

LVM Mirroring

For: Red Hat Cluster Summit
By: Jonathan Brassow

Mirroring support for LVM is currently planned for Red Hat's RHEL4 Update 2 release. Some kernel code is already present in mainline and Red Hat kernels, but currently, there is no way to tolerate failure. The fault tolerance code should be showing up soon. The user-land code is beginning to show up in CVS (see: <http://sources.redhat.com/lvm2>). It manifests itself as new arguments to `lvcreate` ('-m'), which allow mirror creation, and a new command (`lvconvert`), which allows mirror to linear conversion.

A mirror consists of a variable number of replicated volumes and a log. The log is used to track the state of portions of the replicated volumes. The state can be clean (aka in sync), dirty, or not in sync. The clean state implies that portion of the mirror address space is the same. The dirty state is used to identify that a portion is being altered. The not-in-sync state identifies a portion as being not the same. A write to a mirror would consist of writing to the log to mark the destination address as “dirty”, writing to all replicated volumes, and writing to the log to mark the destination address as “clean” once all writes are complete. To prevent excessive logging, the logical address space of the replicated volumes is broken down into “regions”. These regions are powers of 2 typically in the high kilobyte or low megabyte range. So, a second write to the same region of a mirror would not require informing the log, since the first write has already done so. The log is not marked clean until all writes to a region complete. If a failure occurs, regions which are in the dirty state are now considered to be out of sync.

During I/O, reads which occur to an in-sync region are free to choose the mirror device from which they read. This has the ability to improve performance, although the current implementation leaves much to be desired. Reads to a region that is not in sync, along with all writes, are queued and processed. Concurrent to normal I/O, recovery of any out-of-sync regions is taking place. There is a single task which handles dispatching queued I/O and recovery I/O and it happens in the following way:

- 1) Update region states: Notify the log to clear regions that have recently completed all writes or which have recovered successfully. Also, dispatch any writes that were delayed behind a recovering region.
- 2) Do (more) recovery: Query the log for an out-of-sync region and note that the region is recovering. Copy the region from the primary device to the other replicate volumes.
- 3) Do reads: If the mirror is in sync now, do read balancing; otherwise read from the primary mirror.
- 4) Do writes: Separate writes into three groups according to whether the region is in-sync, not-in-sync, or flagged as recovering (as noted in #2). Writes to in-sync regions are written to all replicate volumes. Writes to not-in-sync regions are written only to the primary volume. Writes to recovering regions are placed on a list to be written by #1 in the next pass.

The mirror kernel component is a target to device-mapper, which is composed of two files¹ – `dm-raid1.c` and `dm-log.c`. The first file implements the algorithm previously described. It also provides a mechanism to support different logging implementations. In fact, each mirror instantiation could have a different logging implementation. The dm-

¹ There are plans to pull out the region handling code so that it can be used by future device-mapper targets.

log.c file holds the implementation of two kinds of logging - “core” and “disk”. The core version tracks region state in memory, while the disk version requires a separate device to which it can write the region state persistently. While the core version is fast during use, it lacks fast recovery and start-up. Indeed, upon device activation, the core version must assume that all regions are out-of-sync² – thus slowing I/O while the mirror re-syncs. When the disk version is used and the device is activated, it can read which regions were dirty when it died/was shutdown – considering them to be out-of-sync. This greatly improves recoverability. It should be obvious that when using a persistent log, there is a trade-off between speed and recoverability when choosing the region size. The larger the region, the fewer the writes to the log (disk). However, this also means that larger portions of the disk need to be re-synced in the event of a failure.

The cluster mirror implementation also makes use of the ability to have multiple logging implementations. The cluster log implementation is a fault-tolerant client/server logging system. It requires shared storage (which should be available, given the fact that the mirror volume is meant to be clustered). The mirroring code makes calls to the log, which are passed on to the server. The server then responds to the client, which passes the results back to the caller. This may cause network traffic (which is inevitable), but it reduces disk traffic. The server is smart enough to know that if one machine has marked a region dirty, it need not mark it again for another machine. Other optimizations have been made as well. For example, if all the regions of a volume are in sync, the clients make note of it. Then, when asked by the mirror code if a region is in-sync, they can immediately respond – saving a round-trip communication with the server. The only way regions get out-of-sync is via node failure – something that all the nodes are notified of via the cluster infrastructure (see cman, Red Hat's cluster manager). Another optimization is to cache log clearing requests. This has two effects. Firstly, if a write occurs to a region that has a pending clear request, the clear request is canceled – saving two round-trip communications with the server and possibly additional log device writes. Second, if another machine is notifying the server to mark the region as dirty, the server can simply respond – saving a write to the log disk. This clear request caching is done sparingly because it can lead to increased recovery times.

A disk failure is treated differently for reads and writes. If an I/O fails during a read, it is simply retried on a different device. If a failure occurs during a write, an event is raised and the completion of the write is blocked until the device is suspended. The reason for this has mostly to do with cluster support. If a disconnect happens to a device on one machine and it completes the write, it is possible for other machines to attempt to read the same area from the device that was disconnected from the first machine. This would cause an inconsistent read. If the device is suspended, I/O is queued up on the other machines until the mirror device is resumed without the misbehaving device. The suspension, reconfiguration, and resumption is done in user-space as a response to the event signaled during the failed write. There is a daemon running (dmeventd) that “waits” for events. If one should occur on a device, the proper handling code for a particular target type (mirror in this case) is run. The proper handling code is made available to the daemon via a dynamically loaded shared object library, which is loaded

² There is an option that can be given upon mirror device activation that tells the log to consider the entire volume in-sync – but you wouldn't really want to automate this, would you?

when the device is “registered” to listen for events. So, in the mirror case, the flow might look like the following:

- 1) The call to create/activate a mirrored volume is made
- 2) The device-mapper table is created and loaded
- 3) A “register for events” call is made
- 4) If there is no dmeventd daemon running, it is started
- 5) The daemon recognizes that a mirror device is being registered for monitoring and loads the appropriate DSO
- 6) If a failure occurs, the daemon sees the event and it calls the processing code in the DSO
- 7) The DSO reconfigures the mirror device to exclude the failed device
- 8) Operations to the device proceed