Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

OCFS2: The Oracle Clustered File System, Version 2

Mark Fasheh
Oracle
mark.fasheh@oracle.com

Abstract

This talk will review the various components of the OCFS2 stack, with a focus on the file system and its clustering aspects. OCFS2 extends many local file system features to the cluster, some of the more interesting of which are posix unlink semantics, data consistency, shared readable mmap, etc.

In order to support these features, OCFS2 logically separates cluster access into multiple layers. An overview of the low level DLM layer will be given. The higher level file system locking will be described in detail, including a walkthrough of inode locking and messaging for various operations.

Caching and consistency strategies will be discussed. Metadata journaling is done on a per node basis with JBD. Our reasoning behind that choice will be described.

OCFS2 provides robust and performant recovery on node death. We will walk through the typical recovery process including journal replay, recovery of orphaned inodes, and recovery of cached metadata allocations.

Allocation areas in OCFS2 are broken up into groups which are arranged in self-optimizing "chains." The chain allocators allow OCFS2 to do fast searches for free space, and deallocation in a constant time algorithm. Detail on the layout and use of chain allocators will be given.

Disk space is broken up into clusters which can range in size from 4 kilobytes to 1 megabyte. File data is allocated in extents of clusters. This allows OCFS2 a large amount of flexibility in file allocation.

File metadata is allocated in blocks via a sub allocation mechanism. All block allocators in OCFS2 grow dynamically. Most notably, this allows OCFS2 to grow inode allocation on demand.

1 Design Principles

A small set of design principles has guided most of OCFS2 development. None of them are unique to OCFS2 development, and in fact, almost all are principles we learned from the Linux kernel community. They will, however, come up often in discussion of OCFS2 file system design, so it is worth covering them now.

1.1 Avoid Useless Abstraction Layers

Some file systems have implemented large abstraction layers, mostly to make themselves portable across kernels. The OCFS2 developers have held from the beginning that OCFS2 code would be Linux only. This has helped us in several ways. An obvious one is that it made

the code much easier to read and navigate. Development has been faster because we can directly use the kernel features without worrying if another OS implements the same features, or worse, writing a generic version of them.

Unfortunately, this is all easier said than done. Clustering presents a problem set which most Linux file systems don't have to deal with. When an abstraction layer is required, three principles are adhered to:

- Mimic the kernel API.
- Keep the abstraction layer as thin as possible.
- If object life timing is required, try to use the VFS object life times.

1.2 Keep Operations Local

Bouncing file system data around a cluster can be very expensive. Changed metadata blocks, for example, must be synced out to disk before another node can read them. OCFS2 design attempts to break file system updates into node local operations as much as possible.

1.3 Copy Good Ideas

There is a wealth of open source file system implementations available today. Very often during OCFS2 development, the question "How do other file systems handle it?" comes up with respect to design problems. There is no reason to reinvent a feature if another piece of software already does it well. The OCFS2 developers thus far have had no problem getting inspiration from other Linux file systems.[1] In some cases, whole sections of code have been lifted, with proper citation, from other open source projects!

[1] Most notably Ext3.

2 Disk Layout

Near the top of the `ocfs2_fs.h` header, one will find this comment:

```
/*
 * An OCFS2 volume starts this way:
 * Sector 0: Valid ocfs1_vol_disk_hdr that cleanly
 * fails to mount OCFS.
 * Sector 1: Valid ocfs1_vol_label that cleanly
 * fails to mount OCFS.
 * Block 2: OCFS2 superblock.
 *
 * All other structures are found
 * from the superblock information.
 */
```

The OCFS disk headers are the only amount of backwards compatibility one will find within an OCFS2 volume. It is an otherwise brand new cluster file system. While the file system basics are complete, there are many features yet to be implemented. The goal of this paper then, is to provide a good explanation of where things are in OCFS2 today.

2.1 Inode Allocation Structure

The OCFS2 file system has two main allocation units, *blocks* and *clusters*. Blocks can be anywhere from 512 bytes to 4 kilobytes, whereas clusters range from 4 kilobytes up to one megabyte. To make the file system mathematics work properly, cluster size is always greater than or equal to block size. At format time, the disk is divided into as many cluster-sized units as will fit. Data is always allocated in clusters, whereas metadata is allocated in blocks

Inode data is represented in extents which are organized into a b-tree. In OCFS2, extents are represented by a triple called an *extent record*.

Extent records are stored in a large in-inode array which extends to the end of the inode block. When the extent array is full, the file system will allocate an *extent block* to hold the current

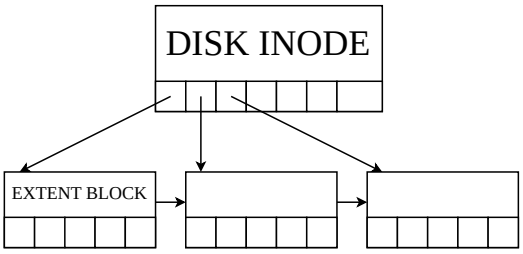

Figure 1: An Inode B-tree

Record Field	Field Size	Description
e_cpos	32 bits	Offset into the file, in clusters
e_clusters	32 bits	Clusters in this extent
e_blkno	64 bits	Physical disk offset

Table 1: OCFS2 extent record

array. The first extent record in the inode will be re-written to point to the newly allocated extent block. The e_clusters and e_cpos values will refer to the part of the tree underneath that extent. Bottom level extent blocks form a linked list so that queries accross a range can be done efficiently.

2.2 Directories

Directory layout in OCFS2 is very similar to Ext3, though unfortunately, htree has yet to be ported. The only difference in directory entry structure is that OCFS2 inode numbers are 64 bits wide. The rest of this section can be skipped by those already familiar with the dirent structure.

Directory inodes hold their data in the same manner which file inodes do. Directory data is arranged into an array of *directory entries*. Each directory entry holds a 64-bit inode pointer, a 16-bit record length, an 8-bit name length, an 8-bit file type enum (this allows us to avoid reading the inode block for type), and of course the set of characters which make up the file name.

2.3 The Super Block

The OCFS2 super block information is contained within an inode block. It contains a standard set of super block information—block size, compat/incompat/ro features, root inode pointer, etc. There are four values which are somewhat unique to OCFS2.

- s_clustersize_bits – Cluster size for the file system.
- s_system_dir_blkno – Pointer to the system directory.
- s_max_slots – Maximum number of simultaneous mounts.
- s_first_cluster_group – Block offset of first cluster group descriptor.

s_clustersize_bits is self-explanatory. The reason for the other three fields will be explained in the next few sections.

2.4 The System Directory

In OCFS2 file system metadata is contained within a set of *system files*. There are two types of system files, *global* and *node local*. All system files are linked into the file system via the hidden *system directory*[2] whose inode number is pointed to by the superblock. To find a system file, a node need only search the system directory for the name in question. The most common ones are read at mount time as a performance optimization. Linking to system files

[2] debugfs.ocfs2 can list the system dir with the ls // command.

from the system directory allows system file locatations to be completely dynamic. Adding new system files is as simple as linking them into the directory.

Global system files are generally accessible by any cluster node at any time, given that it has taken the proper cluster-wide locks. The `global_bitmap` is one such system file. There are many others.

Node local system files are said to be *owned* by a mounted node which occupies a unique *slot*. The maximum number of slots in a file system is determined by the `s_max_slots` superblock field. The `slot_map` global system file contains a flat array of node numbers which details which mounted node occupies which set of node local system files.

Ownership of a slot may mean a different thing to each node local system file. For some, it means that access to the system file is exclusive—no other node can ever access it. For others it simply means that the owning node gets preferential access—for an allocator file, this might mean the owning node is the only one allowed to allocate, while every node may delete.

A node local system file has its slot number encoded in the file name. For example, the journal used by the node occupying the third file system slot (slot numbers start at zero) has the name `journal:0002`.

2.5 Chain Allocators

OCFS2 allocates free disk space via a special set of files called *chain allocators*. Remember that OCFS2 allocates in clusters and blocks, so the generic term *allocation units* will be used here to signify either. The space itself is broken up into *allocation groups*, each of which contains a fixed number of allocation units. These

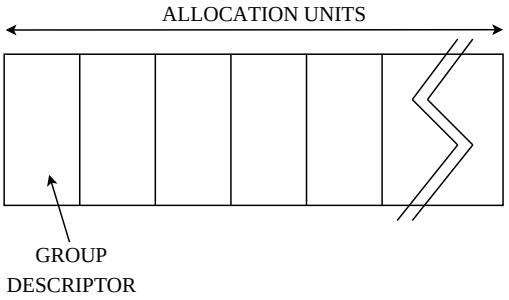

Figure 2: Allocation Group

groups are then *chained* together into a set of singly linked lists, which start at the allocator inode.

The first block of the first allocation unit within a group contains an `ocfs2_group_descriptor`. The descriptor contains a small set of fields followed by a bitmap which extends to the end of the block. Each bit in the bitmap corresponds to an allocation unit within the group. The most important descriptor fields follow.

- `bg_free_bits_count` – number of unallocated units in this group.
- `bg_chain` – describes which group chain this descriptor is a part of.
- `bg_next_group` – points to the next group descriptor in the chain.
- `bg_parent_dinode` – pointer to disk inode of the allocator which owns this group.

Embedded in the allocator inode is an `ocfs2_chain_list` structure. The chain list contains some fields followed by an array of `ocfs2_chain_rec` records. An `ocfs2_chain_rec` is a triple which describes a chain.

- `c_blkno` – First allocation group.

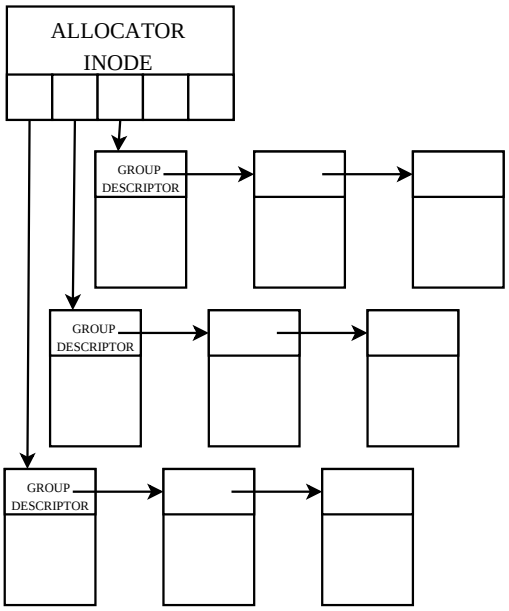

Figure 3: Chain Allocator

- `c_total` – Total allocation units.
- `c_free` – Free allocation units.

The two most interesting fields at the top of an `ocfs2_chain_list` are: `cl_cpg`, *clusters per group*; and `cl_bpc`, *bits per cluster*. The product of those two fields describes the total number of blocks occupied by each allocation group. As an example, the cluster allocator whose allocation units are clusters has a `cl_bpc` of 1 and `cl_cpg` is determined by `mkfs.ocfs2` (usually it just picks the largest value which will fit within a descriptor bitmap).

Chain searches are said to be self-optimizing. That is, while traversing a chain, the file system will re-link the group with the most number of free bits to the top of the list. This way, full groups can be pushed toward the end of the list and subsequent searches will require fewer disk reads.

2.6 Sub Allocators

The total number of file system clusters is largely static as determined by `mkfs.ocfs2` or optionally grown via `tunefs.ocfs2`. File system blocks however are dynamic. For example, an inode block allocator file can be grown as more files are created.

To grow a block allocator, `cl_bpc` clusters are allocated from the cluster allocator. The new `ocfs2_group_descriptor` record is populated and that block group is linked to the top of the smallest chain (wrapping back to the first chain if all are equally full). Other than the descriptor block, zeroing of the remaining blocks is skipped—when allocated, all file system blocks will be zeroed and written with a file system generation value. This allows `fsck.ocfs2` to determine which blocks in a group are valid metadata.

2.7 Local Alloc

Very early in the design of OCFS2 it was determined that a large amount of performance would be gained by reducing contention on the cluster allocator. Essentially the *local alloc* is a node local system file with an in-inode bitmap which caches clusters from the global cluster allocator. The local alloc file is **never** locked within the cluster—access to it is exclusive to a mounted node. This allows the block to remain valid in memory for the entire lifetime of a mount.

As the local alloc bitmap is exhausted of free space, an operation called a *window slide* is done. First, any unallocated bits left in the local alloc are freed back to the cluster allocator. Next, a large enough area of contiguous space is found with which to re-fill the local alloc. The cluster allocator bits are set, the local

alloc bitmap is cleared, and the size and offset of the new window are recorded. If no suitable free space is found during the second step of a window slide, the local alloc is disabled for the remainder of that mount.

The size of the local alloc bitmap is tuned at mkfs time to be large enough so that most block group allocations will fit, but the total size would not be so large as to keep an inordinate amount of data unallocatable by other nodes.

2.8 Truncate Log

The *truncate log* is a node local system file with nearly the same properties of the local alloc file. The major difference is that the truncate log is involved in *de*-allocation of clusters. This in turn dictates a difference in disk structure.

Instead of a small bitmap covering a section of the cluster allocator, the truncate log contains an in-inode array of `ocfs2_truncate_rec` structures. Each `ocfs2_truncate_rec` is an extent, with a start (`t_start`) cluster and a length (`t_clusters`). This structure allows the truncate log to cover large parts of the cluster allocator.

All cluster de-allocation goes through the truncate log. It is flushed when full, two seconds after the most recent de-allocation, or on demand by a `sync(2)` call.

3 Metadata Consistency

A large amount of time is spent inside a cluster file system keeping metadata blocks consistent. A cluster file system not only has to track and journal dirty blocks, but it must understand which clean blocks in memory are still valid with respect to any disk changes which other nodes might initiate.

3.1 Journaling

Journal files in OCFS2 are stored as node local system files. Each node has exclusive access to its journal, and retains a cluster lock on it for the duration of its mount.

OCFS2 does block based journaling via the JBD subsystem which has been present in the Linux kernel for several years now. This is the same journaling system in use by the Ext3 file system. Documentation on the JBD disk format can be found online, and is beyond the scope of this document.

Though the OCFS2 team could have invented their own journaling subsystem (which could have included some extra cluster optimizations), JBD was chosen for one main reason—stability. JBD has been very well tested as a result of being in use in Ext3. For any journaled file system, stability in its journaling layer is critical. To have done our own journaling layer at the time, no matter how good, would have inevitably introduced a much larger time period of unforeseen stability and corruption issues which the OCFS2 team wished to avoid.

3.2 Clustered Uptodate

The small amount of code (less than 550 lines, including a large amount of comments) in `fs/ocfs2/updtodate.c` attempts to mimic the `buffer_head` caching API while maintaining those properties across the cluster.

The Clustered Uptodate code maintains a small set of metadata caching information on every OCFS2 memory inode structure (`struct ocfs2_inode_info`). The caching information consists of a single `sector_t` per block. These are stored in a 2 item array unioned with a red-black tree root item `struct rb_root`. If the number of buffers that require tracking

grows larger than the array, then the red-black tree is used.

A few rules were taken into account before designing the Clustered Uptodate code:

1. All metadata changes are done under cluster lock.
2. All metadata changes are journaled.
3. All metadata reads are done under a read-only cluster lock.
4. Pinning `buffer_head` structures is not necessary to track their validity.
5. The act of acquiring a new cluster lock can flush metadata on other nodes and invalidate the inode caching items.

There are actually a very small number of exceptions to rule 2, but none of them require the Clustered Uptodate code and can be ignored for the sake of this discussion.

Rules 1 and 2 have the effect that the return code of `buffer_jbd()` can be relied upon to tell us that a `buffer_head` can be trusted. If it is in the journal, then we must have a cluster lock on it, and therefore, its contents are trustable.

Rule 4 follows from the logic that a newly allocated buffer head will not have its `BH_Uptodate` flag set. Thus one does not need to pin them for tracking purposes—a block number is sufficient.

Rule 5 instructs the Clustered Uptodate code to ignore `BH_Uptodate` buffers for which we do not have a tracking item—the kernel may think they're up to date with respect to disk, but the file system knows better.

From these rules, a very simple algorithm is implemented within `ocfs2_buffer_uptodate()`.

1. If `buffer_uptodate()` returns false, return false.
2. If `buffer_jbd()` returns true, return true.
3. If there is a tracking item for this block, return true.
4. Return false.

For existing blocks, tracking items are inserted after they are succesfully read from disk. Newly allocated blocks have an item inserted after they have been populated.

4 Cluster Locking

4.1 A Short DLM Tutorial

OCFS2 includes a DLM which exports a pared-down VMS style API. A full description of the DLM internals would require another paper the size of this one. This subsection will concentrate on a description of the important parts of the API.

A lockable object in the OCFS2 DLM is referred to as a *lock resource*. The DLM has no idea what is represented by that resource, nor does it care. It only requires a unique name by which to reference a given resource. In order to gain access to a resource, a process [3] acquires *locks* on it. There can be several locks on a resource at any given time. Each lock has a lock *level* which must be compatible with the levels of all other locks on the resource. All lock resources and locks are contained within a DLM *domain*.

[3] When we say *process* here, we mean a process which could reside on any node in the cluster.

Name	Access Type	Compatible Modes
EXMODE	Exclusive	NLMODE
PRMODE	Read Only	PRMODE, NLMODE
NLMODE	No Lock	EXMODE, PRMODE, NLMODE

Table 2: OCFS2 DLM lock Modes

In OCFS2, locks can have one of three levels, also known as lock *modes*. Table 2 describes each mode and its compatibility.

Most of the time, OCFS2 calls a single DLM function, `dlmlock()`. Via `dlmlock()` one can acquire a new lock, or *upconvert*, and *downconvert* existing locks.

```
typedef void (dlm_astlockfunc_t)(void *);

typedef void (dlm_bastlockfunc_t)(void *, int);

enum dlm_status dlmlock(
  struct dlm_ctxt *dlm,
  int mode,
  struct dlm_lockstatus *lksb,
  int flags,
  const char *name,
  dlm_astlockfunc_t *ast,
  void *data,
  dlm_bastlockfunc_t *bast);
```

Upconverting a lock asks the DLM to change its mode to a level greater than the currently granted one. For example, to make changes to an inode it was previously reading, the file system would want to upconvert its `PRMODE` lock to `EXMODE`. The currently granted level stays valid during an upconvert.

Downconverting a lock is the opposite of an upconvert—the caller wishes to switch to a mode that is more compatible with other modes. Often, this is done when the currently granted mode on a lock is incompatible with the mode another process wishes to acquire on its lock.

All locking operations in the OCFS2 DLM are asynchronous. Status notification is done via a set of callback functions provided in the arguments of a `dlmlock()` call. The two most important are the *AST* and *BAST* calls.

The DLM will call an `AST` function after a `dlmlock()` request has completed. If the status value on the `dlm_lockstatus` structure is `DLM_NORMAL` then the call has suceeded. Otherwise there was an error and it is up to the caller to decide what to do next.

The term `BAST` stands for *Blocking AST*. The `BAST` is the DLMs method of notifying the caller that a lock it is currently holding is blocking the request of another process.

As an example, if process A currently holds an `EXMODE` lock on resource *foo* and process B requests an `PRMODE` lock, process A will be sent a `BAST` call. Typically this will prompt process A to downconvert its lock held on *foo* to a compatible level (in this case, `PRMODE` or `NLMODE`), upon which an `AST` callback is triggered for both process A (to signify completion of the downconvert) and process B (to signify that its lock has been acquired).

The OCFS2 DLM supports a feature called *Lock Value Blocks*, or *LVBs* for short. An LVB is a fixed length byte array associated with a lock resource. The contents of the LVB are entirely up to the caller. There are strict rules to LVB access. Processes holding `PRMODE` and `EXMODE` locks are allowed to read the LVB value. Only processes holding `EXMODE` locks are allowed to write a new value to the LVB. Typically a read is done when acquiring or upconverting to a new `PRMODE` or `EXMODE` lock, while writes to the LVB are usually done when downconverting from an `EXMODE` lock.

4.2 DLM Glue

DLM glue (for lack of a better name) is a performance-critical section of code whose job it is to manage the relationship between the file system and the OCFS2 DLM. As such, DLM glue is the only part of the stack which knows about the internals of the DLM—regular file system code never calls the DLM API directly.

DLM glue defines several cluster lock types with different behaviors via a set of function pointers, much like the various VFS ops structures. Most lock types use the generic functions. The OCFS2 metadata lock defines most of its own operations for complexity reasons.

The most interesting callback that DLM glue requires is the *unblock* operation, which has the following definition:

int (∗unblock)(struct ocfs2_lock_res ∗, int ∗);

When a blocking AST is recieved for an OCFS2 cluster lock, it is queued for processing on a per-mount worker thread called the *vote thread*. For each queued OCFS2 lock, the vote thread will call its `unblock()` function. If possible the `unblock()` function is to downconvert the lock to a compatible level. If a downconvert is impossible (for instance the lock may be in use), the function will return a non-zero value indicating the operation should be retried.

By design, the DLM glue layer **never** determines lifetiming of locks. That is dictated by the container object—in OCFS2, this is predominantly the `struct inode` which already has a set of lifetime rules to be obeyed.

Similarly, DLM glue is **only** concerned with multi-node locking. It is up to the callers to serialize themselves locally. Typically this is done via well-defined methods such as holding `inode->i_mutex`.

The most important feature of DLM glue is that it implements a technique known as *lock caching*. Lock caching allows the file system to skip costly DLM communication for very large numbers of operations. When a DLM lock is created in OCFS2 it is never destroyed until the container object's lifetime makes it useless to keep around. Instead, DLM glue maintains its current mode and instead of creating new locks, calling processes only take references on a single cached lock. This means that, aside from the initial acquisition of a lock and barring any BAST calls from another node, DLM glue can keep most lock / unlock operations down to a single integer increment.

DLM glue will not block locking processes in the case of an upconvert—say a `PRMODE` lock is already held, but a process wants exclusive access in the cluster. DLM glue will continue to allow processes to acquire `PRMODE` level references while upconverting to `EXMODE`. Similarly, in the case of a downconvert, processes requesting access at the target mode will not be blocked.

4.3 Inode Locks

A very good example of cluster locking in OCFS2 is the inode cluster locks. Each OCFS2 inode has three locks. They are described in locking order, outermost first.

1. `ip_rw_lockres` which serializes file read and write operations.
2. `ip_meta_lockres` which protects inode metadata.
3. `ip_data_lockres` which protects inode data.

The inode metadata locking code is responsible for keeping inode metadata consistent across

the cluster. When a new lock is acquired at PRMODE or EXMODE, it is responsible for refreshing the `struct inode` contents. To do this, it stuffs the most common inode fields inside the lock LVB. This allows us to avoid a read from disk in some cases. The metadata `unblock()` method is responsible for waking up a checkpointing thread which forces journaled data to disk. OCFS2 keeps transaction sequence numbers on the inode to avoid checkpointing when unecessary. Once the checkpoint is complete, the lock can be downconverted.

The inode data lock has a similar responsibility for data pages. Complexity is much lower however. No extra work is done on acquiry of a new lock. It is only at downconvert that work is done. For a downconvert from EXMODE to PRMODE, the data pages are flushed to disk. Any downconvert to NLMODE truncates the pages and destroys their mapping.

OCFS2 has a cluster wide rename lock, for the same reason that the VFS has `s_vfs_rename_mutex`—certain combinations of `rename(2)` can cause deadlocks, even between multiple nodes. A comment in `ocfs2_rename()` is instructive:

```
/* Assume a directory hierarchy thusly:
 * a/b/c
 * a/d
 * a,b,c, and d are all directories.
 *
 * from cwd of 'a' on both nodes:
 * node1: mv b/c d
 * node2: mv d   b/c
 *
 * And that's why, just like the VFS, we need a
 * file system rename lock. */
```

Serializing operations such as `mount(2)` and `umount(2)` is the *super block lock*. File system membership changes occur only under an EXMODE lock on the super block. This is used to allow the mounting node to choose an appropriate slot in a race-free manner. The super block lock is also used during node messaging, as described in the next subsection.

4.4 Messaging

OCFS2 has a network *vote* mechanism which covers a small number of operations. The vote system stems from an older DLM design and is scheduled for final removal in the next major version of OCFS2. In the meantime it is worth reviewing.

Vote Type	Operation
OCFS2_VOTE_REQ_MOUNT	Mount notification
OCFS2_VOTE_REQ_UMOUNT	Unmount notification
OCFS2_VOTE_REQ_UNLINK	Remove a name
OCFS2_VOTE_REQ_RENAME	Remove a name
OCFS2_VOTE_REQ_DELETE	Query an inode wipe

Table 3: OCFS2 vote types

Each vote is broadcast to all mounted nodes (except the sending node) where they are processed. Typically vote messages about a given object are serialized by holding an EXMODE cluster lock on that object. That way the sending node knows it is the only one sending that exact vote. Other than errors, all votes except one return `true`. Membership is kept static during a vote by holding the super block lock. For mount/unmount that lock is held at EXMODE. All other votes keep a PRMODE lock. This way most votes can happen in parallel with respect to each other.

The mount/unmount votes instruct the other mounted OCFS2 nodes to the mount status of the sending node. This allows them in turn to track whom to send their own votes to.

The rename and unlink votes instruct receiving nodes to look up the dentry for the name being removed, and call the `d_delete()` function against it. This has the effect of removing the name from the system. If the vote is an unlink vote, the additional step of marking the inode as possibly orphaned is taken. The flag `OCFS2_`

INODE_MAYBE_ORPHANED will trigger additional processing in ocfs2_drop_inode(). This vote type is sent after all directory and inode locks for the operation have been acquired.

The delete vote is crucial to OCFS2 being able to support POSIX style unlink-while-open across the cluster. Delete votes are sent from ocfs2_delete_inode(), which is called on the last iput() of an orphaned inode. Receiving nodes simply check an *open count* on their inode. If the count is anything other than zero, they return a busy status. This way the sending node can determine whether an inode is ready to be truncated and deleted from disk.

5 Recovery

5.1 Heartbeat

The OCFS2 cluster stack heartbeats on disk and via its network connection to other nodes. This allows the cluster to maintain an idea of which nodes are alive at any given point in time. It is important to note that though they work closely together, the cluster stack is a separate entity from the OCFS2 file system.

Typically, OCFS2 disk heartbeat is done on every mounted volume in a contiguous set of sectors allocated to the heartbeat system file at file system create time. OCFS2 heartbeat actually knows nothing about the file system, and is only given a range of disk blocks to read and write. The system file is only used as a convenient method of reserving the space on a volume. Disk heartbeat is also never initiated by the file system, and always started by the mount.ocfs2 program. Manual control of OCFS2 heartbeat is available via the ocfs2_hb_ctl program.

Each node in OCFS2 has a unique node number, which dictates which heartbeat sector it will periodically write a timestamp to. Optimizations are done so that the heartbeat thread only reads those sectors which belong to nodes which are defined in the cluster configuration. Heartbeat information from all disks is accumulated together to determine node liveness. A node need only write to one disk to be considered alive in the cluster.

Network heartbeat is done via a set of keepalive messages that are sent to each node. In the event of a *split brain* scenario, where the network connection to a set of nodes is unexpectedly lost, a majority-based *quorum* algorithm is used. In the event of a 50/50 split, the group with the lowest node number is allowed to proceed.

In the OCFS2 cluster stack, disk heartbeat is considered the final arbiter of node liveness. Network connections are built up when a node begins writing to its heartbeat sector. Likewise network connections will be torn down when a node stops heartbeating to all disks.

At startup time, interested subsystems register with the heartbeat layer for *node up* and *node down* events. Priority can be assigned to callbacks and the file system always gets node death notification before the DLM. This is to ensure that the file system has the ability to mark itself needing recovery before DLM recovery can proceed. Otherwise, a race exists where DLM recovery might complete before the file system notification takes place. This could lead to the file system gaining locks on resources which are in need of recovery—for instance, metadata whose changes are still in the dead node's journal.

5.2 File System Recovery

Upon notification of an unexpected node death, OCFS2 will mark a *recovery bitmap*. Any file system locks which cover recoverable resources have a check in their locking path for any set bits in the recovery bitmap. Those paths will then block until the bitmap is clear again. Right now the only path requiring this check is the metadata locking code—it must wait on journal replay to continue.

A recovery thread is then launched which takes a `EXMODE` lock on the super block. This ensures that only one node will attempt to recover the dead node. Additionally, no other nodes will be allowed to mount while the lock is held. Once the lock is obtained, each node will check the `slot_map` system file to determine which journal the dead node was using. If the node number is not found in the slot map, then that means recovery of the node was completed by another cluster node.

If the node is still in the slot map then journal replay is done via the proper JBD calls. Once the journal is replayed, it is marked clean and the node is taken out of the slot map.

At this point, the most critical parts of OCFS2 recovery are complete. Copies are made of the dead node's truncate log and local alloc files, and clean ones are stamped in their place. A worker thread is queued to reclaim the disk space represented in those files, the node is removed from the recovery bitmap and the super block lock is dropped.

The last part of recovery—replay of the copied truncate log and local alloc files—is (appropriately) called *recovery completion*. It is allowed to take as long as necessary because locking operations are not blocked while it runs. Recovery completion is even allowed to block on recovery of other nodes which may die after its work is queued. These rules greatly simplify the code in that section.

One aspect of recovery completion which has not been covered yet is *orphan recovery*. The orphan recovery process must be run against the dead node's orphan directory, as well as the local orphan directory. The local orphan directory is recovered because the now dead node might have had open file descriptors against an inode which was locally orphaned—thus the `delete_inode()` code must be run again.

Orphan recovery is a fairly straightforward process which takes advantage of the existing inode life-timing code. The orphan directory in question is locked, and the recovery completion process calls `iget()` to obtain an inode reference on each orphan. As references are obtained, the orphans are arranged in a singly linked list. The orphan directory lock is dropped, and `iput()` is run against each orphan.

6 What's Been Missed!

Lots, unfortunately. The DLM has mostly been glossed over. The rest of the OCFS2 cluster stack has hardly been mentioned. The OCFS2 tool chain has some unique properties which would make an interesting paper. Readers interested in more information on OCFS2 are urged to explore the web page and mailing lists found in the references section. OCFS2 development is done in the open and when not busy, the OCFS2 developers love to answer questions about their project.

7 Acknowledgments

A huge thanks must go to all the authors of Ext3 from which we took much of our inspiration.

Also, without JBD OCFS2 would not be what it is today, so our thanks go to those involved in its development.

Of course, we must thank the Linux kernel community for being so kind as to accept our humble file system into their kernel. In particular, our thanks go to Christoph Hellwig and Andrew Morton whose guidance was critical in getting our file system code up to kernel standards.

8 References

The OCFS2 home page can be found at `http://oss.oracle.com/projects/ocfs2/`.

From there one can find mailing lists, documentation, and the source code repository.