MD Cluster

The cluster MD is a shared-device RAID for a cluster, it supports two levels: raid1 and raid10 (limited support).

1. On-disk format

Separate write-intent-bitmaps are used for each cluster node. The bitmaps record all writes that may have been started on that node, and may not yet have finished. The on-disk layout is:

```
0                    4k                     8k                    12k
-------------------------------------------------------------------
| idle                | md super            | bm super [0] + bits |
| bm bits[0, contd]   | bm super[1] + bits  | bm bits[1, contd]   |
| bm super[2] + bits  | bm bits [2, contd]  | bm super[3] + bits  |
| bm bits [3, contd]  |                     |                     |
```

During "normal" functioning we assume the filesystem ensures that only one node writes to any given block at a time, so a write request will

- set the appropriate bit (if not already set)
- commit the write to all mirrors
- schedule the bit to be cleared after a timeout.

Reads are just handled normally. It is up to the filesystem to ensure one node doesn't read from a location where another node (or the same node) is writing.

2. DLM Locks for management

There are three groups of locks for managing the device:

2.1 Bitmap lock resource (bm_lockres)

The bm_lockres protects individual node bitmaps. They are named in the form bitmap000 for node 1, bitmap001 for node 2 and so on. When a node joins the cluster, it acquires the lock in PW mode and it stays so during the lifetime the node is part of the cluster. The lock resource number is based on the slot number returned by the DLM subsystem. Since DLM starts node count from one and bitmap slots start from zero, one is subtracted from the DLM slot number to arrive at the bitmap slot number.

The LVB of the bitmap lock for a particular node records the range of sectors that are being re-synced by that node. No other node may write to those sectors. This is used when a new nodes joins the cluster.

2.2 Message passing locks

Each node has to communicate with other nodes when starting or ending resync, and for metadata superblock updates. This communication is managed through three locks: "token", "message", and "ack", together with the Lock Value Block (LVB) of one of the "message" lock.

2.3 new-device management

A single lock: "no-new-dev" is used to co-ordinate the addition of new devices - this must be synchronized across the array. Normally all nodes hold a concurrent-read lock on this device.

3. Communication

Messages can be broadcast to all nodes, and the sender waits for all other nodes to acknowledge the message before proceeding. Only one message can be processed at a time.

3.1 Message Types

There are six types of messages which are passed:

3.1.1 METADATA_UPDATED

informs other nodes that the metadata has been updated, and the node must re-read the md superblock. This is performed synchronously. It is primarily used to signal device failure.

3.1.2 RESYNCING

informs other nodes that a resync is initiated or ended so that each node may suspend or resume the region. Each RESYNCING message identifies a range of the devices that the sending node is about to resync. This overrides any previous notification from that node: only one ranged can be resynced at a time per-node.

3.1.3 NEWDISK

informs other nodes that a device is being added to the array. Message contains an identifier for that device. See below for further details.

3.1.4 REMOVE

A failed or spare device is being removed from the array. The slot-number of the device is included in the message.

3.1.5 RE_ADD:

A failed device is being re-activated - the assumption is that it has been determined to be working again.

3.1.6 BITMAP_NEEDS_SYNC:

If a node is stopped locally but the bitmap isn't clean, then another node is informed to take the ownership of resync.

3.2 Communication mechanism

The DLM LVB is used to communicate within nodes of the cluster. There are three resources used for the purpose:

3.2.1 token

The resource which protects the entire communication system. The node having the token resource is allowed to communicate.

3.2.2 message

The lock resource which carries the data to communicate.

3.2.3 ack

The resource, acquiring which means the message has been acknowledged by all nodes in the cluster. The BAST of the resource is used to inform the receiving node that a node wants to communicate.

The algorithm is:

1. receive status - all nodes have concurrent-reader lock on "ack":

```
sender                          receiver                receiver
"ack":CR                        "ack":CR                "ack":CR
```

2. sender get EX on "token", sender get EX on "message":

```
sender                         receiver                receiver
"token":EX                     "ack":CR                "ack":CR
"message":EX
"ack":CR
```

   Sender checks that it still needs to send a message. Messages received or other events that happened while waiting for the "token" may have made this message inappropriate or redundant.

3. sender writes LVB

   sender down-convert "message" from EX to CW

   sender try to get EX of "ack"

```
    [ wait until all receivers have *processed* the "message" ]

                                    [ triggered by bast of "ack" ]
                                    receiver get CR on "message"
                                    receiver read LVB
                                    receiver processes the message
                                    [ wait finish ]
                                    receiver releases "ack"
```

```
                                        receiver tries to get PR on "message"

    sender                          receiver                  receiver
    "token":EX                      "message":CR              "message":CR
    "message":CW
    "ack":EX
```

4. triggered by grant of EX on "ack" (indicating all receivers have processed message)

   sender down-converts "ack" from EX to CR

   sender releases "message"

   sender releases "token"

```
                                receiver upconvert to PR on "message"
                                receiver get CR of "ack"
                                receiver release "message"

    sender                      receiver                  receiver
    "ack":CR                    "ack":CR                  "ack":CR
```

4. Handling Failures

4.1 Node Failure

When a node fails, the DLM informs the cluster with the slot number. The node starts a cluster recovery thread. The cluster recovery thread:

- acquires the bitmap<number> lock of the failed node
- opens the bitmap
- reads the bitmap of the failed node
- copies the set bitmap to local node
- cleans the bitmap of the failed node
- releases bitmap<number> lock of the failed node
- initiates resync of the bitmap on the current node md_check_recovery is invoked within recover_bitmaps, then md_check_recovery -> metadata_update_start/finish, it will lock the communication by lock_comm. Which means when one node is resyncing it blocks all other nodes from writing anywhere on the array.

The resync process is the regular md resync. However, in a clustered environment when a resync is performed, it needs to tell other nodes of the areas which are suspended. Before a resync starts, the node send out RESYNCING with the (lo,hi) range of the area which needs to be suspended. Each node maintains a suspend_list, which contains the list of ranges which are currently suspended. On receiving RESYNCING, the node adds the range to the suspend_list. Similarly, when the node performing resync finishes, it sends RESYNCING with an empty range to other nodes and other nodes remove the corresponding entry from the suspend_list.

A helper function, ->area_resyncing() can be used to check if a particular I/O range should be suspended or not.

4.2 Device Failure

Device failures are handled and communicated with the metadata update routine. When a node detects a device failure it does not allow any further writes to that device until the failure has been acknowledged by all other nodes.

5. Adding a new Device

For adding a new device, it is necessary that all nodes "see" the new device to be added. For this, the following algorithm is used:

1. Node 1 issues mdadm --manage /dev/mdX --add /dev/sdYY which issues ioctl(ADD_NEW_DISK with disc.state set to MD_DISK_CLUSTER_ADD)
2. Node 1 sends a NEWDISK message with uuid and slot number
3. Other nodes issue kobject_uevent_env with uuid and slot number (Steps 4,5 could be a udev rule)
4. In userspace, the node searches for the disk, perhaps using blkid -t SUB_UUID=""
5. Other nodes issue either of the following depending on whether the disk was found: ioctl(ADD_NEW_DISK with disc.state set to MD_DISK_CANDIDATE and disc.number set to slot number) ioctl(CLUSTERED_DISK_NACK)
6. Other nodes drop lock on "no-new-devs" (CR) if device is found
7. Node 1 attempts EX lock on "no-new-dev"
8. If node 1 gets the lock, it sends METADATA_UPDATED after unmarking the disk as SpareLocal
9. If not (get "no-new-dev" lock), it fails the operation and sends METADATA_UPDATED.

10. Other nodes get the information whether a disk is added or not by the following METADATA_UPDATED.

6. Module interface

There are 17 call-backs which the md core can make to the cluster module. Understanding these can give a good overview of the whole process.

6.1 join(nodes) and leave()

These are called when an array is started with a clustered bitmap, and when the array is stopped. join() ensures the cluster is available and initializes the various resources. Only the first 'nodes' nodes in the cluster can use the array.

6.2 slot_number()

Reports the slot number advised by the cluster infrastructure. Range is from 0 to nodes-1.

6.3 resync_info_update()

This updates the resync range that is stored in the bitmap lock. The starting point is updated as the resync progresses. The end point is always the end of the array. It does *not* send a RESYNCING message.

6.4 resync_start(), resync_finish()

These are called when resync/recovery/reshape starts or stops. They update the resyncing range in the bitmap lock and also send a RESYNCING message. resync_start reports the whole array as resyncing, resync_finish reports none of it.

resync_finish() also sends a BITMAP_NEEDS_SYNC message which allows some other node to take over.

6.5 metadata_update_start(), metadata_update_finish(), metadata_update_cancel()

metadata_update_start is used to get exclusive access to the metadata. If a change is still needed once that access is gained, metadata_update_finish() will send a METADATA_UPDATE message to all other nodes, otherwise metadata_update_cancel() can be used to release the lock.

6.6 area_resyncing()

This combines two elements of functionality.

Firstly, it will check if any node is currently resyncing anything in a given range of sectors. If any resync is found, then the caller will avoid writing or read-balancing in that range.

Secondly, while node recovery is happening it reports that all areas are resyncing for READ requests. This avoids races between the cluster-filesystem and the cluster-RAID handling a node failure.

6.7 add_new_disk_start(), add_new_disk_finish(), new_disk_ack()

These are used to manage the new-disk protocol described above. When a new device is added, add_new_disk_start() is called before it is bound to the array and, if that succeeds, add_new_disk_finish() is called the device is fully added.

When a device is added in acknowledgement to a previous request, or when the device is declared "unavailable", new_disk_ack() is called.

6.8 remove_disk()

This is called when a spare or failed device is removed from the array. It causes a REMOVE message to be send to other nodes.

6.9 gather_bitmaps()

This sends a RE_ADD message to all other nodes and then gathers bitmap information from all bitmaps. This combined bitmap is then used to recovery the re-added device.

6.10 lock_all_bitmaps() and unlock_all_bitmaps()

These are called when change bitmap to none. If a node plans to clear the cluster raid's bitmap, it need to make sure no other nodes are using the raid which is achieved by lock all bitmap locks within the cluster, and also those locks are unlocked accordingly.

7. Unsupported features

There are somethings which are not supported by cluster MD yet.

- change array_sectors.