Solaris Volume Manager: Multi-Owner Disksets

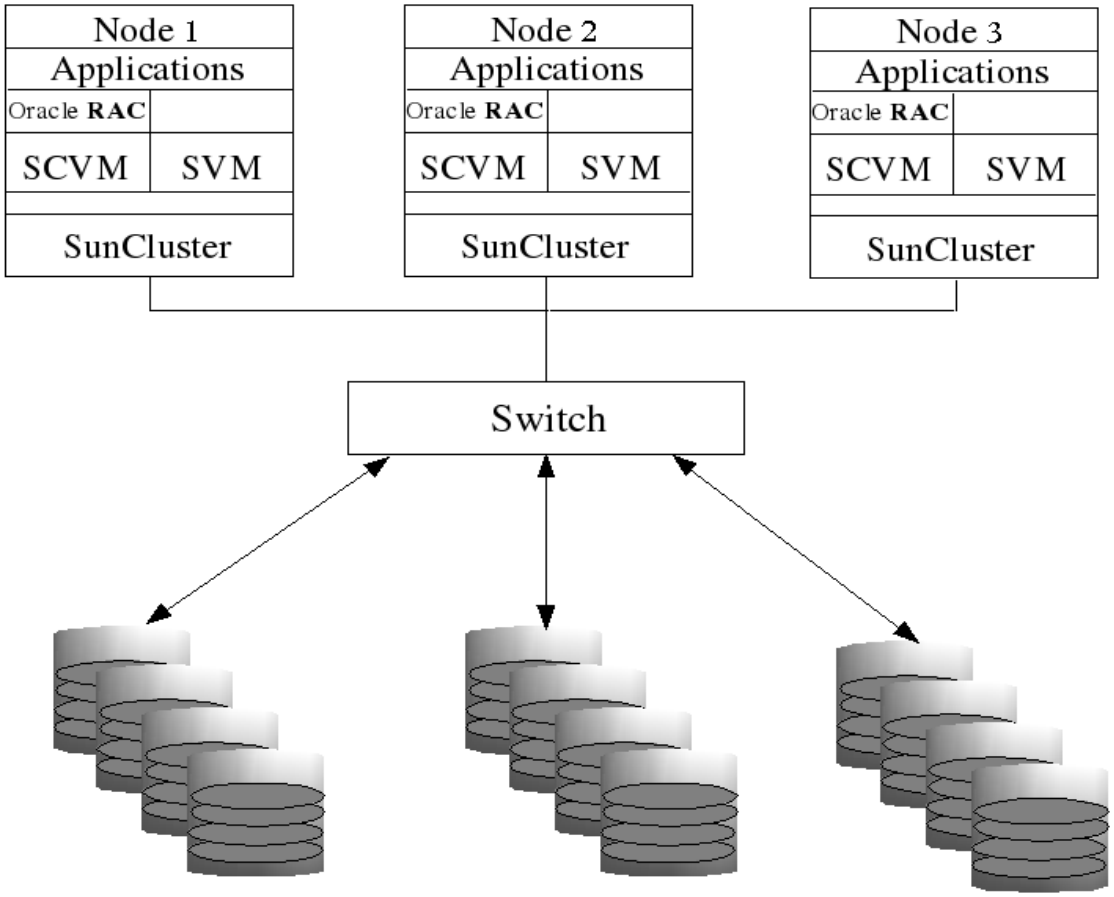
Why Multi-Owner Disksets

- Requested by SunCluster to offer alternative solution to Veritas Cluster Server
- Project name “Oban”
- Created to support Oracle RAC
- User application must handle overlapping writes since there is no cluster wide lock mgr in Oban

What are MO Disksets?

- Manages storage grouped as multi-owner disk sets
- Multiple nodes can perform I/O to shared storage simultaneously
- Multi-owner disk set functionality is enabled only in a SunCluster environment

Solaris Volume Manager for Sun Cluster Overview



MO Diskset Limits

- Multi-owner disk sets:
  - Can support up to 128 nodes
  - Support a maximum of 8192 volumes
  - Have a larger state database replica. The default is 16 Mbytes, with a limit of 256 Mbytes
- Maximum number of disk sets still are 32
- No support for RAID or trans

Commands and Daemons

- Metaclust
  - Only called by reconfig scripts
- Metaset
  - -M Creates a multi-owner disk set
  - -j/-w Join/withdraw to/from diskset
- rpc.mdcommd
  - Coordinates node communication
- Mddoors

Common Terms

- Master – Each multi-node diskset has a master node that acts as the controller for all configuration updates to the cluster.
- Slave – All of the nodes in a multi-node diskset that are not a master. It is possible for a node to be a master of one diskset and a slave for another.
- Metadb, MDDB, replica – the metadata storage for SVM

MO Diskset Concepts

- Each MO diskset has a master
  - Multiple masters can exist simultaneously in different sets
  - Master manages and updates the state database replica changes for its set
- Two ways the master is chosen
  - First node to add a disk to the disk set
  - The node with the lowest node id

Mirrors in MO disksets

Two types of mirrors:

- Optimized resync mirrors
  - Only accessed by 1 node at a time
  - Intended for use of Oracle RAC log files that are only accessed by 1 node.
- ABR enabled mirrors
  - Can be accessed by all nodes simultaneously

Mirrors in MO Disksets (2)

- Mirror Owners
  - A mirror owner is the node that is currently performing a resync or with optimized resync mirrors the last node to perform i/o
  - Optimized resync mirrors always have an owner
  - ABR enabled mirrors have an owner only during resyncs

Optimized Resync Mirror

metastat -s blue

blue/d22: Mirror

Submirror 0: blue/d20

State: Okay

Submirror 1: blue/d21

State:Okay

Pass: 1

Read option: roundrobin (default)

Write option: parallel (default)

Resync option: optimized resync

Owner: nodetwo

Size 1238895 blocks (604 MB)

ABR Enabled Mirror

```
# metastat -s blue
blue/d22: Mirror
   Submirror 0: blue/d20
      State: Okay
   Submirror 1: blue/d21
      State:Okay
   Pass: 1
   Read option: roundrobin (default)
   Write option: parallel (default)
   Resync option: application based
   Owner: None
   Size 1238895 blocks (604 MB)
```

Creation of MN diskset

- Similar to creation of traditional diskset (uses rpc.metad)
- Starts rpc.mdcommd and mddoors
  - rpc.mdcommd is an rpc daemon used to communicate configuration and state changes across the cluster nodes
  - mdoors is a door interface between the kernel and rpc.mdcommd
- Subsequent metaset and metadb commands use rpc.mdcommd in addition to rpc.metad

Metaset output after creation

```
# metaset -s foo1 -aMh staffa ulva
# metaset
Multi-owner Set name = foo1, Set number = 1, Master =

Host                    Owner           Member
  staffa                                Yes
  ulva                                  Yes
```

- No owner since no disk in diskset
- Both nodes are members
- No master since no node is an owner

Owner and Members

- Multiple owners allowed in MN diskset
- Node is a member if node is in the membership list (/var/run/nodelist)
- Node is always a member on a single node system with no membership list

Membership File

```
# cat /var/run/nodelist
1 staffa 172.16.193.1
2 ulva 172.16.193.2
```

- The node number starts at 1
- Written during reconfig cycle
- SunCluster will create and maintain this file so if running/testing in a non-SC environment it is necessary to create/maintain this manually

Join and Withdraw

- Options -j/-w like -t/-r
- First node to join is master
- Master not allowed to withdraw until all other nodes have withdrawn
- Customers not expected to use since SunCluster reconfig automatically joins nodes

Join and Withdraw (2)

- Slave node inherits state of diskset
- If master is running at 50% available mddbs, slave node is allowed to join diskset
- If master is running in STALE state when slave node joins, slave node is also in STALE state

Join and Withdraw (3)

```
ulva# metaset -w
ulva# metaset

Multi-owner Set name = foo1, Set number = 1, Master =
Master and owner information unavailable until joined
  (metaset -j)

Host                    Owner           Member
  staffa                 multi-owner    Yes
  ulva                                  Yes

Drive      Dbase
c1t16d0    Yes
```

- Ulva no longer owner

Join and Withdraw (4)

```
Staffa# metaset

Multi-owner Set name = foo1, Set number = 1, Master =
  staffa

Host                    Owner           Member
  staffa                 multi-owner    Yes
  ulva                                  Yes

Drive      Dbase
c1t16d0    Yes
```

- Staffa is still owner and master

Node Records

- Each node in MN diskset has a node record stored in local metadbs
- Node record added/deleted during addition or removal of host from set
- Node record updated during join/withdraw of host

Diskset Failure Recovery

- Recovery is more comprehensive due to leverage of SunCluster reconfig cycle
- Nodes communicate during recovery to develop consistent view
- Should never have situation where 2 nodes have different views of diskset

Metadb

- Only master node allowed to write to diskset mddbs
- When master changes the mddb it sends a PARSE message which causes the slaves to re-read the mddb in from disk
- Optimized resync record updates do not cause a PARSE message

Metaset and Metadb

- These commands are only run on the local node
- These commands can change the nodelist and so can't use rpc.mdcommd
- These commands suspend rpc.mdcommd in order to lock out other meta* commands.

MO Diskset Commands

- Most can be run from any node
- Metarecover must be run on master node
- Metastat doesn't contact other nodes in multi-owner set
- Most are executed on all nodes using rpc.mdcommd

MO Diskset Commands (2)

- Most run a -n (dryrun) option first
- If dryrun fails, command is done.
- Once a command (non-dryrun) is issued on master node, it will be issued to all nodes even if command failed.

MO Diskset Commands (3)

- When invoked by rpc.mdcommd, exec name is metaxxxx.rpc_call. This enables the command to determine if it has been invoked by the user or by rpc.mdcommd.
- The command must return identical values on all nodes, if not, rpc.mdcommd will Panic the Master node.
- A disk failure on one node is reflected as a failure on all the nodes.

Metainit example

- User types “metainit -sA d0 1 1 c1t3d0s0”
- Metainit sends dryrun command to rpc.mdcommd on local node.
- rpc.mdcommd sends message to Master.
- Master executes dryrun command locally.
- If successful, dryrun command sent to all slave nodes, including originating node.
- If dryrun fails on any node, failure is returned to the originator which aborts command.

Metainit example (2)

- If dryrun succeeds, metainit sends the metainit to rpc.mdcommd locally.
- rpc.mdcommd sends message to Master.
- Master executes metainit command locally, metadbs updated.
- Command then sent to all slave nodes
- If the result on any node differs from the result on the Master, rpc.mdcommd forces a panic of the Master node as there is an inconsistency between the two nodes.

Metainit example (3)

- We now have identical results for the command, return the result to the originating metainit command. Note that it may be that the command fails on all nodes, this is an acceptable outcome.

Multiple Argument Cmds

- Metaclear and metasync can refer to multiple metadevices, either explicitly
  metaclear -sA d1 d2 d3
  metasync -sA d10 d11
  or implicitly
  metaclear sA -a, metasync -sA -r
- Here, the originating command, sends a series of individual metacommands to the Master node. This avoids the requirement for a long timeout for the messages.

Oban Libmeta Support

- Libmeta functions used by metacommands
  - meta_mn_send_command()
  - meta_mn_send_resync_starting()
  - meta_mn_suspend_writes()
  - meta_mn_send_setsync() (S10)
  - meta_mn_send_metaclear_command()
  - meta_is_mn_set() - also sets sp to current set if set/dxx

rpc.mdcommd

- Communication daemon
- Runs on every node
- Accepts messages
- Guarantees response to message
- Class oriented (classes 1- 8)
- Set oriented (1- 32)

rpc.mdcommd Classes

- rpc.mdcommd can only process one message per set/class at a time.
- Classes enable the processing of one message to send another message.

rpc.mdcommd Classes (2)

- While processing one message, a message can be generated with a higher class than the originating message.

Components of rpc.mdcommd

- Local rpc.mdcommd accepts requests from initiators
- Master rpc.mdcommd accepts messages from local rpc.mdcommd on all nodes
- Slave rpc.mdcommd accepts messages from Master rpc.mdcommd

Using rpc.mdcommd

- User level initiator uses mdmn_send_message
- Kernel level initiator uses mdmn_ksend_message
- Kernel request sent to local rpc.mdcommd via mddoors
- Initiator can hold no locks across send

Change log

- Implemented as user records in diskset mddb (previously user records only used in local set)
- 16 user records allocated when diskset mddb is created (2 for each class)
- Only accessed by master node
- Persists across full cluster reboot

Message Completion Table

- Memory mapped file on each node (not persistent across boot)
- Holds ID of last message completed on this node for all message types and their results
- Used when replaying messages during reconfig cycle so message isn't executed twice

Mddoors

- Fast kernel to user interface for rpc.mdcommd messages
- Less complex and just as fast as kernel RPC implementation

Message Handlers

- Each message has a handler or a submessage generator
- Handler can only cause higher priority class messages to be sent
- Handlers are run on master and slave nodes unless specific flags are set

Message Handlers(2)

- Most message handlers make ioctl calls into SVM drivers.
- SVM ioctls are single-threaded, hence deadlock if sending a message from within an ioctl.
- Hence we implement multi-threaded ioctls.

Multi-threaded ioctls

- Multi-threaded ioctl can be executed while a single threaded ioctl is active.
- Single-threaded ioctl cannot be executed while multi-threaded active.
- Multiple multi-threaded ioctls can be concurrently active.

Submessage Generator

Submessage generator is used for a multipart message that needs to be entered into the change log as one message but logically breaks out into smaller submessages.

The smaller submessages are not logged.

Special rpc.mdcommd routines

- Lock, Unlock
  - > used during test
- Suspend, Reinit, Resume, Ping
  - > used when nodelist is being manipulated by metaset and metadb commands. Forces rpc.mdcommd to get new nodelist from rpc.metad

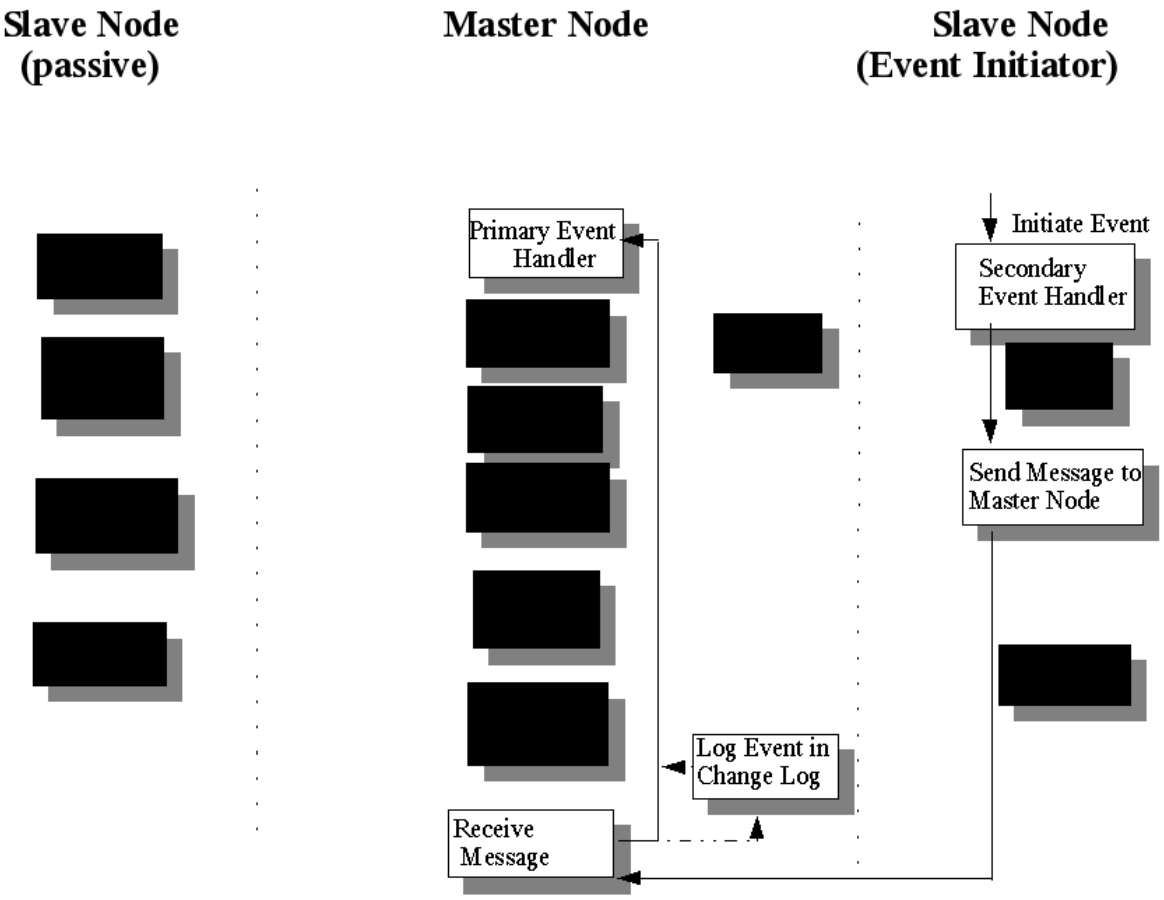
Lock Issues

- No locks should be held across call to send a message to rpc.mdcommd
- Lots of changes made to code to stop deadlock situations
- There are a few places where a lock must be held while sending a message but beware!

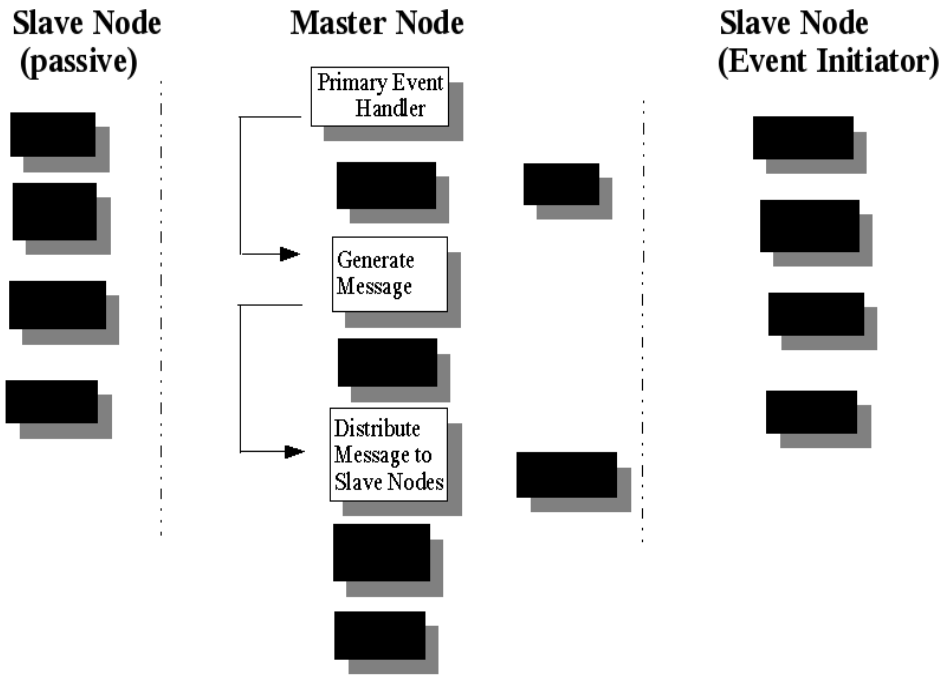
rpc.mdcommd debugging

- rpc.mdcommd debug output is setup by adding the following lines to /etc/lvm/runtime.cf and restarting rpc.mdcommd
  - commd_out_file=/commd_log/commd.out
  - commd_verbosity=0x2000ffff
- The commd_out_file isn't reset at reboot and can become quite large.

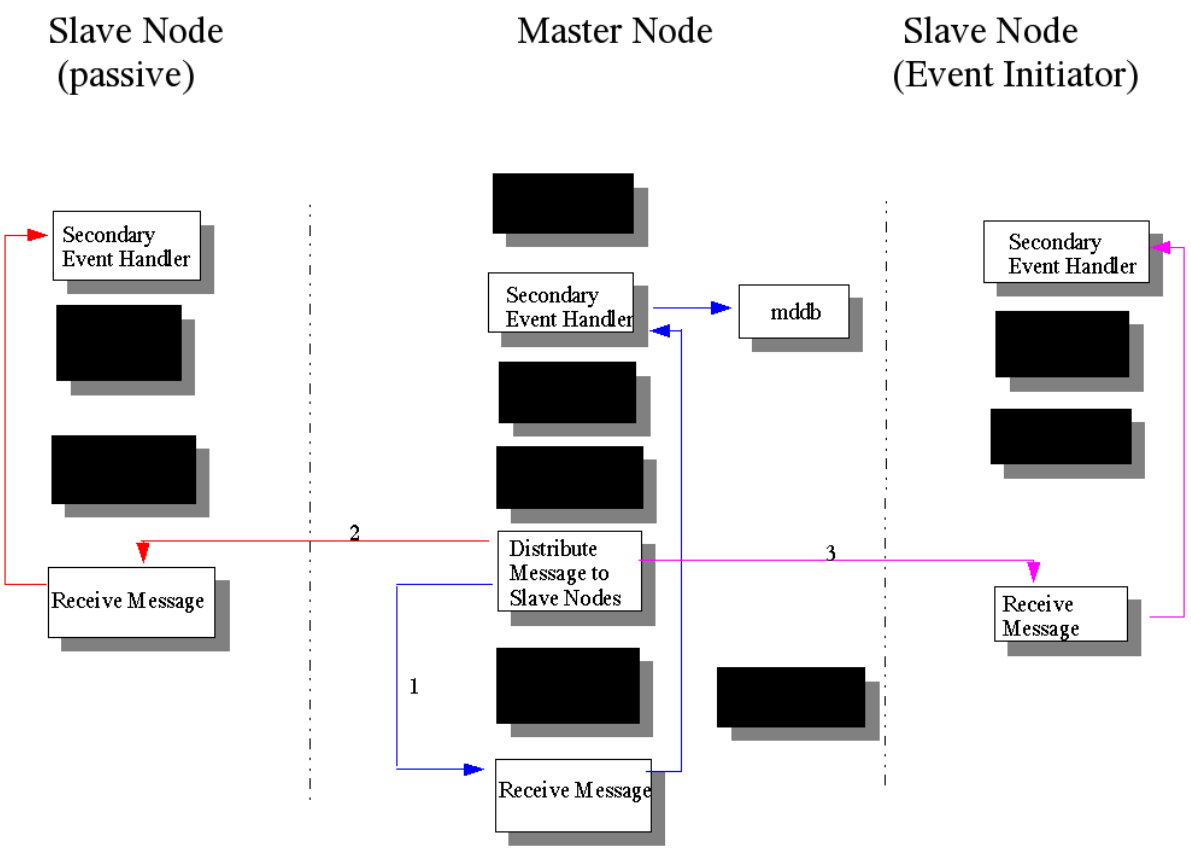
Event Flow – Phase 1



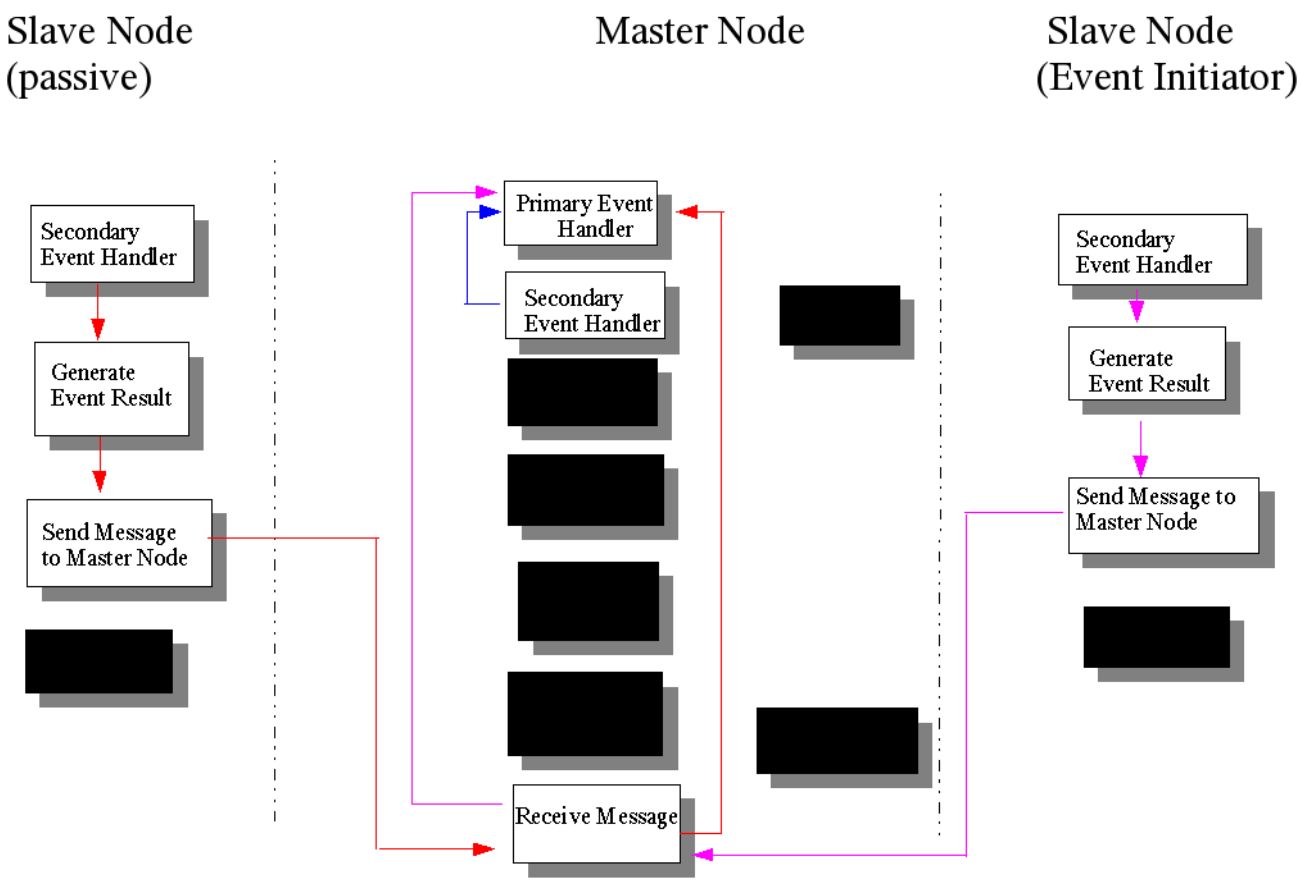
Event Flow – Phase 2



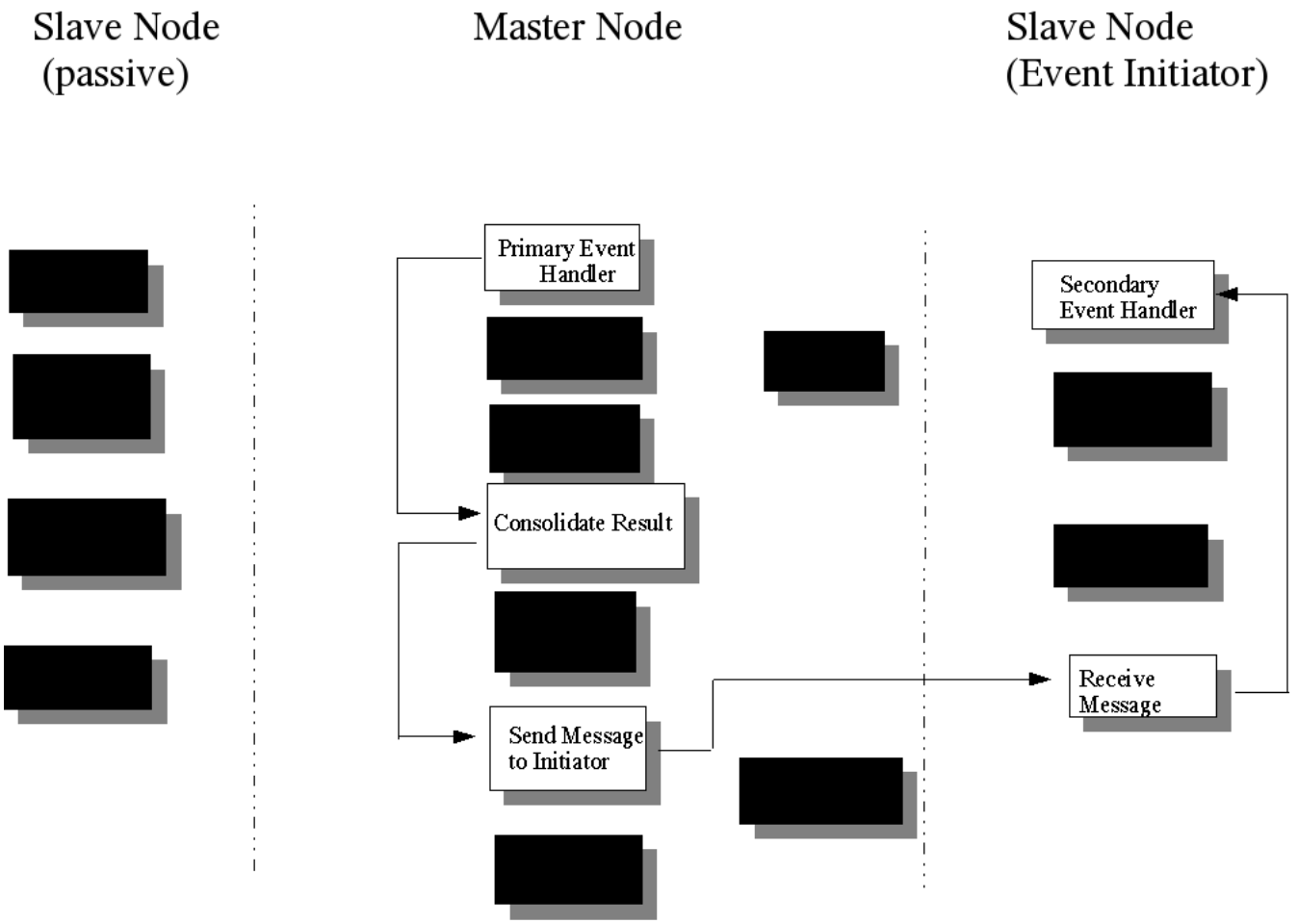
Event Flow – Phase 3



Event Flow – Phase 4



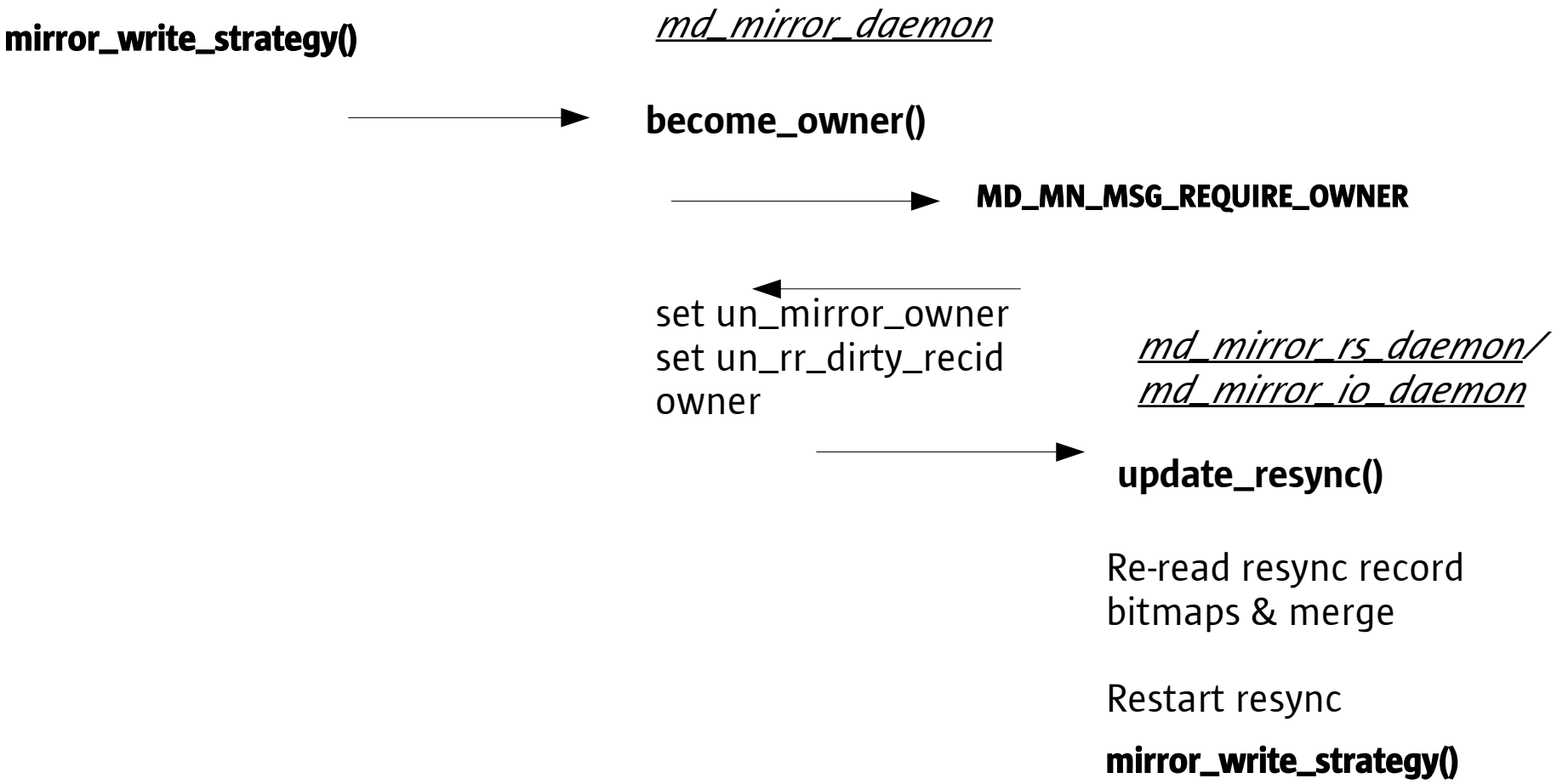
Event Flow – Phase 5



DRL Mirror

Each node in a cluster can write to a shared mirror.
    Every time the writer changes, the associated DRL records need to be updated and transferred to the new owner
Ownership can change on every call to mirror_write_strategy()
metastat displays the current owner of a mirror; 'None' is displayed if the mirror has not been written to or is being treated as an ABR mirror

Ownership change



Data Structures

Daemon structures used for blockable contexts

- mirror_daemon_queue
  - Used to start ownership change request. Populated by mirror_write_strategy(), serviced by become_owner()
- mirror_rs_daemon_queue
  - Used to service resync derived ownership change. Populated by become_owner(), serviced by update_resync() and daemon_io()
- mirror_io_daemon_queue
  - Used to service i/o driven ownership change. Populated by become_owner(), serviced by update_resync() and daemon_io()

Data structures contd.

mm_unit_t structure elements used

- un_owner_state
  - MM_MN_OWNER_SENT set while ownership change in flight
  - MM_MN_BECOME_OWNER set on originating node when mirror_set_owner() runs; MM_MN_OWNER_SENT cleared
  - MM_MN_PREVENT_CHANGE set to prevent the ownership moving from the current owner. Used by the soft-partition creation code
- un_mirror_owner
  - set to node-id of mirror owner. Consistent after successful completion of MD_MN_MSG_REQUIRE_OWNER message
- un_owner_mx
  - controls access to un_owner_state

MD_MN_MSG_REQUIRE_OWNER

- Calls mirror_set_owner() to update the node's in-core ownership field. If the node is not the owner of the mirror and is not the requesting node (i.e. the to-be-assigned owner), update the un_mirror_owner field to the specified node ID.
- If node *is* the requesting new owner, set MM_MN_BECOME_OWNER in un_mirror_owner_state. Do *not* update un_mirror_owner. This is done on successful completion of the message handler.
- If node owns mirror, relinquish ownership, drain i/o, block resync, transfer resync record ownership

Application Based Recovery

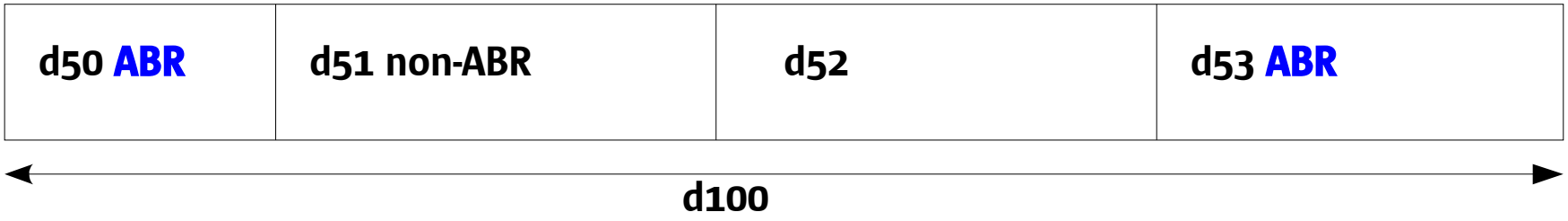
- ABR means that the standard SVM dirty region list is not updated on a mirror write. The application must handle the resync of the data if a node panics during the write.
- Provides higher throughput as the DRL is not written to and then cleared as a result of each write to the metadevice
- ABR capability is stored in a per-unit ui→ui_tstate flag to allow mirror_write_strategy to perform necessary processing

ABR continued...

- As soft partitions can be created on top of a mirror the ABR behavior needs to be propagated through the softpart driver
  - MD_STR_ABR indicates an ABR write request originating from a higher metadevice (e.g. a softpart
- Direct kernel accesses (QFS) can also use the ABR behavior by setting the buf flag B_ABRWRITE on each individual I/O.

ABR and ownership

- ABR mirrors do not have an owner unless a submirror or component resync is in progress
- Top-level soft-partitions can provide ABR capability but the underlying mirror will not be marked as ABR. Each i/o through the soft-part will have MD_STR_ABR set in the strategy flags. This allows for non-intuitive setups:



ABR and ownership contd...

- d100 will maintain resync-records for all non-ABR soft-partitions built on it. write()s to d50, d53 will not incur resync-record updates
- Optimized resyncs will occur for all regions with a resync-record marked (i.e. not d50,d53)
- Component and submirror resyncs will occur as normal
- DMR will recover application data from all sides of the mirror associated with d50, d53

Performance

- Frequently changing owners is a bad thing to do because:
  - heavy i/o load on the current owner will delay the switch away from that owner
  - the time to switch scales linearly with the number of nodes in a cluster as the message has to be passed to each node in sequence
  - it probably means the database isn't set-up correctly

Types Of Resync

- Optimized
- Submirror
- Component

Optimized Resync

- Ensures that mirrors are in sync following
  - System failure
  - metaoffline/metaonline
- Only need to resync the blocks that have been written to since the last time we know they were in sync.
- Mirror is split into a maximum of 1001 contiguous regions and the dirty-region bitmap is stored in mddb
- When writing to a mirror we maintain this dirty-region bitmap, un_dirty_bm in the unit structure. When we are about to add a new dirty region, we commit the resync record to disk.

Optimized Resync – cont

- When a write completes, the count in un_outstanding_writes is decremented.
- Every md_mdelay seconds (default 10), check_resync_regions is called to check if any resync regions are now clean and if so commits the resync record. The resync record has a record of the regions that have been written to in up to the last 10 seconds.
- Before calling optimized_resync(), un_dirty_bm is copied to un_resync_bm.
- check_resync_regions() does not clear any dirty region if the region has not yet been resynced, ie bit set in un_resync_bm.

optimized_resync()

- Sets MD_UN_RESYNC_ACTIVE and MD_UN_WAR in un_status.
- Calculates un_resync_2_do as the number of dirty regions, un_resync_done set 0.
- Loops through all of the dirty regions, calling resync_read_blk_range() for each dirty region.
- Increments un_resync_done for each region resynced.
- Clears bit un_resync_bm for each region resynced.

submirror_resync()

- Resyncs a submirror following a metattach
- Sets MD_UN_RESYNC_ACTIVE and MD_UN_WAR in un_status.
- Splits the submirror into chunks. The size of each chunk is 1/100 of the mirror size if <= 1TB or 1/1000 of the mirror size if > 1TB.
- Sets un_resync_2_do to the number of chunks, un_resync_done set to 0.
- Loops through the mirror calling resync_read_blk_range() for each chunk.
- Increments un_resync_done for each region resynced.
- Continue until all chunks resynced.

Component_resync()

- Called to resync a submirror component following a hotspare allocation or a metareplace.
- Scans all submirror components calling check_comp_4_resync() for each one.

Resync Thread – resync_unit()

- The 3 types of resync are controlled by a resync thread, one thread per mirror.
- When a resync is required due to metasync, metattach, metaonline, metareplace or hotspare allocation command, mirror_resync_unit() is called to create the resync thread, resync_unit().
- resync_unit loops through resyncs in the order:
  - > Optimized
  - > Component
  - > Submirror

Resync Thread – resync_unit()

- It continues to loop until no resync has been performed by any of the 3 resync functions. The variable un_dropped_lock controls this. un_rs_dropped_lock is set 0 at the start of the loop and is set to 1 by the resync functions when a resync is being done.
- At the end, MD_UN_RESYNC_ACTIVE is cleared in un_status and the thread terminates.

Data Structures

- The key data structure for resync is the mirror unit structure.
  - c.un_status – MD_UN_RESYNC_ACTIVE set while resyncs are running and MD_UN_WAR set while an individual resync is running.
  - un_rs_done – number of resync segments that have been resynced.
  - un_rs_2_do – number of segments to be resynced
  - un_rs_dropped_lock – set if a resync has been run during the current loop through the resyncs
  - un_rs_flg – controls commits to resync record

Oban Resync Requirements

- When a mirror changes ownership, the resync should migrate to the new owner.
- When a node fails while performing a resync, it must be possible to restart the resync on another node.
- When resyncing an ABR mirror, the resync must prevent writes on other nodes from overlapping with resync writes.

Starting a Resync

- In metattach, metareplace, metaonline and metasync, we first execute the command on every node, which, for a MN set, does not start the resync thread.
- Once the command has been executed, send a message, RESYNC_STARTING, to start the resync thread on every node.
- Message handler calls mirror_resync_unit() to start the resync thread.
- Only the mirror owner will perform the resync, on non-owner nodes the mirror_thread will block.

Starting a Resync(2)

- mirror_resync_unit() also starts a resync_progress_thread on each node.
- Once the resync thread starts, if the node has been joined to the set, issue GET_SM_STATE message to get the submirror state from the master. This ensures that the state on all nodes are in sync.

Overlapping writes

- For non-Oban sets wait_for_overlaps() is called to ensure that there are no pending writes that overlap this block.
- Before adding an entry, if the current block overlaps any entry on the linked-list, wait_for_overlaps() waits until there are no overlaps and then adds the entry
- Once the write is complete, the entry is removed from the list.

Overlapping writes(2)

- For a non-ABR mirror, the old algorithm works since only 1 node can write
- For an ABR mirror, writes can be executed on all nodes while a resync is in progress on one of the nodes. Writes on the other nodes that overlap the current resync block must be blocked
- For ABR mirrors, the application must handle the overlapping writes on different nodes

Overlapping writes(3)

- Split the resync into 4Mb regions.
- Before resyncing each 4Mb region, send a RESYNC_NEXT message to all other nodes. This is done in resync_read_blk_range().
- On receipt of this message, all nodes, except the owner, setup un_rs_prev_ovrlap with the current 4Mb region and call wait_for_overlaps() with this as the argument, to ensure that there are no writes that overlap this region.

Overlapping writes(4)

- The current resync region must be on the overlaps chain before we execute an ABR write on a non-owner node.
- Can't rely on the call to wait_for_overlaps() in the RESYNC_NEXT handler as it releases the unit lock and a write may be started before resync region is on the chain.
- Add a call to wait_for_overlaps() in mirror_write_strategy() to add the resync region before the write is processed.

Overlapping writes(5)

- An additional flag has been added to wait_for_overlaps() to allow multiple calls for the same parent_structure, MD_OVERLAP_ALLOW_REPEAT.
- Needed because we now may call wait_for_overlaps() several times for the same resync region.
- After wait_for_overlaps(), reacquire the unit lock and then check that we are still the owner. If not we have to remove the resync region from the overlaps chain.

Changing Ownership

- When the resync thread is run on non-owner nodes, set MD_RI_BLOCK_OWNER in un_rs_thread_flags.
- Before the main loop, call resync_kill_pending() which
  - Blocks on un_rs_thread_cv if MD_RI_BLOCK_OWNER
  - Exits 1 if the resync has been finished or aborted
  - Exits 0 if mirror owner
- When a node becomes the owner, MD_RI_BLOCK_OWNER is cleared and the resync thread is signalled to continue

Changing Ownership(2)

- When a node relinquishes ownership, MD_RI_BLOCK_OWNER is set to cause the resync thread to block.
- Ownership can only be changed when the unit lock is free. This lock is held during I/O but any time the lock is released in a resync, resync_kill_pending() must be called as there may have been a change of ownership.
- When resuming after resync_kill_pending() we need to check we are still performing the same resync. It is possible that another node may have completed the resync and started a new one.

Changing Ownership(3)

- In each resync function, resync_read_blk_range() is called to write a number of contiguous blocks.
- In resync_read_blk_range(), resync_kill_pending() is called after each I/O to deal with change of ownership and resync termination.
- resync_read_blk_range() exits 1 if the resync has been cancelled. The calling resync function will break out of the resync in this case.

Changing Ownership – another problem

- When issuing a resync read, mirror_read_strategy() calls drops the unit lock before calling wait_for_overlaps().
- When the lock is dropped, an ownership change may occur.
- After getting the lock, if we have lost ownership, we request ownership before continuing.
- Once ownership has been obtained must check that the block requested is still in the current resync region.

Changing Ownership – another problem(2)

- This may not be the case if another node has completed this 4Mb resync region and progressed to the next while this node was waiting to become the owner.
- If outside of the current resync region just abort the resync read/write.
- The same may occur in mirror_write_strategy() when performing a write-after_read (a resync write) so check if in the current resync_region.

Maintaining Resync Position on all nodes

- RESYNC_NEXT message sent every 4Mb before issuing the I/O.
- This message includes un_rs_type, un_rs_startbl, un_rs_resync_2_do and un_rs_done.
- On receipt of this message, resync state is updated.
- When there is a change of ownership, this state is used to determine where to start from.

Maintaining Resync Position in the metadb

- The resync_progress_thread runs every 5 minutes and, on the master node, it commits the mirror record to the metadb. Following total system failure, this state can be used to restart the resync.

Terminating a Resync

- When a resync has completed, send a RESYNC_PHASE_DONE msg to all nodes
- The handler for this message clears MD_UN_WAR in c.un_status and un_rs_type
- When all resyncs are complete and the thread is about to terminate, send a RESYNC_FINISH message to all nodes.
- The handler terminates the thread, clearing MD_UN_RESYNC_ACTIVE in c.un_status and clears un_rs_thread.

Restarting a resync following system failure

- The resync_progress_thread maintains the current (within the last 5 minutes) state in the metadb.
- This state is snarfed and when a resync is started in the Sun Cluster reconfig cycle, the resync is started from position recorded in the unit structure.
- Before starting this resync, call optimized_resync() to deal with dirty regions.
- Each resync function has to deal with restarting a partially complete resync.

Debugging

- In a debug kernel, the mirror driver is instrumented with a number of debug messages. These are only output if mirror_debug_flag is non-zero.
- Either add “set md_mirror:mirror_debug_flag = 1” to /etc/system.
- Or mdb -kw
  >mirror_debug_flag/W 1

Reconfig cycle basics

- Reconfig cycle consists of steps
- All nodes must finish with one step before going to next step (barrier)
- Nodes booting run start step first
- Returning nodes run return step first
- New reconfig cycle can start after any step

Reconfig cycle SVM specific

- SVM interface is metaclust command
- Must recover from a node panic during any operation on master or slave node
- Must recover from coredump of command or daemon
- Must handle case when node panics and reboots fast enough to boot into cluster during next reconfig cycle

Metaclust

- Almost secret command
- Lives in /usr/lib/lvm
- Runs on all nodes
- `metaclust [-t timeout] [-d level] start localnodeid`
- `metaclust [-t timeout] [-d level] step nodelist...`
- `metaclust [-t timeout] [-d level] return nodelist...`
- Step can be
  - Step1 (maps to ucmmstep2)
  - Step2 (maps to ucmmstep3)
  - Step3 (maps to ucmmstep4)
  - Step4 (maps to ucmmstep5)
- -d {Debug level} 5 is the highest level

Start step

- Called by nodes that are joining cluster
- Metaclust start
  - > Suspends rpc.mdcommd
  - > Issues ioctl to set start flag in kernel

Return Step

- Called by nodes that are already in the cluster and prepares the nodes for reconfig
- Metaclust return
  - Suspends mirror resyncs
  - Drains metaset and metadb commands
  - Drains and suspends rpc.mdcommd

SVM step1

- All nodes in nodelist run step1
- Metaclust step1 nodelist
  - Writes new nodelist to /var/run/nodelist
  - Clears any rpc.metad locks left around
  - All nodes choose master
    - If master node already chosen, choose that node
    - If no master node, choose lowest numbered node that is an owner
    - If no owner nodes, choose lowest numbered node

SVM step2

- All nodes in nodelist run step2
- Metaclust step2 nodelist
  - Master node synchronizes user records to be consistent on all nodes in a diskset (this is a big win over traditional diskset recovery)
  - Master node replays entries in change log so that all nodes have consistent view of diskset
  - Master node tells starting nodes to join the set

SVM step3

- All nodes in nodelist run step3
- Metaclust step3 nodelist
  - Reinit rpc.mdcommd which forces rpc.mdcommd to get latest nodelist from rpc.metad
  - Reset mirror owners for mirrors that are owned by nodes that are no longer in the cluster

SVM step4

- All nodes in nodelist run step4
- Metaclust step4 nodelist
  - Resume rpc.mdcommd
  - Check kernel for start flag (set in start step)
  - Reset ABR state on mirrors and softparts
  - Choose owners for orphaned resyncs
  - Restart resyncs

SVM
Multi-Owner Disksets