doesn’t generate lock module request

```
int function1()
{
    glock(101);
    /*  Do something  */
    function2();
    gunlock(101);
}
int function2()
{
    glock(101);
    /*  Do something else  */
    gunlock(101);
}
```

Glock Layer Features

- Handle the buffer cache invalidation
- Syncs dirty data and incore transactions before lock release
- Keep the in–memory inodes up to date

gfs_get_glstruct()

- gfs_glock_t *gfs_get_glstruct(sdp, lock, type, create)
  - *sdp* – a pointer the filesystem data
  - *lock* – a 64–bit lock number
  - *operations* – lock–specific functions run on acquire or release...
  - *parent* – the lock’s parent
  - *create* – TRUE to create the glock structure
- void gfs_put_glstruct(gl)

gfs_glock()

- int gfs_glock(sdp, gl, flags)
  - *sdp* – a pointer the filesystem data
  - *gl* (gfs_glock_t *) – the lock structure
  - *flags* (int) – GL_SHARED, GL_DEFERRED, GL_NOEXP, GL_TRY
  - Returns: a negative standard errno, 0 for success, or GLR_TRYFAILED

gfs_gunlock()

- int gfs_gunlock(sdp, gl, flags)
  - *sdp* – a pointer the filesystem data
  - *gl* (gfs_glock_t *) – the lock structure
  - *flags* (int) – GL_INCR, GL_SYNC, GL_NOCACHE
  - Returns: standard errno

Simple Example

```
int64 gfs_write(gfs_inode_t *ip, char *buf, uint64 offset,
                uint64 size)
{
    gfs_direct_t *direct;


    gfs_glock_i(ip, 0);

    gfs_bmapi_write(ip, offset, size, &direct);

    /*  Write the Data  */

    gfs_gunlock_i(ip, 0);


    return(0);
}
```

Why multiple simultaneous Glocks?

```
int gfs_lookup(gfs_inode_t *dip, char *name, gfs_inode_t **ip)
{
     uint64 inode_number, glock;
     gfs_glock_t *gl;


     gfs_glock_i(dip, GL_SHARED);

     inode_number = gfs_dir_search(dip, name);

     gfs_gunlock_i(dip, 0);

     /*  It's bad if an unlink on "name" happens here  */

     gl = gfs_get_glstruct(dip->i_sbd, inode_number,
                           LM_TYPE_META, CREATE);

     gfs_glock(dip->i_sbd, gl, GL_SHARED);

     *ip = iget(dip->i_sbd, inode_number);

     gfs_gunlock(dip->i_sbd, gl, 0);


     return 0;
}
```

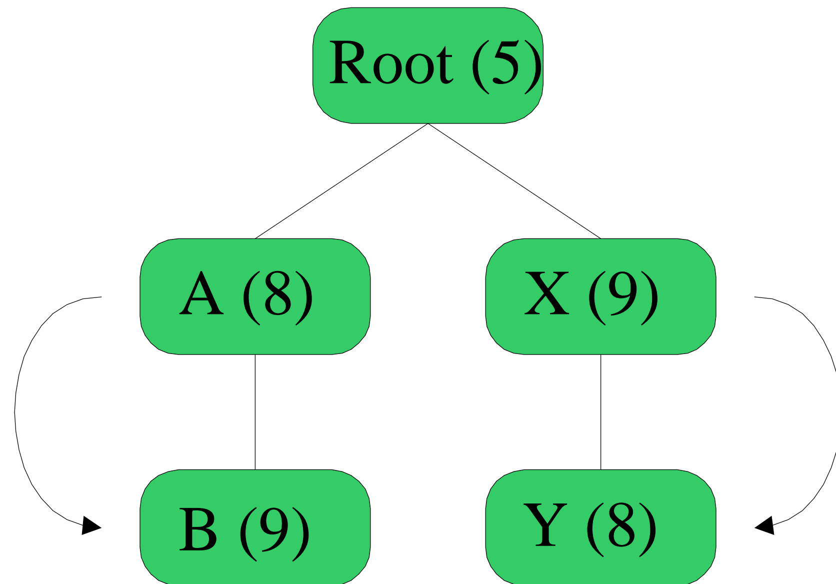
Overlapping Glocks

```
int gfs_lookup(gfs_inode_t *dip, char *name, gfs_inode_t **ip)
{
     uint64 inode_number, glock;
     gfs_glock_t *gl;


     gfs_glock_i(dip, GL_SHARED);

     inode_number = gfs_dir_search(dip, name);

     gl = gfs_get_glstruct(dip->i_sbd, glock, inode_number,
                           LM_TYPE_META, CREATE);

     gfs_glock(dip->i_sbd, gl, 0);

     *ip = iget(dip->i_sbd, inode_number);

     gfs_gunlock(dip->i_sbd, gl, 0);


     gfs_dunlock_i(dip, 0);


     return 0;
}
```

Deadlock

- Anytime you’re acquiring overlapping locks you have to deal with deadlock
- Complicated by a many–to–one inode to lock mapping
  - Can’t rely on directory hierarchy for ordering
- GFS has implemented multiple approaches over the years
  - backoff–and–retry
  - sorting lock numbers
- Now have a Sparse Lock space in DMEP
  - one–to–one (or better) inode to lock mapping

Deadlock with Aliased Lock Space



Write Caching

- GFS can’t have dirty cached data after its Glock is released (or demoted from exclusive to shared)
- If Glocks are acquired and released each operation, caching is essentially write–through
- If Glocks are held for longer periods of time, write caching can be more useful
- Glock layer manages coherency of the buffer cache

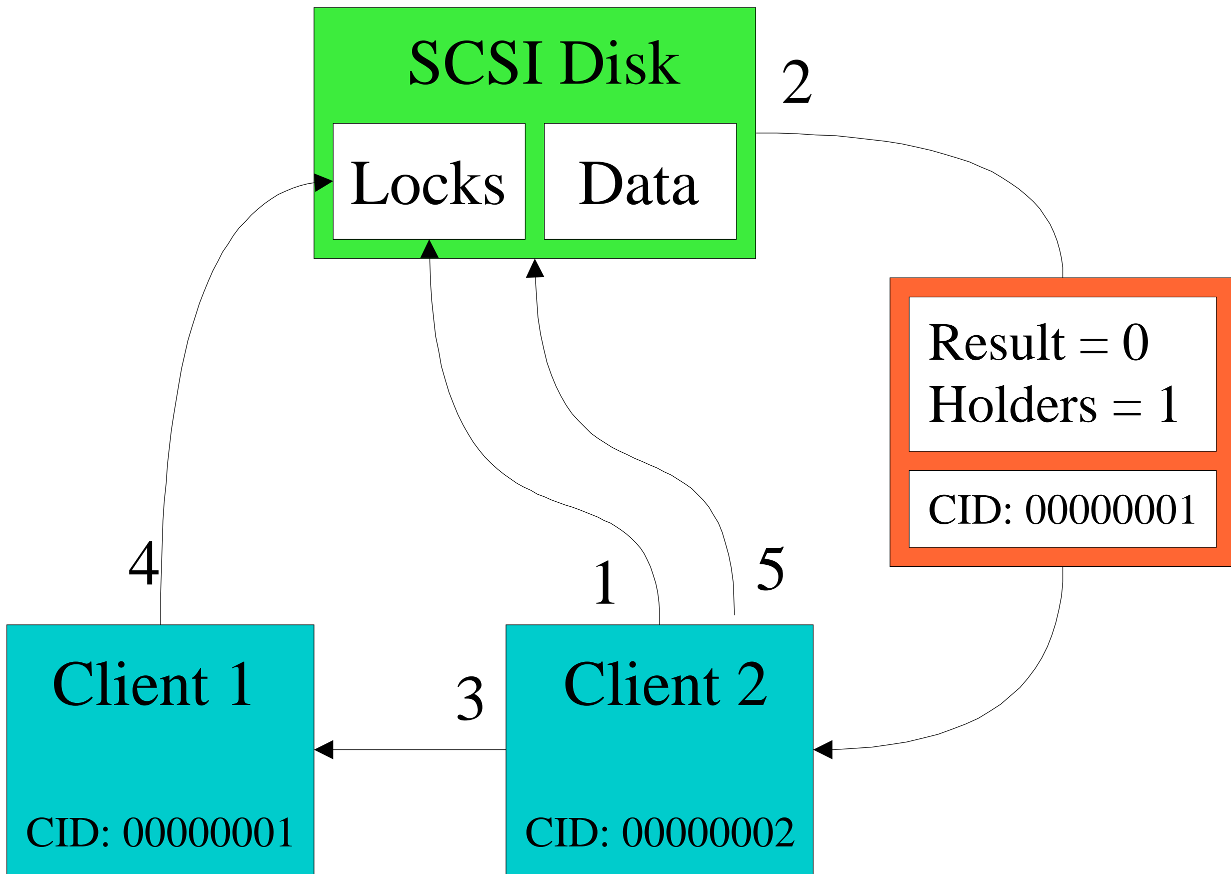
Glock States

- Not held
  - This machine doesn't hold the lock
  - There is no dirty data for this lock
- Held
  - This machine holds the lock, but no process is currently using it
  - There can be dirty data for this lock
  - The lock can be released any time another machine needs it
- Locked
  - This machine holds the lock, a process is using it
  - There can be dirty data for this lock
  - The lock can't be released until it is dropped to Held

Call Backs

- Client 1 first holds the lock
  1) Client 2 tries to acquire the lock
  2) The failure message from the Dlock device contains the Client ID of the client holding the lock (Client 1)
  3) Client 2 asks Client 1 to release the lock
  4) Client 1 syncs back its data and releases the lock
  5) Client 2 can then acquire the lock

Call Backs



Interchangeable Locking Modules

- Want GFS to be independent of the type of inter–machine locking available
- Created a locking interface to allow modules to plug into GFS
- Each module translates between the locking that GFS expects and the locking available
- Responsible for configuration of the cluster (which client mounts which journal)
- Responsible for issuing callbacks between machines

Registration

- Locking modules register themselves with GFS using the function *register_lock_proto*()
- The module registers a structure containing the a structure of operations that the module implements
- Operations: mount, unmount, get_lock, put_lock, lock, unlock, cancel, reset, hold_lvb, unhold_lvb, sync_lvb, reset_expired

Operations

- *Mount* – Called once at mount time to set up the lock space
  - Table Name – a name identifying the lock space to be used. (e.g. The Pool name)
  - Call Back – Allows the locking module to ask GFS to unlock a lock
  - Returns the Journal Number the client should mount
  - A flag to tell GFS to check all the journals
- *Unmount* – Called at unmount time to close the lock space

Operations

- *get_lock* – Get a lock structure given a lock description
  - Lock Number
  - Lock Type
  - Lock’s parent
- *put_lock* – get rid of a lock structure

Operations

- *Lock* – Acquire a lock
  - Lock Structure
  - Old lock state
  - Requested lock state
  - Flags – Try, NoExp (same as Glock layer)
  - Returns the new lock state and an array of bits – Cacheable, Need_S, Need_D, Need_E, Canceled
- *Unlock* – Unlock a lock
  - Lock Structure
  - Flags – Modified

Operations

- *Cancel* – Used to temporarily cancel a pending *lock* command during recovery
- *Reset* – Used to place a lock in a known state after a locking error
- *Reset_Expired* – Reset a machines expired locks

Operations

- Lock Value Block (LVB) operations
  - *hold_lvb* – Associate a LVB with a given lock
  - *unhold_lvb* – Disassociate LVB
  - *sync_lvb* – Write out the LVB contents to the network
  - LVBs are read in on acquiring a lock and are written when releasing an exclusive lock or calling *sync_lvb* when holding an exclusive lock

The Callback

- An asynchronous message from the Lock Module to GFS
  - Lets GFS know another machine needs a lock (either exclusively or shared)
  - Lets GFS know when there is a shortage of free locks and some cached locks need to be released
  - Lets GFS know about a failed client

Currently Implemented Protocols

- Nolock – Dummy locks for local file systems
- memexp – for DMEP SCSI devices and an IP DMEP server
- Future: DLM

Namespaces

- The lock modules are in charge of maintaining namespace data for each client in the cluster
- Each client has a bunch of attributes: Journal Number, Dlock Client ID, IP address for callbacks, stomith method, etc...
- GFS doesn’t know about any of these things
- The lock module (and the underlying cluster infrastructure) is in charge of maintaining this configuration data

Namespaces and Lock Modules

- For the IP Lock Server, the server takes care of these bindings
- For DMEP, they are recorded in a configuration Pool on the SAN
- A DLM would have its own method of storing this configuration.

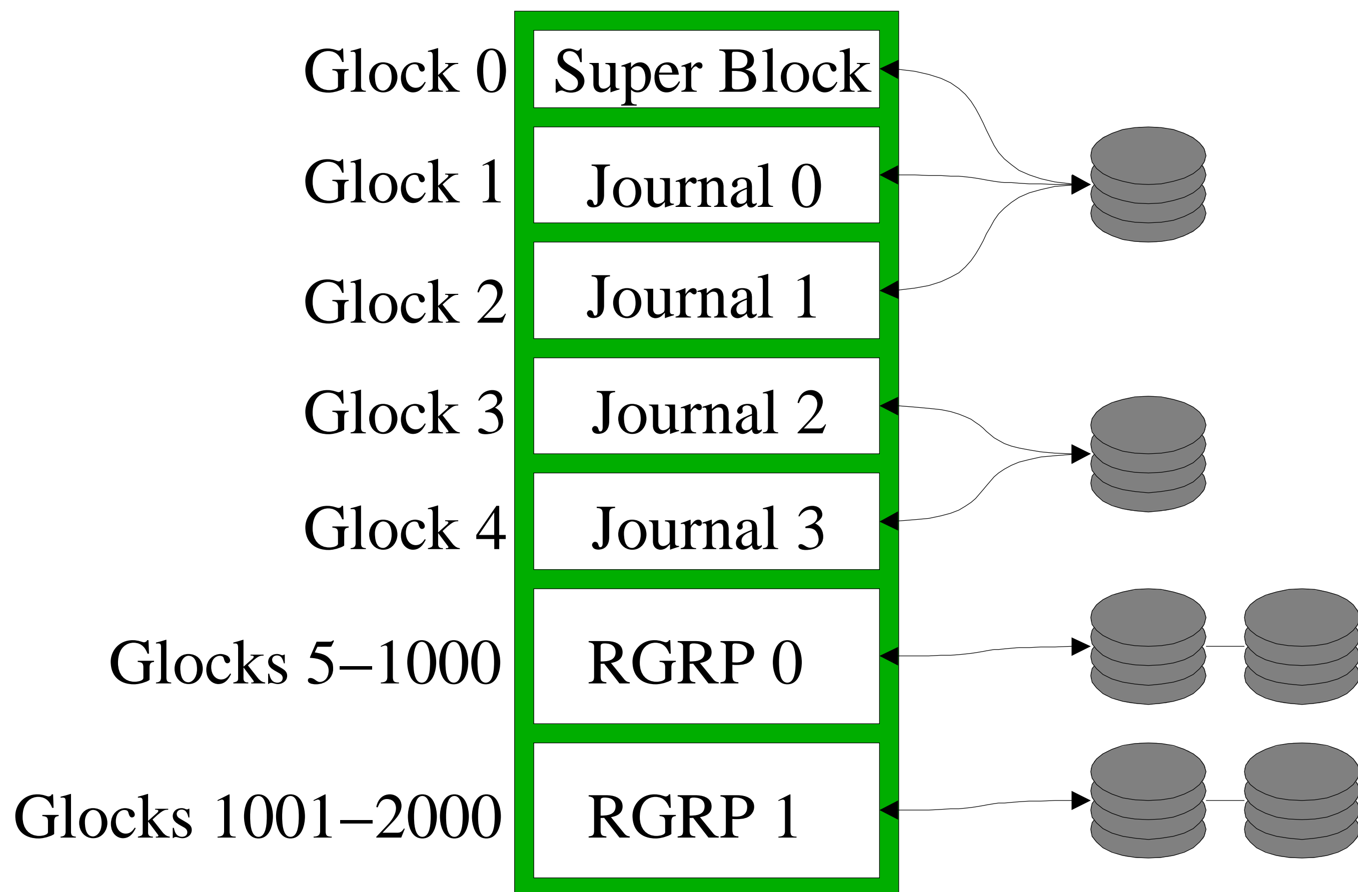
Recovery

- A FSCK is the classic means of recovery after a crash
  - Slow (time proportional to FS size)
  - The file system must be offline
  - Not acceptable for shared disk file systems
  - Now functional for GFS, will be improved
- Journaling solves these problems
  - Recovery time proportional to FS activity
  - Online recovery is possible

Layout for Journaling

- Having multiple clients share a journal is too complex and inefficient
- Each client gets its own journal space
- Each journal space is protected by one lock that is acquired at mount time and released at unmount (or crash) time.
- Each journal can be on its own disk for greater parallelism
- Each journal must be visible to all clients (for recovery)

GFS Layout



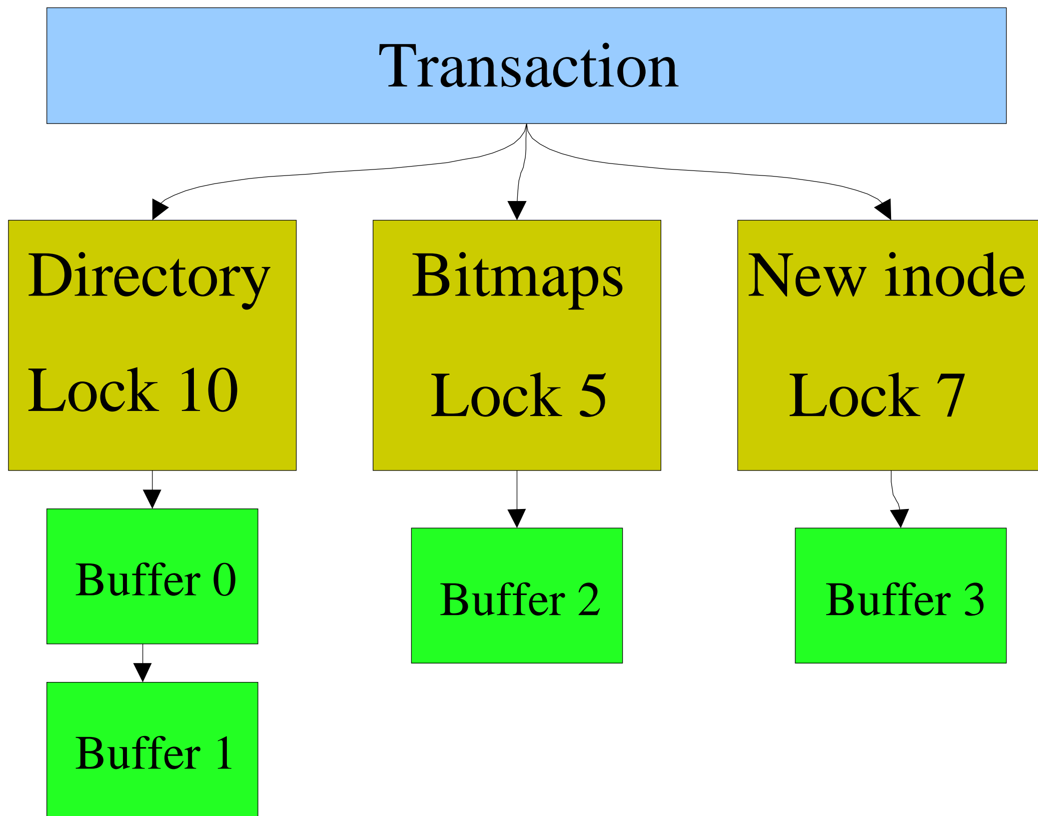
Journaling

- Asynchronous
  - Multiple journal entries are cached (and combined) in–core
  - Entries are committed to disk in groups asynchronously
  - Metadata buffers for a journal entry are pinned in memory (can’t be synced) until the entry is committed.
  - When journal write is complete, dirty metadata buffers can be synced

Journal Entries

- Composed of the metadata blocks changed during that operation (and a header)
- Each entry has one or more Glocks associated with it
  - Standard GFS locks that protect each piece of metadata
  - For instance, a creat() entry would have locks for the directory, the new dinode, and the bitmaps.

A Journal Entry (in memory)



Transaction Ordering

- Transactions that don’t have Glocks (or inodes) in common aren’t dependent on each other
- Independent transactions can be commited to the ondisk log in any order
- Directory renames are synchronized on a global rename lock
- As each transaction is commited to the incore log, it is combined with the transaction(s) that share its locks

Transaction Order Example

Dir A	Dir B	File X	File Y	RG 10	RG 11	Glock Operation
*		*		*		Create /A/X
	*		*		*	Create /B/Y
*	*	*				Rename /A/X /B/Q

Handling Lock Callbacks

- All journal entries are linked to one or more Glocks
- Before Glock is released to another machine:
  1. Flush incore transaction for Glock to log
  2. Sync in–place metadata buffers
  3. Sync in–place data buffers
- Only the transaction dependent on the requested Glock need to be flushed

Expiration

- Recovery based on lock expiration
  - Expired locks prevent a client from reading inconsistant data left by a failed client
  - A non–recovery lock operation blocks on an expired lock until it becomes unexpired
  - GFS will work with a DLM–style lock manager that doesn’t support expiration (It’s up to the lock module)

Recovery – Initiation

- Journaled recovery is initiated:
  - At mount time to check the journal the machine is mounting
  - When the lock module when detects an expired machine
- In each case, the expired client’s journal ID is passed to a recovery kernel thread
- The thread attempts to begin recovery by stomithing the failed client and trying to acquire its journal lock

Recovery – I/O Fencing

- A client which fails to heartbeat its locks but is still alive could do I/O while other machines are trying to recover for it.
  1) Machine A stalls
  2) Machine A’s locks expire
  3) Machine B decides A is dead, recovers A’s journal, and resets the locks A owned
  4) Machine A wakes up thinking it still holds locks and starts writing
  5) "Where did I put my backup tapes?"

I/O Fencing Methods

- Forcibly disable the failed client (STOMITH)
  - X10
  - Network Power Switches
  - Serial Port reset
- Prevent I/O from the zombie client from reaching the storage
  - Hard Zoning in Fibre Channel switches
  - SCSI Persistent Reservation
  - Banning in GNBD and IP DMEP server

Generation Numbers

- Each piece of metadata has a 64–bit Generation Number
- Each time the metadata is logged in a journal, its version number is incremented
- Allows the recovery code to determine if a piece of metadata has been modified since it was logged in a journal
- Metadata in the journal is only replayed if its generation number is bigger than the in–place copy

Recovery of the Journal

1) STOMITH the client
2) Acquire the failed client's journal lock
3) Acquire the transaction lock exclusively (stop all metadata modifications)
4) Find head and tail of the journal (i.e. the active region)
5) For each piece of metadata in the log:
   - If the generation number in the log is bigger than the in–place copy, replay the entry
6) Mark the journal as recovered, reset the expired locks for the failed client, and unlock the journal

Recovery

- The generation number allow journals to be replayed independently
- Allows easy handling of multiple simultaneous machine failures – just recover each journal sequentially
- Machines can continue to do read–only work during recovery unless they need a lock which was held by a failed client

Current State

- Production release out (4.0)
  - 1000+ downloads so far
  - Users include NASA, ILM, Fermilab, ISPs, ASPs, etc
- Try it!
  - www.globalfilesystem.org

Future Work

- File System Snapshotting
- Integration with (cluster) Linux LVM
- Ports to FreeBSD and other OSs (Solaris, PalmOS, etc...)
- Per–block locking
- Scalability: 32, 64, 128, ... 2^64
- Application level tuning: NFS and web serving clusters, etc...