

DRBD[®] 9 & Device-Mapper

Linux[®] Block Level Storage Replication¹

Lars Ellenberg
LINBIT
Vienna, Austria
<http://www.linbit.com>
lars@linbit.com

1 Abstract

High-Availability clusters often use central shared storage as a data back end. With DRBD we provide a technology for Linux to replicate storage, rather than share it. While in my – biased – opinion DRBD is the best open source solution available for this purpose, we can improve it in some aspects. To make future enhancements easier, we need to transform DRBD into a more modular architecture. Not coincidentally we are going to leverage the device-mapper framework for this as much as possible. The new dm-replicator [1] target by Heinz Mauelshagen provides the best starting point.

I present design ideas to combine multiple replication targets, introducing a concept of "data generation tags" to uniquely and conveniently identify any data generation of any data set, and outline upcoming new features, all of which can be combined: • replication to multiple nodes • optimized for long-distance small bandwidth • daisy chaining replication links • multi-master replication.

As a side effect, the necessary infrastructure makes it possible to create snapshots after the fact, or to do point-in-time recovery of arbitrary block device data.

2 Background

2.1 Current Limitations of DRBD

While DRBD 8.2 does a good job at what it does, the monolithic design of DRBD does not easily allow for changes. We don't have replication groups, so we cannot keep possible inter-device write-ordering dependencies consistent when replicating multiple devices. We cannot easily do more than two nodes. We cannot easily do long distance high-latency low-bandwidth replication. During (bitmap-based) resynchronization, the sync-target is inconsistent.

We can stack DRBD, use event-handlers to snapshot before resync is started, use the DRBD-proxy, etc. But the real solution for these limitations is a modular rewrite.

The LVM/device-mapper people are working on a replicator target, and the necessary supporting sub-targets and infrastructure. The current focus of this dm-replicator project appears to be long-distance asynchronous replication between LUNs imported from high-end SANs, mainly for disaster recovery purposes. But the infrastructure is close to what we need for DRBD, having our focus on synchronous or near-synchronous replication suitable for HA-clustering, and for no-SAN, shared nothing, configurations as well.

¹LINUX[®] is the registered trademark of Linus Torvalds in the U.S. and other countries. DRBD[®] and LINBIT[®] are registered trademarks of LINBIT Information Technologies GmbH, Vienna, Austria, in the U.S. and other countries. Other names may be trademarks or registered trademarks of their respective owners.

2.2 What is all the fuss about, replication is straight forward and easy?

Data replication is easy. You just track changes to your data set, and ship a batch of changes from one consistency point to an other, then apply that batch at the replication target. Done.

Most computer science students should be able to implement that long before graduation.

Things are more complex than that. The interesting part is:

- How to deal with IO errors, transfer errors, replication link outages?
- How good exactly is your asynchronous off-site replica, once you need it because your primary site is irrecoverably damaged?
- How can you verify the replica, how can you check that the replicas are in fact equivalent?
- If you have multiple replicas, how can you tell which one has the "best", consistent and most up-to-date version?
- A very nasty problem in data replication is data divergence. How do you prevent data divergence, detect it, recover from it?

Also, you want to be as efficient as possible while not trading performance for correctness.

2.3 Some Terminology

data set the data stored within a volume

clone one incarnation/copy/replica of a data set

secondary, slave, inactive a clone that denies access to its data, but contacts all other nodes of the replication group, listening to incoming replication streams.

primary, master, active a clone that may actively access and modify the data, journaling and replicating the changes to all reachable other nodes of this replication group.

data set generation tag/id a tag to unambiguously identify the "version" of a data set

replicated volume a volume, which replicates changes to its data set to other clones of itself on other nodes

write completion the acknowledgment of any given subsystem that a certain modification request has been noticed. The modification may have only reached volatile caches yet, though, so it may not survive a crash/power outage/reboot.

write completion (stable) the acknowledgment of any given subsystem that a certain modification request has been committed to stable storage, where it will survive power outages/reboots. Example: submit a Barrier Request, and wait for its completion.

write confirmation when a replicated volume confirms a write to its users (upper layers)

From the perspective of the upper layers that is a "write completion".

For the replicated volume, this is the confirmation that the modification request submitted by the user has indeed been committed to "enough" stable storage, i.e. the replicated volume has collected enough "completion" events from its subsystems, whether that be local disks or remote disks via replication links; where "enough" needs to be dynamically (re)configurable.

lock-step replication (synchronous) when write-confirmation to upper layers is done only if several clones have completed the modification request, those clones are said to be in lock-step mode.

online replication, (asynchronous, near-synchronous, best-effort) continuously ship modification to other nodes, keeping the other clones as up-to-date as possible (though not in lock-step due to latency and bandwidth constraints).

offline/delayed replication (asynchronous) collect modifications, and ship and apply remotely at convenient times. The other clones may lag behind considerably - which may be seen as a feature as well.

resynchronization bring one clone (sync target) forward to match the generation of some other, better clone (sync source). During this process the sync target may be inconsistent.

clean if some data set (generation) is consistently and cleanly usable as is, without any need for recovery actions, it is "clean". Example: recently fsck'ed unmounted or read-only mounted file system.

consistent if some data set (generation) is recoverable to a "clean" state, it is in a "consistent" state.

Example: without any hardware errors or software bugs, the on-disk representation of journaling file systems or data base table spaces with write-intent logging are supposed to be "consistent" at any point in time.

inconsistent if a data set (generation) is not (easily) recoverable to a clean state, it is inconsistent.

When modifying a clone violating write ordering constraints, that clone will most certainly become inconsistent.

Example:

- A clean file system on sda, "dd if=/dev/sda of=/dev/sdb", then crash half way through. Once you are online again, the data on sdb may look like a file system at first glance, but when doing a full traversal, eventually you will hit bad references, and invalid data content.
- You have multiple volumes, with inter-volume dependencies, say one holds the application data, and the other the application logs necessary for crash recovery. You replicate both of them independently. Replication links drop out at different times. Now, even though the volumes may be consistent within themselves, the application data would be inconsistent, since data and logs reference different points in time.
- When you crash half way through a dirty-bitmap based replication/resynchronization, the target will be inconsistent; it will only reach a consistency point again when the resync resumes and eventually completes.

consistency point a point in time where the on-disk data-set is in fact consistent.

I assume that, when applying the incoming request stream from the upper layers exactly, without reordering anything, the on-disk image is always consistent, regardless of the position in the replication stream.

To use this incoming request stream as replication stream, and strictly keep its write ordering, may be inefficient for long distance replication.

However, because journaling file systems have write-after-write dependencies, when changing the strict write ordering imposed by the upper layers, it is important to always keep an on-disk state where crash-recovery to a consistency point and resuming replication is possible.

replication stream a data stream that contains information to move a clone to a more recent generation

reorder domain a collection of writes that may be reordered, as they have no ordering dependencies. Any write submitted while a previous write has not been completed yet obviously has no ordering dependency on the in-flight request.

Write requests submitted between two completion events form an implicit, minimal reorder domain.

If the application or file system provides explicit dependency information by using barrier requests, all write requests between explicit barriers may be treated as explicit, maximum reorder domain.

atomic batch a batch of modifications that is to be either completely applied, or not at all. If you crash half-way through, your data set would be inconsistent, so you need to keep a write-intent log of atomic batches, to be able to recover from there.

2.4 Why Replicate

A centralized SAN can still break, or be scrambled by uncoordinated simultaneous writes from multiple nodes. Latency and bandwidth may be better for directly attached storage compared with the SAN overhead.

On the other hand, replicated copies may diverge, if modified independently, and replication bears some overhead in itself.

Replication serves two main purposes.

First, for High Availability clusters, in case the primary goes down, we want to just switch to an other copy and continue at exact the latest transaction. We need to replicate synchronously, in "lockstep". No write may be confirmed as written until it has been completed on at least n copies; chose the n most suitable for your level of paranoia and the importance of that data. This is typically done within one site, or between two sites in geographical proximity.

Second, to guard against disasters (complete site loss; think of earthquakes, floods, or other, man-made scourges), we are happy to have an as-recent-as-possible copy, even if the most recent few transactions are missing. We choose long-distance, asynchronous best effort replication to a disaster recovery site, which we hope to be geographically outside of the affected area. Latency and replication link bandwidth costs forbid synchronous replication. Furthermore, asynchronous replication levels write bursts and may allow us to even compress or otherwise trim down the amount of data to be replicated.

We may combine both approaches, of course. If the remote (the near, HA, lockstep; or the far, DR, asynchronous) site is unreachable, or an asynchronous peer lags too far behind, we have to make a policy decision. On the primary site, do we continue anyways? Do we throttle, or stall, until the remote site becomes available again, or catches up sufficiently?

2.5 Why Replicate on the Block Level

The availability of every service depends on the availability of the data it serves. For Business Continuity, it is obvious that you have to do backups of that data, in case it gets destroyed somehow. For High Availability, backups are not enough. You want a live copy of your data you can switch to if the master copy becomes unavailable. This is what RAID 1 does for you within the local storage subsystem. Replication can do this to some remote storage subsystem. DRBD (Distributed Replicated Block Device) is basically a RAID 1 between separate computers.

As there are different RAID levels, there are different variations of replication, each with its own specific advantages and drawbacks. RAID levels differ in performance, in how well they can cope with (multiple) disk failures, how fast they can recover, and in cost.

Remote replication solutions differ in performance (or relative performance degradation), resilience, ease of recovery, amount of data loss on primary crash, bandwidth usage and other things.

For replication, we need to duplicate data, which may occur at the application layer, the file system layer, the block layer, or even the storage hardware layer. The most generic and flexible approach is to hook into the generic block layer, as it is agnostic to the type of data – which means it works with any file system and any application – but still in a position to use the full power and flexibility of the operating system.

2.6 Challenges

For reliable replication, we need to be able to deal with component failures such as disk, link or node failures. We must not lose transactions (unless allowed by policy), and need to avoid, or at least detect reliably, data divergence, data corruption, or "time warps" into outdated versions of the data, if we have more recent data available somewhere. We need to correctly handle "healing" of components (hardware replacements, link reestablished, node rebooted).

And we want to do all that efficiently, trading just as little performance as necessary. Being cost effective won't hurt either.

As mentioned, current DRBD (8.2.x as of this writing) replicates each device independently. It can only replicate to one peer per device. And during resynchronization (e.g. after replication link outage), the sync target cannot be kept consistent.

3 Direction of development

When a replication target is not reachable for some time, we need to track the changes, so we can (re-)transmit only the changed blocks when the replication link is reestablished. By combining an "activity log" and an "quick-sync bitmap", both persistent on-disk, DRBD already provides necessary tools. (This concept is called "dirty-log" in device-mapper). During such a bitmap based resync, the sync target is inconsistent, as write ordering is violated: the changed blocks are simply sent in one huge sequential sweep over the device. To be able to keep the write ordering even during resync, we need to not only track which blocks changed, but we need to keep a full log, a journal, of these changes.

3.1 Tracking Changes

To be able to replicate an incoming data stream, you basically only need to "tee" it, sending the same data stream to several targets. If you block the incoming data stream whenever you lose any of your replication/site links, that would work – but would be not usable in a typical two-node HA failover cluster.

To be able to update one site to the current data set generation after it had been unreachable for a while, you have several options.

3.1.1 Recording changed Block Numbers

bitmap When one site becomes unreachable, create (on stable storage) a tracking bitmap, tagged with the last completed (by the now unreachable site) generation id, where each set bit represents a certain amount of blocks at the corresponding on-disk position modified since that site became unreachable.

If you lose your replication link to several sites in close succession, and their last known to be completed generation IDs are sufficiently close, you can use the same bitmap for all of them without losing efficiency.

extent map Instead of a bitmap, we can store "offset;length" tuples of dirty areas.

Both methods basically only record *which areas* changed, not how, or in which order. Based on this information, any resynchronization may be efficient, but during it the sync-target will be inconsistent.

3.1.2 Recording Data Changes

full data journaling, "plain-text" REDO log Instead of recording only the position of changes, and ignoring the order, it is possible to do the other extreme, namely record all incoming writes *in order* to an on-disk ring-buffer journal, before applying them to the real storage area.

This is inefficient for frequent overwrites with similar data, and for asynchronous high latency low bandwidth replication links in general.

Once the ring buffer runs full, we can fall back to block-number based tracking of changes.

XOR-based, compressed UNDO log Before shipping the replication stream, we can do some optimizations on it to reduce necessary transfer bandwidth. Since we have the new, to-be-written, data in the log, and the previous, to-be-replaced, data still in the real storage, we can do a binary diff: an XOR operation between previous and new data, and pipe that through zlib. It may be efficiently combined with software RAID 5 or 6, where this XOR needs to be calculated anyways.

This has several advantages.

For typical real-world read-copy-write cycles of small overwrites, the resulting XOR-diff is expected to contain long runs of zero bits, thus compressing very well.

And the XOR-diff can be applied in both directions. Keeping this compressed, XOR-based undo log, makes it possible to create "snapshots" of the data set a long time after it already has been modified, by undoing these changes into a writable snapshot of the real storage.

Of course that same undo operation can be applied to an accidentally diverged data set, so we could consistently resynchronize even those.

Keeping snapshots or frozen clones, plus this XOR-diff stream, essentially provides a basis for point-in-time recovery of generic block device data, which may be interesting especially in virtualisation environments.

atomic batches (scrubbing/coalescing overwrites within batches) Both (the full plain-text log, and the compressed XOR-log) can be combined with write-order dependency analysis and overwrite detection, forming atomic batches by omitting overwrites within a configurable window based on wall clock time, number of write requests or number of modified sectors.

activity log All journaling methods need to keep track of where the most recent version of any given data block resides, in case a read-request is issued before the corresponding data is finally committed to the real storage area.

On crash recovery, the mapping relations can be reconstructed by walking the log. But it would be more efficient if we have a lazily updated mapping table on stable storage as well, especially if we want to use the log as source of snapshots.

The bitmap/extent map based approach would need to transactionally update on-disk data before actually writing to the real-storage.

This can be optimized by using a so called "activity log" we know from DRBD, which transactionally keeps track of in-use, "hot" areas of your storage, and makes it possible to update on disk bitmaps/extent maps/other meta data only at convenient times, thus considerably reducing the amount and latency overhead of necessary meta data housekeeping IO transactions.

On crash recovery, a block number based tracking mechanism marks all used-to-be "hot" extents as dirty. The log based tracking mechanisms use their last, "lazily" stored checkpoint to only walk the part of the log created later.

Once we have such an on-disk, persistent, journaling log of incoming write requests, we can easily replicate to multiple peers from there, allowing for varying degrees of lag-behind. Resync then becomes just a special case of catching up with asynchronous replication, replaying the log.

Global write ordering (not just within one device, but across multiple logically connected devices, such as a database write-ahead log and the corresponding table spaces) becomes trivial as well: just journal several devices to the same log, and make sure the log gets replayed to all replication targets in the same order.

3.2 Dm-Replicator

We need to journal all writes to some persistent log, we need to communicate and replay this log on the replication targets in order. We need some policy decisions in case the log fills up. We still need the possibility to fall back to bitmap based change tracking when some peer is unavailable for too long. These are the modular building blocks. But we don't want to reinvent the wheel.

Enter dm-replicator, which is a device-mapper replication framework by Heinz Mauelshagen and others, who will give you much more detail on what it is and how it works, and correct me where I'm vague or wrong.

dm-replicator is a text-book example on how to write abstract object oriented code in C in the Linux kernel. It provides the modularization and abstraction for this concept of replication, namely a top level coordinating dm-replicator target, and a registry to provide for various implementations of journal (dm-repl-log), bitmap fall-back (dm-dirty-log), and communication and log-replaying ("site links", dm-repl-slink). There are default classes implemented. The default dm-repl-slink implementation expects the remote site to be visible already as a local block device, for example imported via iSCSI, FC, or GNBD. Basically this is a software RAID 1 where mirror parts may lag behind.

This makes it hard or impossible to communicate additional information other than the actual data payload, and is limited to strictly sequentially replaying the log. Lacking the meta data, on primary crash, it becomes impossible to reliably detect the most recent among several asynchronous remote copies. In case of data divergence, when more than one site thought it was in charge and they modified their respective copy of the data independently, this has to be detected and guarded against somewhere else.

To "reassemble" the surviving parts of a replication group, state information has to be kept, updated and communicated (replicated) elsewhere, for example by some cluster manager. The focus of this implementation apparently is on long-distance replication between existing SAN infrastructure.

4 Replicating Smarter

We can replace the default dm-repl-slink and dm-repl-log implementations with something smarter.

For long-distance replication, we can add overwrite detection and coalescing, or compression. For efficient compression, before we commit a write from the log to the local storage, we can XOR new and old data blocks, which is expected to produce large runs of zero bits for many real-world access patterns, that should compress very well.

As XOR "binary diffs" can be reverted, we can even store this compressed XOR replication stream in a second log, to provide generic point-in-time recovery for arbitrary data.

But, most importantly, we can communicate and store the exact "position" in the replication data stream.

4.1 dagtag: Identify a certain Data Set Version

If the data does not change, you could have created some clones, label them, and be done with it. To uniquely identify a writable replicated data set, you want to know where it is located (clone number), what it is about (label), who modified it last (writer/committer), and when (modification date).

Any data set (call it LUN, partition, LV) starts out new and fresh, with certain capacity but otherwise unused yet. During its lifetime, it receives a continuous stream of write requests, and maybe even change its capacity sometimes. When starting with an other data set with the same capacity on an other host, then feeding it the exact same data stream, it evolves in the same way, making it a clone of the original. Thus the "version" or "generation" of the data set at any point in its lifetime can then be identified by the position in this stream of change requests.

Counting "sectors written" alone is not sufficient, though. To uniquely identify a data set, we need to also identify the starting point, and the stream (of course).

To avoid divergence of the different clones of your data, there is an important simplification: at any given point there may be at most one entity coordinating the modifications to this set. Typically, there is only one master, so that master is the coordinator. For a multi-master setup, one cluster member needs to coordinate and arbitrate concurrent possibly conflicting writes by serialization.

When using replication as data back end for HA-Clusters, during cluster partition ("split brain") it can happen that more than one cluster partition changes their copy of the data.

To be able to reliably detect this data divergence once the cluster partitions rejoin, and to safely resync in an appropriate direction, I propose to describe a data set by a tuple of
<node id|(vol id|capacity)|sectors written|committer id>.

node id, "node name" An arbitrary, preferably "universally unique" id, which identifies which node is responsible for this clone. This id should be the same for all replicated volumes on the same node. It may correspond to the VG id in LVM.

replication volume id, "label" An arbitrary, preferably "universally unique" label, which is identical on all clones and identifies this "volume" set. This corresponds to the LV id in LVM.

capacity An LV may be resized several times in its life time, its capacity can change. I suggest to combine the id and the capacity of a volume to describe the replication stream.

Within the replication stream, we may chose to communicate not only the nominal capacity of the volume, but also the maximum of the block numbers used to far, which may be useful when doing on-demand resizing on the remote end of the replication stream.

monotonic volume time, "modification date" The most obvious, because implicitly derivable strictly monotonic increasing "time" is the number of sectors written to this volume since its creation.

If a replication data stream is optimized (by discarding overwrites, creating atomic batches, which move the data set from one consistency point to an other), it is vital that the original number of sectors written is recorded with the batch and restored when applied, so that this "volume time" will be the same on equivalent data set versions in different clones.

committer id This should be the *node id* of the node that was last responsible for coordinating modifications to this data set.

In multi-master mode (cluster filesystem mode), where the replicated clones logically form a "shared nothing shared disk", we may agree to use the "minimum" node id of the group of nodes simultaneously accessing this volume.

I'm going to call such a tuple a "data generation tag", or *dagtag*.

During successful replication, the sectors written should be increasing monotonically, and the other id fields should change infrequently.

A replication master knows the position of all its peers with each update acknowledgment they send back. We may also periodically broadcast this information to all nodes, so even secondaries currently not able to talk to each other know quite well how much they differ.

When we resize an LV, thus changing the capacity, we can encode this into the replication stream as equivalence of two such dagtags (and should record it into some history buffer).

When we activate a clone (promote it to be master, make it primary), it can first compare its local dagtag with the corresponding most recent dagtags of all respective other clones of that replicated volume (set). It can then suggest to not be activated, if it appears to be lagging behind.

When activated, the next write starts off a new change stream, which is identified by its starting point (the current dagtag), and the equivalence of current dagtag with that same dagtag substituting the new master node id for the committer id. We again store this equivalence in that history buffer, and communicate it. Then this first write, and all successive writes, just increment the number of sectors written.

Actually, current DRBD already uses a similar scheme, though entirely based on UUIDs, which carry meaning only by (in)equality within a (short) history sequence and the implicit knowledge of the situations that cause a new such UUID to be generated.

To shorten the history buffer, we can delete old history entries as soon as all known clones have successfully caught up to at least one newer entry.

This dagtag scheme adds more meaning and value to the data generation id tuples, making it suitable for arbitrary numbers of member nodes while being conveniently communicated implicitly – explicit dagtag exchange is only necessary on handshake and on role change, not during normal operation. As a side effect, the "distance" between two clones can be determined by just comparing the dagtags.

4.2 Daisy Chaining

Note that the dagtag scheme makes it easy and transparent to daisy chain replication links, or to hook up the furthest node in such a chain to the current master in case an intermediate replication node becomes unreachable.

4.3 dagtag ID Conventions

By default, a replicated volume shall be inaccessible by the host node (inactive), and only be target of an incoming replication data stream to be applied. We may activate it on at most one node. Then it becomes the origin of the replication data stream, and must not listen to incoming streams any longer.

Any replication data stream, whether it is the fully serialized original data stream or any "optimized" atomic batch, must record which data generation it is based on, and what data generation results from applying it.

For multiple active nodes, this changes only slightly, because the data stream still has to be serialized by the "coordinator".

Whenever we activate a replicated volume, we rotate its current dagtag into a history FIFO. We do not reset the "monotonic volume time". The new dagtag contains the same "monotonic volume time", with this nodes node id as the committer id. This shall be a "cluster wide transaction". The next modifying commit will contain this nodes node id, increasing the monotonic volume time appropriately (counting written sectors).

Also, whenever we lose a replication link, or when otherwise convenient or necessary for housekeeping, we record, and possibly rotate into the history FIFO, the current dagtag as well.

When the log fills up, and we fall back to bitmap based change tracking, we record the corresponding dagtag of the starting "position" in the replication stream against which this bitmap tracks changes. As long as a reconnecting node has a data generation at least as recent as the bitmap base dagtag (and not diverged), we can use the bitmap to efficiently resync only the changed blocks. If it has diverged from there, we still can construct a suitable quick-sync bitmap from the logs and bitmaps of both sides. During bitmap based resync, we may need a working copy of the bitmap to be able to resume an interrupted bitmap-based resync later without affecting correctness of later resynchronization of possible other disconnected nodes.

We may choose to rotate dagtags into that history FIFO while keeping the committer id, periodically or on administrative request, to deal with possible wrap-around of the "sectors written". Though I'd rather allow for a number as large as necessary to avoid wrap around in any case. Whether that means 64, 80, 128 or 160 bit wide integers is not important, as they don't get transmitted often, but only implicitly by the replication stream. If we store wall-clock time and some arbitrary user-supplied string along with the dagtags, this can be used as generic tagging mechanism of consistency points, and essentially implements check-pointing.

Time-Shift Replication, Point-in-Time Recovery If we occasionally (every few minutes, every N IO requests, every x MB written) communicate wall-clock time with the dagtag, we can easily implement a time-shift mode of operation, where one of the sites lags behind a defined delay on purpose, to provide a basis for easy point-in-time recovery after data corruption or operator error (together with the log, which needs to be large enough).

4.4 Multi-Master

DRBD already provides a "shared storage semantics" mode, where it allows two primaries to concurrently access the replicated storage, typically using cluster file systems.

Even though conflicting writes should never happen, as the cluster file systems are supposed to coordinate using a (distributed) lock manager, we need to be able to reliably detect concurrent conflicting writes, and arbitrate appropriately, so all clones will still end up in the same state. The algorithm we currently use in DRBD was last described in [2], and can be generalized to work in an n-master dm-replicator setup. Minor modifications are needed to make the replicator aware of membership and let the site links act bi-directional, not only being a log sink, replicating it to the remote sites, but also feeding the log with writes from remote.

One approach is to determine one coordinating master, which would receive all updates from all active clones via their site links, commit to its local log first, and let the respective other site links just replicate as normal. Conflict detection becomes trivial: the coordinator's log is the serialization point. Unfortunately that considerably adds latency for all write requests, unless originating from the coordinator itself.

The dagtag would use the coordinator's node id as "last committer id".

An other option is to establish a fully connected mesh of streams between all active clones. Still, all data is copied multiple times, once for each stream connection. So maybe we can use a reliable multicast protocol, where all datagrams are received by all members, but queued in memory until order (and conflict arbitration) is done by committing into the coordinator's log.

To implement the two-master case in this framework seems straight forward enough (using the first variation), and can even feed multiple not-active secondaries from there.

4.5 dagtag Examples

Assume we have a volume labeled "foo", and replicate that between several nodes with IDs A,B,C,D. We start from scratch, so there nothing has been written to this replicated volume yet.

initially configured A:foo:0:0 | B:foo:0:0 | C:foo:0:0 | D:foo:0:0
all nodes inactive

first activation Arbitrarily chose node B to fist activate it on: B:foo:0:B, communicating and recording equivalence of foo:0:0 and foo:0:B

first writes Do a mkfs, and assume it writes 300 sectors. The dagtag will change on node B to B:foo:300:B and it should be replicated to all other nodes, so they should eventually reach A:foo:300:B | B:foo:300:B | C:foo:300:B | D:foo:300:B. In case any other node is simultaneously activated, this is detected and can be arbitrated and roled back.

switchover deactivate B (nothing changes), and activate A (still, nothing changes in the dagtag)
i/a/i/i A:foo:300:B | B:foo:300:B | C:foo:300:B | D:foo:300:B

Now communicate and record the equivalence of foo:300:B and foo:300:A, so that any inactive node now knows that further modification shall be expected to originate from A. Again, this can be rolled back if for example somehow D would be simultaneously activated.

Write the first sector on the new master: A:foo:301:A which will be replicated to the other nodes. Since every replication stream must contain its starting point, that replication payload would look like

starting point	foo:300:A,
offset/length	XYZ/one sector
result	foo:301:A
actual data block	binary blob

where the result tag is redundant, because implicit by (starting point + length). Actually, for all but the first request in an established stream even the starting point may be left off, as it is implicit the result of the previous request. The dagtag scheme does not incur any communication overhead.

4.5.1 Communicate Changes

Think of any part of the replication stream as a binary diff, describing how you can move from a certain data generation to an other.

Whether we communicate and apply this diff in strict sequential order, just like the original write requests reached the active device, or optimize away overwrites within a certain time frame or write-amount window, is not so important, as long as we keep the information about the starting point (which is assumed to be a consistency point) and the generation tag resulting from applying this diff.

We still have to apply each such diff "atomically", see *atomic batches*.

4.5.2 Detect and Use the Best Data Set

When two clones establish their replication link, they do some handshake authenticating against each other, and exchanging their data generation tags.

Using their knowledge of previous data generations, they can determine which node has the most recent data, and where the replication needs to be resumed.

4.5.3 Avoid Data Diversion

When activating a clone, the former coordinating/committing node must agree, or must have been fenced. This is a job for a cluster manager.

4.5.4 Detect and Recover from Data Diversion

During handshake, it can easily be detected if and when (in monotonic volume time) the clones diverged from a common "ancestor". It is then possible to tell the "distance" of each clone's current data set generation from that common ancestor, and thus from each other, and present the administrator with this helpful information to sort out the mess, who has to decide which data set generation to keep, and which version to throw away, since generic block level merging of arbitrary data is impossible.

Depending on those distances, and whether we kept enough XOR-based undo-log information, it may be possible to smoothly move the to-be-overwritten clone consistently to the selected data.

If that is not possible, there needs to be a partial resynchronization, using and combining the dirty-bitmaps, or constructing dirty bitmaps from the redo-log information, or even a full sync. The sync target will be inconsistent during such a bitmap based resync..

4.6 Multi-Master, asynchronous

At first, it would appear that the replication protocol for multi-master has to be strictly synchronous.

Now, this is a vague idea and may be proven wrong. But when we transfer all membership and lock exchange communication in-band with the ordered change stream, it may be possible to run in slightly

asynchronous, "near-synchronous" mode, to run even stretched clusters with cluster file systems and acceptable local performance, as long as most of the time the different "branch offices" actively access only different parts of the device.

The recovery from replication link loss becomes more involved then, this needs to be discussed with cluster file system developers.

For access patterns that require write-locks to bounce between sites too frequently, such a setup is obviously prohibitive, though.

5 Conclusion

We are going to add necessary hooks for the dagtag scheme to dm-replicator, and add "intelligent" dm-repl-slinks. We probably need to write an optimized, and optimizing, dm-repl-log to leverage batch and compression possibilities. I don't expect anything production-ready soon, but we may be in good shape for a beta in summer 2009. Please direct comments, complaints, questions and suggestions to the author, or to drbd-dev@lists.linbit.com.

Disclaimer

All thoughts and ideas presented here are my own. Which of course does not mean they are unique. Storage replication has been around for decades, ever since there is storage to replicate. I believe in convergent evolution, in the sense that people thinking about similar problems having similar resources will find similar solutions.

I have to expect that most of my proposals have already been solved and implemented decades ago in some proprietary replication solution. Which again makes it likely that some of this is covered by software patents. Possibly calculating XOR parity information in storage context is covered, maybe the idea of combining compression and replication, or even the idea of storage replication *per se*. As software patents are (still) illegal in Europe, at least theoretically, I don't care, and did not investigate.

I want to recommend a paper [3] I stumbled across after the fact. They give a detailed taxonomy of replication solutions and present a "provable" finite-state engine for the remote mirroring protocol, which we might use as a starting point for our implementation. I borrowed from their terminology, as my own was not yet that concise.

References

- [1] Heinz Mauelshagen, *Device Mapper Remote Replication Target*, Proceedings of Linux-Kongress 2008; (this very proceedings).
- [2] Lars Ellenberg, *DRBD v8.0.x and beyond*, LinuxConf Europe, September 2-5, 2007, Cambridge, England; available online at <http://www.drbd.org/home/publications/>
- [3] Minwen Ji, Alistair Veitch, John Wilkes, HP Laboratories, Palo Alto, CA
Seneca: remote mirroring done write,
Proceedings of USENIX Technical Conference (San Antonio, TX), pages 253–268, June 2003.
USENIX, Berkeley, CA.
last seen online at http://www.hpl.hp.com/personal/John_Wilkes/papers/Seneca-USENIX03-paper.pdf