Device-Mapper Remote Replication Target

Heinz Mauelshagen
Red Hat GmbH
Stuttgart, Germany
heinzm@redhat.com

1 Abstract

The Device-Mapper, as a general purpose mapping subsystem, is part of the Linux kernel since back in the 2.5 development cycle being shared by applications such as LVM2, dmraid, kpartx, multipathd etc.

Details of any particular mapping (e.g. mirrored) are being abstracted via pluggable mapping targets (i.e. kernel modules), hence enabling addition of new mappings to a running system.

A new target, the author has been working on since late 2007, is the Replicator Target to cope with disaster recovery requirements in Enterprises. It allows for remote data replication of arbitrary groups of block devices as a single, write-ordered entity to multiple remote sites, supports synchronous and asynchronous replication and fallbehind thresholds to automatically switch from asynchronous to synchronous mode and back, beside other features.

2 Background

2.1 Device-Mapper Architecture

In order to be able to understand, how the Replicator Target fits into Device-Mapper (aka "DM"), a brief overview is appropriate.

The generic block remapping subsystem „Device-Mapper" handles virtual block devices called mapped devices (aka „MD"), which get remapped to other underlying block devices in a stacked way. The logical address space of MDs is defined via ASCII formated mapping tables. Mapping table lines define segments in the address space of the MD in units of sectors (512 bytes). An MD can have an arbitrary amount of address segments.

DM falls appart into the following kernel and user space components:

– kernel
  - o core
  - o ioctl interface
  - o io
  - o kcopyd
  - o mapping targets
  - o dirty log
  - o region hash

– user space
  - o library (libdevmapper)
  - o dmsetup tool

The kernel core covers all functionality to create and remove mapped devices, to load and unload their mapping tables and to register mapping targets. It has no concept how to remap an io which is the task of mapping targets. Mapping targets are plugins (loadable at runtime) for a growing list of mappings such as linear, striped, mirror, multipath, crypt, snapshot, zero, error, delay, raid45 and, of course, replicator. The target interface allows passing an io (i.e. a bio in Linux 2.6) to the target, which can do whatever it feels appropriate with it.

That can be as simple as changing bio content (i.e. bi_bdev and bi_sector) and pass it to the lower layers, hence remapping the bio to another block device or it can be more complex as queuing it to an internal list to merge it into an internal log for io, hence causing io to be carried out to multiple block devices such as in case of dm-replicator.

The device mapper core will call into the target for other purposes as the mapping of an io such as mapping table segment create/destroy, to retrieve the actual table or the MD status, pass messages to the target in order to influence its state or suspend/resume io activity on a target as well.

The ioctl interface exposes all DM kernel functions available to the user space library, which in turn is used by applications like dmsetup, dmraid, kpartx, multipath or lvm2.
It is recommended that applications utilizing DM use the DM library (libdevmapper), because it hides different versions of the ioctl interface being exposed to access the kernel DM features from them.

dmsetup, which of course accesses the ioctl interface through libdevmapper, is a simple 'volume manager' tool, with the main functionality to of mapped device create/destroy and load/unload mapping tables beside others.

Mapping tables define the logical address space of any given mapped device.

Their syntax is: 'start length target target_paramters'.

Start+length define a segment in the mapped devices address space. All size parameters in DM (such as start) are in units of sectors of 512 bytes.
'target' is the keyword, which selects one of the mapping targets (e.g. 'linear').

This very powerful syntax allows an arbitrary amount of segments and each of them can use a different target. For-instance.:

```
0 1024 linear /dev/hda 0
1024 4096 striped 3 128 /dev/hda 1024 /dev/hdb 0 /dev/hdc 0
```

maps the first 1024 sector to offset 0 on /dev/hda using the 'linear' mapping target and stripes the next 3072 sectors to 3 devices with stride size 128 sectors.

An io module (dm-io) which contains low-level routines for synchronous and asynchronous io operations is reused by dm-kcopyd and the targets in order to avoid code duplication. dm-kcopyd is the DM kernel copy daemon, which allows copy requests from one source disk to as many as 32 destination disks. The mirror target for-instance uses dm-kcopyd to resynchronize mirror sets in case they got created or partially ran out of sync because e.g. of a system crash.

A dirty log with different types (transient, resident and cluster) is used to keep track of the state of regions (size configurable) within the MD's address space in order to be able to resynchronize dirty regions after e.g. a system crash, when any dirty regions in the log get promoted to NOSYNC, hence subject to region resynchronization.

As mentioned, utilizing DM means calling the libdevmapper functions to create an MD, load its mapping table and access the device node created for the MD (default namespace: /dev/mapper/). In test environments this can be done using your favorite text editor to create a mapping table and hand that to dmsetup together with a name for the MD. In case you use lvm2, you normally ain't see DM at all, because you will be accessing the well known LVM UI or even a GUI deployed with the Linux distribution you're using (e.g. system-config-lvm for Fedora Core or RHEL).

2.2 Data Replication Basics

In order to provide a base to understand the aim of the Replicator Target, this parapgraph provides some technical and business background.
I’m referring to newer techniques of continuous data protection (CDP) in the context of this paper.

As opposed to traditional backup concepts, which take copies of the production data on a regular time basis (e.g. daily differential, weekly, differential/full backups), hence opening large time windows with non-protected data, CDP continuously copies primary data on (remote) secondary storage separate from the primary storage holding the work copy. IOW: data is being duplicated when it is being written rather than infrequently.

Because of this, storage replication is the mechanism of choice to allow for failover of services in disaster recovery situations, providing up-to-date redundant copies of production data, hence allowing for minimal data loss (see below).

Solutions generally fall into 2 major classes: active-active (AA), where read/write access to all copies of the data is allowed and active-passive (AP), where such access is only possible to the primary copy with copying of data happening either synchronously or asynchronously to one or more secondary copies. AA solutions involve overhead for synchronization of the accesses to the copies of the data, which can happen in any location, and are hence typically limited to 2-3 copies, whereas AP solutions don't require such synchronization overhead, because at any given point in time, there is only one primary copy in one location with write access, which may switch to a different location on request. AA requires computing nodes at each site for mandatory cluster software to run on; AP don't do that, they only require networked storage.

Both types support physically distant locations for primary and secondary storage sites, hence protecting data in disaster scenarios, where a complete storage site is lost and valid production data is still available on one or more other sites. Storage replication can happen on various levels (e.g. application, filesystem or block device). This paper addresses block device replication.

In disaster planning, recovery point objectives (RPOs) and recovery time objectives (RTOs) need to be defined, based on the requirements of the individual business.

An RPO defines how much data copies may fall behind the primary data they are being copied from. It is essentially a technical feasibility/cost balance on a scale defined by the best case, very expensive and eventually not technically doable objective to have identical copies at any point in time and affordable costs which result in a data delta between the copies.

The best case requires low-latency, high-bandwidth, high-available storage interconnects, which are either not affordable or not technically available long-distance, because of the high costs of long-distance links and the high round-trip times on those (e.g. 10 kilometers already cause 67 µs because of the speed of light vs. 10-20 µs to local cache) delaying writes too much, hence preventing acceptable application write performance.

The doable case will typically involve asynchronous replication (see also: asynchronous mirroring; remark below), which is able to cope with bandwidth and latency restrictions by buffering the written application data in order in a persistent log and copying it to any devices from there in order (either local or remote), hence allowing for write ordering fidelity. With such log it is possible to report end IO in the asynchronous case to the application, when the write has reached the log stable storage together with metadata describing it and to handle temporary outages of device links. The drawback of this asynchronous mode is, that the replica (i.e. the device being replicated to) will fall behind the replicated data. Even so, ensuring write ordering allows an application to recover, because it will always find a data set on the replica enabling e.g. transaction rollbacks.

On the other hand synchronous replication may be required in restrictive business cases, where it is mandatory to always have the data on the replica being an exact duplicate of the replicated device (see also: synchronous mirroring), which is the 'best case' point above (costly or not technically feasible). Such cause application stalls, which is sometimes forgotten about, unless synchronous replicas are being dropped or information is being kept on intended writes to them, when the transport fails. In the later case, a synchronous replica can be recovered after the link returns.

A major difference between replication and mirroring is, that replication can cope with (temporarily) unavailable storage transports, whereas a timeout on the transport will cause a mirror to be dropped as faulty. Hence, replication typically leads to an higher level of availability vs. mirroring.

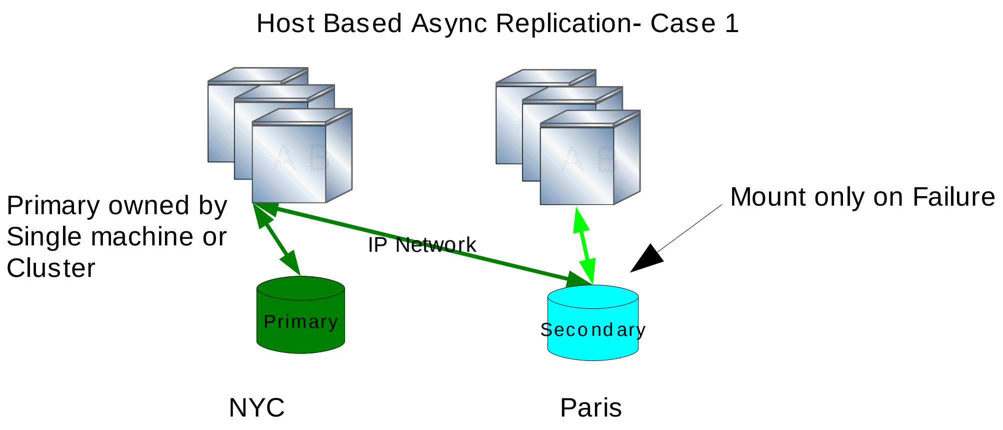
An RTO is the the time until all services have to be online again at a different (or the same) site after a disaster put them down to allow for business processes to continue. I.e. the intended time delay after any application failed because of a disastrous case until such application is fully operational and accessible to the user to allow for continuing his business. This has to cope with regaining access to all necessary hardware and software resources and, for the most important point, actual production data. RPOs and RTOs are established during the business impact analysis, hence being driven by the evaluated impact.

2.3 Remote Replication Scenarios

A kernel level mechanism to support active-passive type replication. It aims to deal with the following configuration use cases, optionally in combination with the given mirror and snapshot targets:

2.3.1 Base (A)synchronous Replication

In this case a single machine, or local cluster owns the primary device/volume which is then (a)synchronously replicated to 1 or more geographies. The replication needs to withstand short term

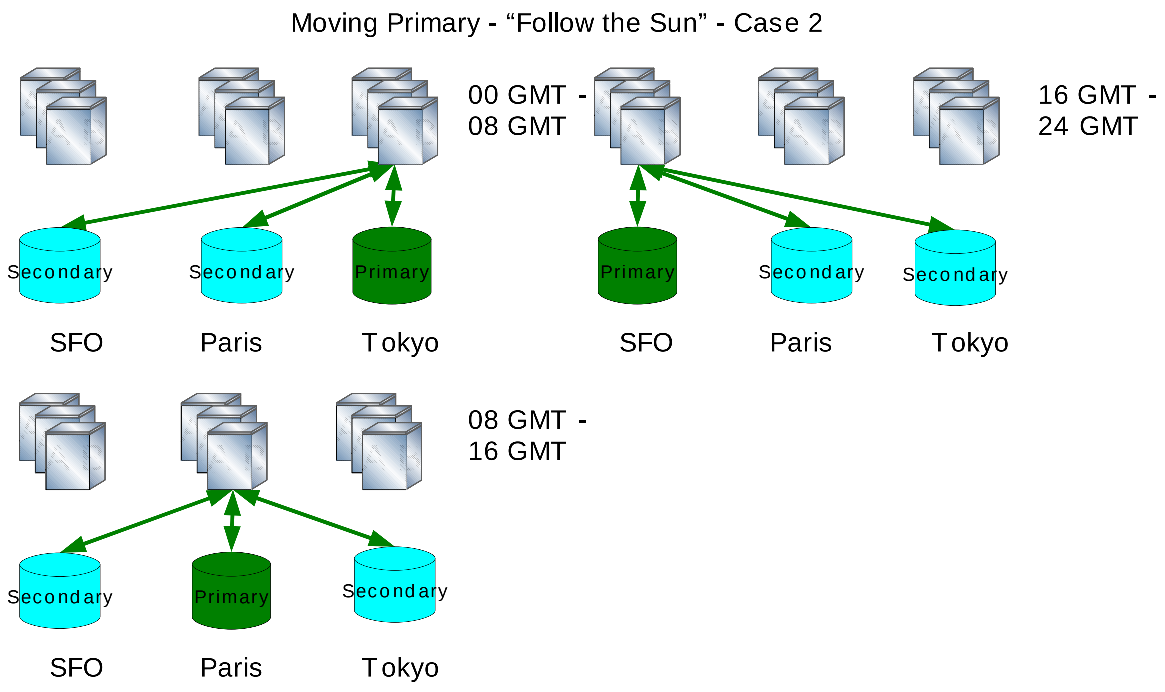

Host Based Async Replication- Case 1
Primary owned by Single machine or Cluster
IP Network
Mount only on Failure
Primary
Secondary
NYC
Paris

outages and recover the replicated devices/volumes without the need to copy them completely. This involves the ability to fall back to keeping track of non-synchronized regions in remote locations based on fall-behind thresholds once access to distant devices fails for longer periods of time and using that region information to resynchronize when access returns. While such resynchronization is happening, no write ordering is ensured until it finishes.

It is mandatory to be able to ensure write ordering fidelity for a group of devices to support applications (e.g. DBMSs), which typically utilize a list of devices, to recover on secondary sites.

2.3.2 Follow The Sun

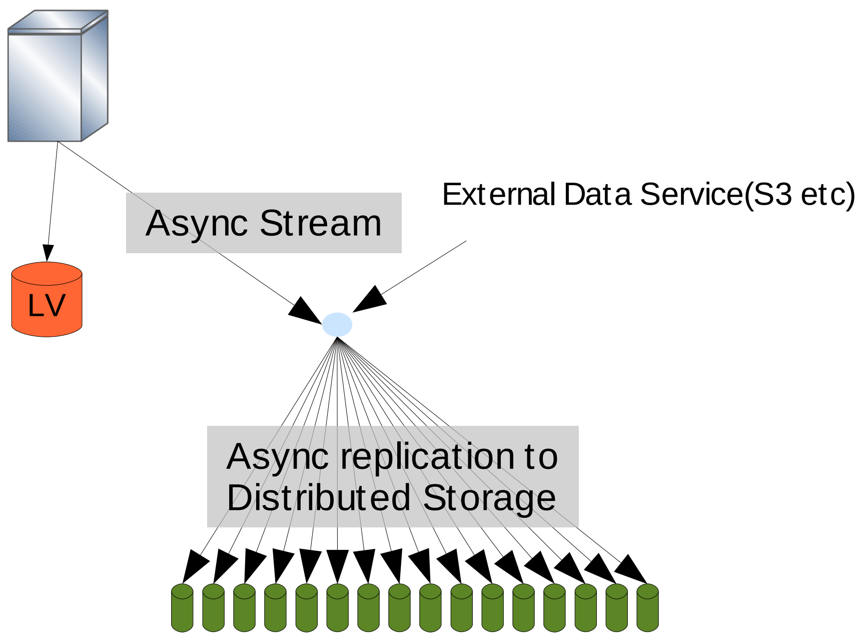
n this model of replication the primary device/volume may be moved on a regular schedule. Only one system (or cluster) owns the primary devices/volume at any point in time, but the primary can be moved where it is



active in a given geography.

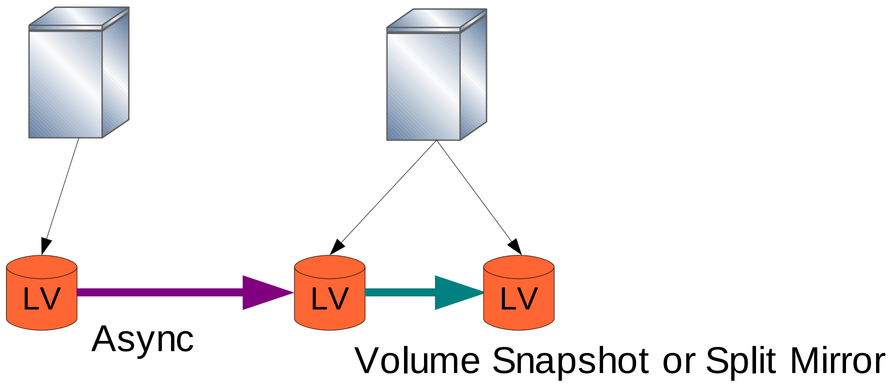
2.3.3 Asynchronous Replication to Data Service

Besides writing directly to a physical or logical device, future extensions will support writing to a "foreign


Async Stream
External Data Service(S3 etc)
LV
Async replication to Distributed Storage

interface" like S3. This requires a plug-in model where new adopters can be constructed to write to one of these alternate services. In this example the remote data service provides a RAID-like set of copies.

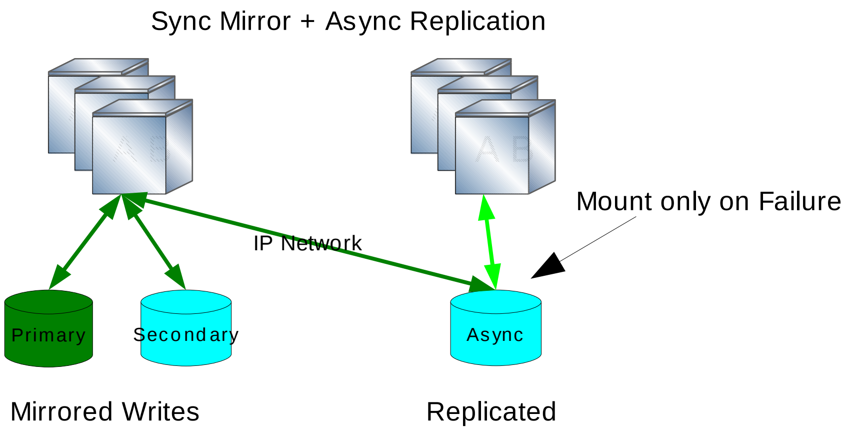
2.3.4 Split Mirror Processing



In this case a mirror can be created and split off as a snapshot for processing.

2.3.5 Synchronous + Asynchronous

This model combines both local, synchronous mirroring with asynchronous replication. This provides completely accurate, local volume copies, with copies at a geography not subject to local catastrophes.



2.4 Device-Mapper Remote Replication Target

2.4.1 Target Notions

Transports (e.g. Over “IP Network” above) are called “site links” or SLINKs which have attributes (e.g. synchronous or asynchronous).

The devices at a primary site are named Local Device or LDs and hang off SLINK 0.

Devices at a remote/secondary site are named RDs amd hang off SLINKS > 0.

2.4.2 Target Architecture

The targets core implements the constructor, destructor, mapping, etc, functions needed by the DM core. It knows nothing about how to handle a replication log or copying data across SLINKs to LDs or RDs. It only copes with grouping of devices on SLINKs and associating them to a replication log.

The pluggable “default” replication type implements a ring buffer type log, the logic to (de)allocate space in the ring buffer for entries and to keep state with each entry plus the ability to (a)synchrnously copy entries data across SLINKS. An entry contains the written data hanging off a bio, its correlation to device numbers and SLINKS it needs copying across. The replication log doesn’t know anything about SLINKs (but their numbers), device paths or other transport endpoints, just device numbers to address devices. It keeps track of fallbehind thresholds if set in case of asynchronous replication and if hit, switches to synchronous mode, hence reporting end IO, when the data has reached the LD/RD. When the amount of data not being copied to an SLINK falls below the threshold, asynchronous mode is reestablished.

The pluggable “local” site link type implements device node based access to devices. I.e. it supports all devices being accessible via local device nodes such as those being accessed via iSCSI or FC transports. It is mapping between SLINK and device numbers and transports. In the “local” case, it maps between those and device nodes and it delivers SLINK status (e.g. “down”) and checks, if SLINKs are accessible.

In case a better performing replication log is invented, another replication log handler can implement it and be added to the target because of the plugin design. Likewise, a non-device node type transport (i.e. S3) can be implemented in another SLINK handler.

2.4.3 Mapping Table Exapmles

Any application using this target for remote replication has to set up a mapping table, create a mapped device and load the table via libdevmapper. Support in LVM2 is planned for this year.

```
“0 311296 replicator \
default 4 /dev/vg00/replog 0  auto 2097152 \
local 2 0 asynchronous \
2 0 /dev/vg00/lvol0 \
– 0 \
local 3 1 asynchronous ios=100 \
2 0 /dev/sdz \
core 2 2048 nosync “
```

creates one LD and one RD.

Line 1 defines a segment start “0” and length “311296” in sectors and selects the “replicator” target.

Line 2 selects the “default” replication log handler with “4” following parameters and either creates or opens (“auto”) the log backing store device on “/dev/vg00/replog”. In case it creates the log, it’ll try sizing it to “2097152” sectors.

Line 3 selects the “local” SLINK handler with “2” following parameters, SLINK “0” (i.e. the one to access LDs) and selects “asynchronous” replication.

Line 4 with “2” device parameters defines an LD with device number “0” on “/dev/vg00/lvol0”.

Line 5 defines no dirty log for this LD because LDs are never allowed to fall back to non-write-ordered mode.

Line 6 selects the “local” SLINK handler with “3” following parameters, SLINK “1” (i.e. one to access RDs) and selects “asynchronous” replication with a fallbehind threshold of 100 ios.

Line 7 with “2” device parameters defines RD “/dev/sdz” as device number “0”, hence pairing it with the previously defined LD number 0.

Line 8 defines a “core” dirty log with “2” parameters being “2048” the region siue and “nosync” prohibiting any regions to be resynchronized.

When adding another 3 lines to the above mapping table, the LD will be replicated to 2 sites:

```
“local 3 2 asynchronous ios=100 \
2 0 /dev/sdw \
core 2 2048 nosync”
```

Data is allowed to fall behind 100 ios (“ios=100”) on SLINK 2 in this extended example.

If 2 devices shall be replicated as a write ordered group to one secondary site:

```
“0 311296 replicator \
default 4 /dev/vg00/replog 0  auto 2097152 \
local 2 0 asynchronous \
2 0 /dev/vg00/lvol0 \
– 0 \
local 2 0 asynchronous \
2 1 /dev/vg00/lvol1 \
– 0 \
local 3 1 asynchronous ios=100 \
2 0 /dev/sdc \
core 2 2048 nosync
local 3 1 asynchronous ios=100 \
2 1 /dev/sdd \
core 2 2048 nosync “
```

Replacing the “core” dirty log by a “disk” one allows for partial resynchronization of devices.

3 Conclusion

The Remote Replication Target offers the kernel subsystem extension to Device-Mapper to support a variety of active-passive replication applications allowing to group devices while ensuring write ordering fidelity for the whole group and to replicate the group to multiple secondary sites.

We’re designing and implementing an LVM2 enhancement to take advantage if the targets features.

References

[1] `http://sources.redhat.com/dm`

[2] `http://sources.redhat.com/lvm2`

[3] `http://people.redhat.com/heinzm/sw`