The DRBD9 and LINSTOR User's Guide

Table of Contents

Please Read This First

This guide is intended to serve users of the Distributed Replicated Block Device version 9 (DRBD-9) as a definitive reference guide and handbook.

It is being made available to the DRBD community by LINBIT, the project's sponsor company, free of charge and in the hope that it will be useful. The guide is constantly being updated. We try to add information about new DRBD features simultaneously with the corresponding DRBD releases. An on-line HTML version of this guide is always available at https://links.linbit.com/DRBD9-Users-Guide.

This guide assumes, throughout, that you are using the latest version of DRBD and related tools. If you are using an 8.4 release of DRBD, please use the matching version of this guide from https://links.linbit.com/DRBD84-Users-Guide.

Please use the drbd-user mailing list to submit comments.

This guide is organized as follows:

- Introduction to DRBD deals with DRBD's basic functionality. It gives a short overview of DRBD's positioning within the Linux I/O stack, and about fundamental DRBD concepts. It also examines DRBD's most important features in detail.
- Building and installing the DRBD software talks about building DRBD from source, installing pre-built DRBD packages, and contains an overview of getting DRBD running on a cluster system.
- LINSTOR is about using LINSTOR for centralized management of storage volumes and DRBD resources. This software-defined storage approach is especially useful for large clusters.
- Working with DRBD is about managing DRBD using resource configuration files, as well as common troubleshooting scenarios.
- DRBD-enabled applications deals with leveraging DRBD to add storage replication and high availability to applications. It not only covers DRBD integration in the Pacemaker cluster manager, but also advanced LVM configurations, integration of DRBD with GFS, and adding high availability to Xen virtualization environments.
- Optimizing DRBD performance contains pointers for getting the best performance out of DRBD configurations.
- Learning more dives into DRBD's internals, and also contains pointers to other resources which readers of this guide may find useful.
- Appendices:
    - Recent changes is an overview of changes in DRBD 9.0, compared to earlier DRBD versions.

Users interested in DRBD training or support services are invited to contact us at sales@linbit.com or sales_us@linbit.com.

Introduction to DRBD

Chapter 1. DRBD Fundamentals

DRBD is a software-based, shared-nothing, replicated storage solution mirroring the content of block devices (hard disks, partitions, logical volumes etc.) between hosts.

DRBD mirrors data

- **in real time**. Replication occurs continuously while applications modify the data on the device.
- **transparently**. Applications need not be aware that the data is stored on multiple hosts.
- **synchronously** or **asynchronously**. With synchronous mirroring, applications are notified of write completions after the writes have been carried out on all (connected) hosts. With asynchronous mirroring, applications are notified of write completions when the writes have completed locally, which usually is before they have propagated to the other hosts.

1.1. Kernel module

DRBD's core functionality is implemented by way of a Linux kernel module. Specifically, DRBD constitutes a driver for a virtual block device, so DRBD is situated right near the bottom of a system's I/O stack. Because of this, DRBD is extremely flexible and versatile, which makes it a replication solution suitable for adding high availability to just about any application.

DRBD is, by definition and as mandated by the Linux kernel architecture, agnostic of the layers above it. Thus, it is impossible for DRBD to miraculously add features to upper layers that these do not possess. For example, DRBD cannot auto-detect file system corruption or add active-active clustering capability to file systems like ext3 or XFS.

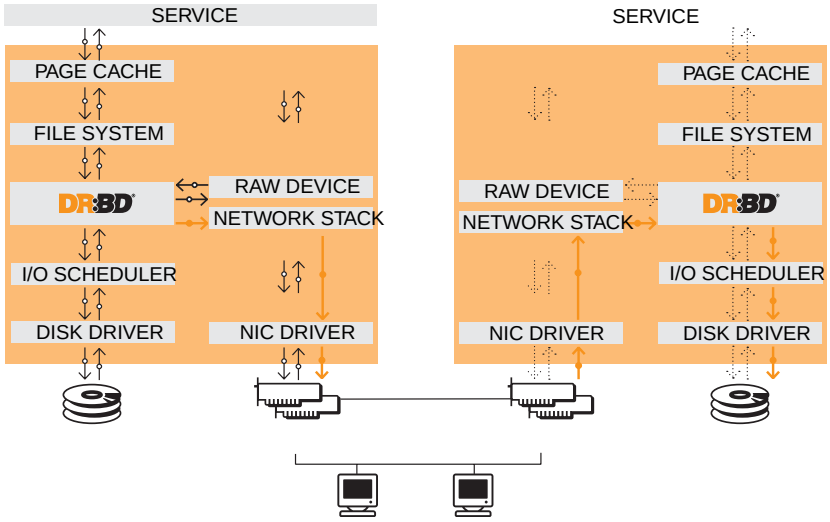


Figure 1. DRBD's position within the Linux I/O stack

1.2. User space administration tools

DRBD comes with a set of administration tools which communicate with the kernel module in order to configure and administer DRBD resources. From top-level to bottom-most these are:

`drbdmanage`

Provided as a separate project, this is the recommended way to orchestrate DRBD resources in a multi-node cluster. DRBD Manage uses one DRBD 9 resource to store its cluster-wide configuration data, and offers a quick and easy way to perform the most often needed administrative tasks, by calling external programs like `lvcreate` and `drbdadm`.

Please see the DRBD Manage entry in this documentation for more details.

`drbdadm`

The high-level administration tool of the DRBD-utils program suite. Obtains all DRBD configuration parameters from

the configuration file `/etc/drbd.conf` and acts as a front-end for `drbdsetup` and `drbdmeta`. `drbdadm` has a *dry-run* mode, invoked with the `-d` option, that shows which `drbdsetup` and `drbdmeta` calls `drbdadm` would issue without actually calling those commands.

`drbdsetup`

Configures the DRBD module that was loaded into the kernel. All parameters to `drbdsetup` must be passed on the command line. The separation between `drbdadm` and `drbdsetup` allows for maximum flexibility. Most users will rarely need to use `drbdsetup` directly, if at all.

`drbdmeta`

Allows to create, dump, restore, and modify DRBD meta data structures. Like `drbdsetup`, most users will only rarely need to use `drbdmeta` directly.

1.3. Resources

In DRBD, *resource* is the collective term that refers to all aspects of a particular replicated data set. These include:

Resource name

This can be any arbitrary, US-ASCII name not containing whitespace by which the resource is referred to.

Volumes

Any resource is a replication group consisting of one of more *volumes* that share a common replication stream. DRBD ensures write fidelity across all volumes in the resource. Volumes are numbered starting with `0`, and there may be up to 65,535 volumes in one resource. A volume contains the replicated data set, and a set of metadata for DRBD internal use.

At the `drbdadm` level, a volume within a resource can be addressed by the resource name and volume number as *`resource/volume`*.

DRBD device

This is a virtual block device managed by DRBD. It has a device major number of 147, and its minor numbers are numbered from 0 onwards, as is customary. Each DRBD device corresponds to a volume in a resource. The associated block device is usually named `/dev/drbd`*X*, where *X* is the device minor number. `udev` will typically also create symlinks containing the resource name and volume number, as in `/dev/drbd/by-res/`*`resource/vol-nr`*.

Very early DRBD versions hijacked NBD's device major number 43. This is long obsolete; 147 is the LANANA-registered DRBD device major.

Connection

A *connection* is a communication link between two hosts that share a replicated data set. With DRBD 9 each resource can be defined on multiple hosts; with the current versions this requires a full-meshed connection setup between these hosts (ie. each host connected to every other for that resource)

At the `drbdadm` level, a connection is addressed by the resource and the connection name (the latter defaulting to the peer hostname), like *`resource:connection`*.

1.4. Resource roles

In DRBD, every resource has a role, which may be *Primary* or *Secondary*.

The choice of terms here is not arbitrary. These roles were deliberately not named "Active" and "Passive" by DRBD's creators. *Primary* vs. *secondary* refers to a concept related to availability of **storage**, whereas *active* vs. *passive* refers to the availability of an **application**. It is usually the case in a high-availability environment that the *primary* node is also the *active* one, but this is by no means necessary.

- A DRBD device in the *primary* role can be used unrestrictedly for read and write operations. It may be used for

creating and mounting file systems, raw or direct I/O to the block device, etc.

- A DRBD device in the *secondary* role receives all updates from the peer node's device, but otherwise disallows access completely. It can not be used by applications, neither for read nor write access. The reason for disallowing even read-only access to the device is the necessity to maintain cache coherency, which would be impossible if a secondary resource were made accessible in any way.

The resource's role can, of course, be changed, either by manual intervention, by way of some automated algorithm by a cluster management application, or automatically. Changing the resource role from secondary to primary is referred to as *promotion*, whereas the reverse operation is termed *demotion*.

Chapter 2. DRBD Features

This chapter discusses various useful DRBD features, and gives some background information about them. Some of these features will be important to most users, some will only be relevant in very specific deployment scenarios. Working with DRBD and Troubleshooting and error recovery contain instructions on how to enable and use these features in day-to-day operation.

2.1. Single-primary mode

In single-primary mode, a resource is, at any given time, in the primary role on only one cluster member. Since it is guaranteed that only one cluster node manipulates the data at any moment, this mode can be used with any conventional file system (ext3, ext4, XFS etc.).

Deploying DRBD in single-primary mode is the canonical approach for High-Availability (fail-over capable) clusters.

2.2. Dual-primary mode

In dual-primary mode, a resource is, at any given time, in the primary role on two cluster nodes [1: Perhaps that feature will be renamed to "Multi Primary" later on, when it actually works ;)]. Since concurrent access to the data is thus possible, this mode requires the use of a shared cluster file system that utilizes a distributed lock manager. Examples include GFS and OCFS2.

Deploying DRBD in dual-primary mode is the preferred approach for load-balancing clusters which require concurrent data access from two nodes, eg. virtualization environments with a need for live-migration. This mode is disabled by default, and must be enabled explicitly in DRBD's configuration file.

See Enabling dual-primary mode for information on enabling dual-primary mode for specific resources.

2.3. Replication modes

DRBD supports three distinct replication modes, allowing three degrees of replication synchronicity.

Protocol A

Asynchronous replication protocol. Local write operations on the primary node are considered completed as soon as the local disk write has finished, and the replication packet has been placed in the local TCP send buffer. In the event of forced fail-over, data loss may occur. The data on the standby node is consistent after fail-over; however, the most recent updates performed prior to the crash could be lost. Protocol A is most often used in long distance replication scenarios. When used in combination with DRBD Proxy it makes an effective disaster recovery solution. See Long-distance replication via DRBD Proxy, for more information.

Protocol B

Memory synchronous (semi-synchronous) replication protocol. Local write operations on the primary node are considered completed as soon as the local disk write has occurred, and the replication packet has reached the peer node. Normally, no writes are lost in case of forced fail-over. However, in the event of simultaneous power failure on both nodes **and** concurrent, irreversible destruction of the primary's data store, the most recent writes completed on the primary may be lost.

Protocol C

Synchronous replication protocol. Local write operations on the primary node are considered completed only after both the local and the remote disk write(s) have been confirmed. As a result, loss of a single node is guaranteed not to lead to any data loss. Data loss is, of course, inevitable even with this replication protocol if all nodes (resp. their storage subsystems) are irreversibly destroyed at the same time.

By far, the most commonly used replication protocol in DRBD setups is protocol C.

The choice of replication protocol influences two factors of your deployment: *protection* and *latency*. *Throughput*, by contrast, is largely independent of the replication protocol selected.

See Configuring your resource for an example resource configuration which demonstrates replication protocol configuration.

2.4. More than 2-way redundancy

With DRBD 9 it's possible to have the data stored simultaneously on more than two cluster nodes.

While this has been possible before via stacking, in DRBD 9 this is supported out-of-the-box for (currently) up to 16 nodes. (In practice, using 3-, 4- or perhaps 5-way redundancy via DRBD will make other things the leading cause of downtime.)

The major difference to the stacking solution is that there's less performance loss, because only one level of data replication is being used.

2.5. Automatic Promotion of Resources

Prior to DRBD 9, promoting a resource would be done with the `drbdadm primary` command. With DRBD 9, DRBD will automatically promote a resource to primary role when the `auto-promote` option is enabled, and one of its volumes is mounted or opened for writing. As soon as all volumes are unmounted or closed, the role of the resource changes back to secondary.

Automatic promotion will only succeed if the cluster state allows it (that is, if an explicit `drbdadm primary` command would succeed). Otherwise, mounting or opening the device fails as it did prior to DRBD 9.

2.6. Multiple replication transports

DRBD supports multiple network transports. As of now two transport implementations are available: TCP and RDMA. Each transport implementation comes as its own kernel module.

2.6.1. TCP Transport

The `drbd_transport_tcp.ko` transport implementation is included with the distribution files of drbd itself. As the name implies, this transport implementation uses the TCP/IP protocol to move data between machines.

DRBD's replication and synchronization framework socket layer supports multiple low-level transports:

TCP over IPv4

This is the canonical implementation, and DRBD's default. It may be used on any system that has IPv4 enabled.

TCP over IPv6

When configured to use standard TCP sockets for replication and synchronization, DRBD can use also IPv6 as its network protocol. This is equivalent in semantics and performance to IPv4, albeit using a different addressing scheme.

SDP

SDP is an implementation of BSD-style sockets for RDMA capable transports such as InfiniBand. SDP was available as part of the OFED stack of most distributions but is now **considered deprecated**. SDP uses an IPv4-style addressing scheme. Employed over an InfiniBand interconnect, SDP provides a high-throughput, low-latency replication network to DRBD.

SuperSockets

SuperSockets replace the TCP/IP portions of the stack with a single, monolithic, highly efficient and RDMA capable socket implementation. DRBD can use this socket type for very low latency replication. SuperSockets must run on specific hardware which is currently available from a single vendor, Dolphin Interconnect Solutions.

2.6.2. RDMA Transport

Alternatively the `drbd_transport_rdma.ko` kernel module is available from LINBIT. This transport uses the verbs/RDMA API to move data over InfiniBand HCAs, iWARP capable NICs or RoCE capable NICs. In contrast to the BSD sockets API (used by TCP/IP) the verbs/RDMA API allows data movement with very little CPU involvement.

2.6.3. Conclusion

At high transfer rates it might be possible that the CPU load/memory bandwidth of the tcp transport becomes the limiting factor. You can probably achieve higher transfer rates using the rdma transport with appropriate hardware.

A transport implementation can be configured for each connection of a resource. See Configuring transport implementations for more details.

2.7. Efficient synchronization

(Re-)synchronization is distinct from device replication. While replication occurs on any write event to a resource in the primary role, synchronization is decoupled from incoming writes. Rather, it affects the device as a whole.

Synchronization is necessary if the replication link has been interrupted for any reason, be it due to failure of the primary node, failure of the secondary node, or interruption of the replication link. Synchronization is efficient in the sense that DRBD does not synchronize modified blocks in the order they were originally written, but in linear order, which has the following consequences:

- Synchronization is fast, since blocks in which several successive write operations occurred are only synchronized once.
- Synchronization is also associated with few disk seeks, as blocks are synchronized according to the natural on-disk block layout.
- During synchronization, the data set on the standby node is partly obsolete and partly already updated. This state of data is called *inconsistent*.

The service continues to run uninterrupted on the active node, while background synchronization is in progress.

A node with inconsistent data generally cannot be put into operation, thus it is desirable to keep the time period during which a node is inconsistent as short as possible. DRBD does, however, ship with an LVM integration facility that automates the creation of LVM snapshots immediately before synchronization. This ensures that a *consistent* copy of the data is always available on the peer, even while synchronization is running. See Using automated LVM snapshots during DRBD synchronization for details on using this facility.

2.7.1. Variable-rate synchronization

In variable-rate synchronization (the default since 8.4), DRBD detects the available bandwidth on the synchronization network, compares it to incoming foreground application I/O, and selects an appropriate synchronization rate based on a fully automatic control loop.

See Variable sync rate configuration for configuration suggestions with regard to variable-rate synchronization.

2.7.2. Fixed-rate synchronization

In fixed-rate synchronization, the amount of data shipped to the synchronizing peer per second (the *synchronization rate*) has a configurable, static upper limit. Based on this limit, you may estimate the expected sync time based on the following simple formula:

$$t_{resync} = \frac{D}{R}$$

Figure 2. Synchronization time

t_{sync} is the expected sync time. D is the amount of data to be synchronized, which you are unlikely to have any influence over (this is the amount of data that was modified by your application while the replication link was broken). R is the rate of synchronization, which is configurable — bounded by the throughput limitations of the replication network and I/O subsystem.

See Configuring the rate of synchronization for configuration suggestions with regard to fixed-rate synchronization.

2.7.3. Checksum-based synchronization

The efficiency of DRBD's synchronization algorithm may be further enhanced by using data digests, also known as checksums. When using checksum-based synchronization, then rather than performing a brute-force overwrite of blocks marked out of sync, DRBD *reads* blocks before synchronizing them and computes a hash of the contents currently found on disk. It then compares this hash with one computed from the same sector on the peer, and omits re-writing this block if the hashes match. This can dramatically cut down synchronization times in situation where a filesystem re-writes a sector with identical contents while DRBD is in disconnected mode.

See Configuring checksum-based synchronization for configuration suggestions with regard to synchronization.

2.8. Suspended replication

If properly configured, DRBD can detect if the replication network is congested, and *suspend* replication in this case. In this mode, the primary node "pulls ahead" of the secondary — temporarily going out of sync, but still leaving a consistent copy on the secondary. When more bandwidth becomes available, replication automatically resumes and a background synchronization takes place.

Suspended replication is typically enabled over links with variable bandwidth, such as wide area replication over shared connections between data centers or cloud instances.

See Configuring congestion policies and suspended replication for details on congestion policies and suspended replication.

2.9. On-line device verification

On-line device verification enables users to do a block-by-block data integrity check between nodes in a very efficient manner.

Note that *efficient* refers to efficient use of network bandwidth here, and to the fact that verification does not break redundancy in any way. On-line verification is still a resource-intensive operation, with a noticeable impact on CPU utilization and load average.

It works by one node (the *verification source*) sequentially calculating a cryptographic digest of every block stored on the lower-level storage device of a particular resource. DRBD then transmits that digest to the peer node(s) (the *verification target(s)*), where it is checked against a digest of the local copy of the affected block. If the digests do not match, the block is marked out-of-sync and may later be synchronized. Because DRBD transmits just the digests, not the full blocks, on-line verification uses network bandwidth very efficiently.

The process is termed *on-line* verification because it does not require that the DRBD resource being verified is unused at the time of verification. Thus, though it does carry a slight performance penalty while it is running, on-line verification does not cause service interruption or system down time — neither during the verification run nor during subsequent synchronization.

It is a common use case to have on-line verification managed by the local cron daemon, running it, for example, once a week or once a month. See Using on-line device verification for information on how to enable, invoke, and automate on-line verification.

2.10. Replication traffic integrity checking

DRBD optionally performs end-to-end message integrity checking using cryptographic message digest algorithms such as MD5, SHA-1, or CRC-32C.

These message digest algorithms are **not** provided by DRBD, but by the Linux kernel crypto API; DRBD merely uses them. Thus, DRBD is capable of utilizing any message digest algorithm available in a particular system's kernel configuration.

With this feature enabled, DRBD generates a message digest of every data block it replicates to the peer, which the peer then uses to verify the integrity of the replication packet. If the replicated block can not be verified against the digest, the connection is dropped and immediately re-established; because of the bitmap the typical result is a retransmission. Thus, DRBD replication is protected against several error sources, all of which, if unchecked, would potentially lead to data corruption during the replication process:

- Bitwise errors ("bit flips") occurring on data in transit between main memory and the network interface on the sending node (which goes undetected by TCP checksumming if it is offloaded to the network card, as is common in recent implementations);
- Bit flips occurring on data in transit from the network interface to main memory on the receiving node (the same considerations apply for TCP checksum offloading);
- Any form of corruption due to a race conditions or bugs in network interface firmware or drivers;
- Bit flips or random corruption injected by some reassembling network component between nodes (if not using direct, back-to-back connections).

See Configuring replication traffic integrity checking for information on how to enable replication traffic integrity checking.

2.11. Split brain notification and automatic recovery

Split brain is a situation where, due to temporary failure of all network links between cluster nodes, and possibly due to intervention by a cluster management software or human error, both nodes switched to the *Primary* role while disconnected. This is a potentially harmful state, as it implies that modifications to the data might have been made on either node, without having been replicated to the peer. Thus, it is likely in this situation that two diverging sets of data have been created, which cannot be trivially merged.

DRBD split brain is distinct from cluster split brain, which is the loss of all connectivity between hosts managed by a distributed cluster management application such as Heartbeat. To avoid confusion, this guide uses the following convention:

- *Split brain* refers to DRBD split brain as described in the paragraph above.
- Loss of all cluster connectivity is referred to as a *cluster partition*, an alternative term for cluster split brain.

DRBD allows for automatic operator notification (by email or other means) when it detects split brain. See Split brain notification for details on how to configure this feature.

While the recommended course of action in this scenario is to manually resolve the split brain and then eliminate its root cause, it may be desirable, in some cases, to automate the process. DRBD has several resolution algorithms available for doing so:

- **Discarding modifications made on the younger primary.** In this mode, when the network connection is re-established and split brain is discovered, DRBD will discard modifications made, in the meantime, on the node which switched to the primary role *last*.
- **Discarding modifications made on the older primary.** In this mode, DRBD will discard modifications made, in the meantime, on the node which switched to the primary role *first*.
- **Discarding modifications on the primary with fewer changes.** In this mode, DRBD will check which of the two nodes has recorded fewer modifications, and will then discard *all* modifications made on that host.
- **Graceful recovery from split brain if one host has had no intermediate changes.** In this mode, if one of

the hosts has made no modifications at all during split brain, DRBD will simply recover gracefully and declare the split brain resolved. Note that this is a fairly unlikely scenario. Even if both hosts only mounted the file system on the DRBD block device (even read-only), the device contents typically would be modified (eg. by filesystem journal replay), ruling out the possibility of automatic recovery.

Whether or not automatic split brain recovery is acceptable depends largely on the individual application. Consider the example of DRBD hosting a database. The discard modifications from host with fewer changes approach may be fine for a web application click-through database. By contrast, it may be totally unacceptable to automatically discard *any* modifications made to a financial database, requiring manual recovery in any split brain event. Consider your application's requirements carefully before enabling automatic split brain recovery.

Refer to Automatic split brain recovery policies for details on configuring DRBD's automatic split brain recovery policies.

2.12. Support for disk flushes

When local block devices such as hard drives or RAID logical disks have write caching enabled, writes to these devices are considered completed as soon as they have reached the volatile cache. Controller manufacturers typically refer to this as write-back mode, the opposite being write-through. If a power outage occurs on a controller in write-back mode, the last writes are never committed to the disk, potentially causing data loss.

To counteract this, DRBD makes use of disk flushes. A disk flush is a write operation that completes only when the associated data has been committed to stable (non-volatile) storage — that is to say, it has effectively been written to disk, rather than to the cache. DRBD uses disk flushes for write operations both to its replicated data set and to its meta data. In effect, DRBD circumvents the write cache in situations it deems necessary, as in activity log updates or enforcement of implicit write-after-write dependencies. This means additional reliability even in the face of power failure.

It is important to understand that DRBD can use disk flushes only when layered on top of backing devices that support them. Most reasonably recent kernels support disk flushes for most SCSI and SATA devices. Linux software RAID (md) supports disk flushes for RAID-1 provided that all component devices support them too. The same is true for device-mapper devices (LVM2, dm-raid, multipath).

Controllers with battery-backed write cache (BBWC) use a battery to back up their volatile storage. On such devices, when power is restored after an outage, the controller flushes all pending writes out to disk from the battery-backed cache, ensuring that all writes committed to the volatile cache are actually transferred to stable storage. When running DRBD on top of such devices, it may be acceptable to disable disk flushes, thereby improving DRBD's write performance. See Disabling backing device flushes for details.

2.13. Trim/Discard support

Trim/*Discard* are two names for the same feature: a request to a storage system, telling it that some data range is not being used anymore [2: For example, a deleted file's data.] and can get recycled.
This call originates in Flash-based storages (SSDs, FusionIO cards, etc.), which cannot easily *rewrite* a sector but instead have to *erase* and write the (new) data again (incurring some latency cost). For more details, see eg. the [[https://en.wikipedia.org/wiki/Trim_%28computing%29,wikipedia page]].

Since 8.4.3 DRBD includes support for *Trim*/*Discard*. You don't need to configure or enable anything; if DRBD detects that the local (underlying) storage system allows using these commands, it will transparently enable them and pass such requests through.

The effect is that eg. a recent-enough `mkfs.ext4` on a multi-TB volume can shorten the initial sync time to a few seconds to minutes - just by telling DRBD (which will relay that information to all connected nodes) that most/all of the storage is now to be seen as invalidated.

Nodes that connect to that resource later on will not have seen the *Trim*/*Discard* requests, and will therefore start a full resync; depending on kernel version and file system a call to `fstrim` might give the wanted result, though.

even if you don't have storage with *Trim/Discard* support, some virtual block devices will provide you with the same feature, for example Thin LVM.

2.14. Disk error handling strategies

If a hard drive fails which is used as a backing block device for DRBD on one of the nodes, DRBD may either pass on the I/O error to the upper layer (usually the file system) or it can mask I/O errors from upper layers.

Passing on I/O errors

If DRBD is configured to pass on I/O errors, any such errors occurring on the lower-level device are transparently passed to upper I/O layers. Thus, it is left to upper layers to deal with such errors (this may result in a file system being remounted read-only, for example). This strategy does not ensure service continuity, and is hence not recommended for most users.

Masking I/O errors

If DRBD is configured to *detach* on lower-level I/O error, DRBD will do so, automatically, upon occurrence of the first lower-level I/O error. The I/O error is masked from upper layers while DRBD transparently fetches the affected block from a peer node, over the network. From then onwards, DRBD is said to operate in diskless mode, and carries out all subsequent I/O operations, read and write, on the peer node(s) only. Performance in this mode will be reduced, but the service continues without interruption, and can be moved to the peer node in a deliberate fashion at a convenient time.

See Configuring I/O error handling strategies for information on configuring I/O error handling strategies for DRBD.

2.15. Strategies for dealing with outdated data

DRBD distinguishes between *inconsistent* and *outdated* data. Inconsistent data is data that cannot be expected to be accessible and useful in any manner. The prime example for this is data on a node that is currently the target of an ongoing synchronization. Data on such a node is part obsolete, part up to date, and impossible to identify as either. Thus, for example, if the device holds a filesystem (as is commonly the case), that filesystem would be unexpected to mount or even pass an automatic filesystem check.

Outdated data, by contrast, is data on a secondary node that is consistent, but no longer in sync with the primary node. This would occur in any interruption of the replication link, whether temporary or permanent. Data on an outdated, disconnected secondary node is expected to be clean, but it reflects a state of the peer node some time past. In order to avoid services using outdated data, DRBD disallows promoting a resource that is in the outdated state.

DRBD has interfaces that allow an external application to outdate a secondary node as soon as a network interruption occurs. DRBD will then refuse to switch the node to the primary role, preventing applications from using the outdated data. A complete implementation of this functionality exists for the Pacemaker cluster management framework (where it uses a communication channel separate from the DRBD replication link). However, the interfaces are generic and may be easily used by any other cluster management application.

Whenever an outdated resource has its replication link re-established, its outdated flag is automatically cleared. A background synchronization then follows.

See the section about the DRBD outdate-peer daemon (dopd) for an example DRBD/Heartbeat/Pacemaker configuration enabling protection against inadvertent use of outdated data.

2.16. Three-way replication via stacking

Available in DRBD version 8.3.0 and above; deprecated in DRBD version 9.x, as more nodes can be implemented on a single level. See Defining network connections for details.

When using three-way replication, DRBD adds a third node to an existing 2-node cluster and replicates data to that node, where it can be used for backup and disaster recovery purposes. This type of configuration generally involves Long-distance replication via DRBD Proxy.

Three-way replication works by adding another, *stacked* DRBD resource on top of the existing resource holding your production data, as seen in this illustration:

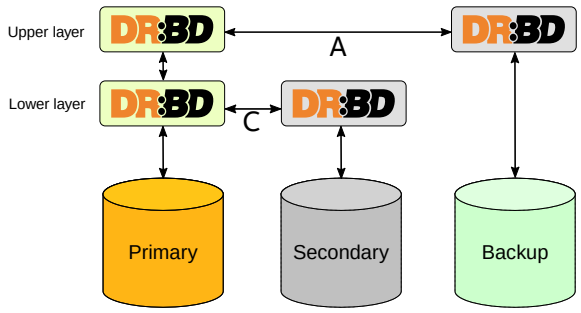


Figure 3. DRBD resource stacking

The stacked resource is replicated using asynchronous replication (DRBD protocol A), whereas the production data would usually make use of synchronous replication (DRBD protocol C).

Three-way replication can be used permanently, where the third node is continuously updated with data from the production cluster. Alternatively, it may also be employed on demand, where the production cluster is normally disconnected from the backup site, and site-to-site synchronization is performed on a regular basis, for example by running a nightly cron job.

2.17. Long-distance replication via DRBD Proxy

DRBD's protocol A is asynchronous, but the writing application will block as soon as the socket output buffer is full (see the `sndbuf-size` option in the man page of `drbd.conf`). In that event, the writing application has to wait until some of the data written runs off through a possibly small bandwidth network link.

The average write bandwidth is limited by available bandwidth of the network link. Write bursts can only be handled gracefully if they fit into the limited socket output buffer.

You can mitigate this by DRBD Proxy's buffering mechanism. DRBD Proxy will place changed data from the DRBD device on the primary node into its buffers. DRBD Proxy's buffer size is freely configurable, only limited by the address room size and available physical RAM.

Optionally DRBD Proxy can be configured to compress and decompress the data it forwards. Compression and decompression of DRBD's data packets might slightly increase latency. However, when the bandwidth of the network link is the limiting factor, the gain in shortening transmit time outweighs the compression and decompression overhead.

Compression and decompression were implemented with multi core SMP systems in mind, and can utilize multiple CPU cores.

The fact that most block I/O data compresses very well and therefore the effective bandwidth increases justifies the use of the DRBD Proxy even with DRBD protocols B and C.

See Using DRBD Proxy for information on configuring DRBD Proxy.

DRBD Proxy is one of the few parts of the DRBD product family that is not published under an open source license. Please contact sales@linbit.com or sales_us@linbit.com for an evaluation license.

2.18. Truck based replication

Truck based replication, also known as disk shipping, is a means of preseeding a remote site with data to be replicated, by physically shipping storage media to the remote site. This is particularly suited for situations where

- the total amount of data to be replicated is fairly large (more than a few hundreds of gigabytes);
- the expected rate of change of the data to be replicated is less than enormous;
- the available network bandwidth between sites is limited.

In such situations, without truck based replication, DRBD would require a very long initial device synchronization (on the order of weeks, months, or years). Truck based replication allows to ship a data seed to the remote site, and so drastically reduces the initial synchronization time. See Using truck based replication for details on this use case.

2.19. Floating peers

This feature is available in DRBD versions 8.3.2 and above.

A somewhat special use case for DRBD is the *floating peers* configuration. In floating peer setups, DRBD peers are not tied to specific named hosts (as in conventional configurations), but instead have the ability to float between several hosts. In such a configuration, DRBD identifies peers by IP address, rather than by host name.

For more information about managing floating peer configurations, see Configuring DRBD to replicate between two SAN-backed Pacemaker clusters.

2.20. Data rebalancing (horizontal storage scaling)

If your company's policy says that 3-way redundancy is needed, you need at least 3 servers for your setup.

Now, as your storage demands grow, you will encounter the need for additional servers. Rather than having to buy 3 more servers at the same time, you can *rebalance* your data across a single additional node.

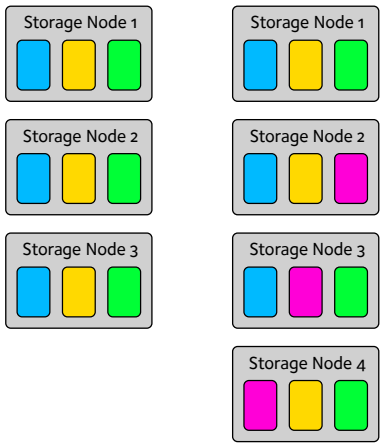


Figure 4. DRBD data rebalancing

In the figure above you can see the *before* and *after* states: from 3 nodes with three 25TiB volumes each (for a net 75TiB), to 4 nodes, with net 100TiB.

DRBD 9 makes it possible to do an online, live migration of the data; please see Data rebalancing for the exact steps needed.

2.21. DRBD client

With the multiple-peer feature of DRBD a number of interesting use-cases have been added, for example the *DRBD client*.

The basic idea is that the DRBD *backend* can consist of 3, 4, or more nodes (depending on the policy of required redundancy); but, as DRBD 9 can connect more nodes than that. DRBD works then as a storage access protocol in addition to storage replication.

All write requests executed on a primary *DRBD client* gets shipped to all nodes equipped with storage. Read requests are only shipped to one of the server nodes. The *DRBD client* will evenly distribute the read requests among all available server nodes.

See Permanently diskless nodes for the configuration file syntax or in case you are using drbdmanage, use its `--client` option when assigning resources to nodes.

2.22. Quorum

In order to avoid split brain or diverging data of replicas one has to configure fencing. It turns out that in real world deployments node fencing is not popular because often mistakes happen in planning or deploying it.

In the moment a data-set has 3 replicas one can rely on the quorum implementation within DRBD instead of Pacemaker level fencing. Pacemaker gets informed about quorum or loss-of-quorum via the master score of the resource.

DRBD's quorum can be used with any kind of Linux based service. In case the service terminates in the moment it is exposed to an IO-error the on quorum loss behavior is very elegant. In case the service does not terminate upon IO-error the systems needs to be configured to reboot a primary node that looses quorum.

See Configuring quorum for more information.

2.22.1. Tiebreaker

The quorum tiebreaker feature is available in DRBD versions 9.0.18 and above.

The fundamental problem with two node clusters is that in the moment they lose connectivity we have two partitions and none of them has quorum, which results in the cluster halting the service. This problem can be mitigated by adding a third, diskless node to the cluster which will then act as a quorum tiebreaker.

See Using a diskless node as a tiebreaker for more information.

2.23. DRBD integration for VCS

Veritas Cluster Server (or Veritas Infoscale Availabilty) is a commercial alternative to the Pacemaker open source software. In case you need to integrate DRBD resources into a VCS setup please see the README in drbd-utils/scripts/VCS on github.

Building and installing the DRBD software

Chapter 3. Installing pre-built DRBD binary packages

3.1. Packages supplied by LINBIT

LINBIT, the DRBD project's sponsor company, provides binary packages to its commercial support customers. These packages are available via repositories (e.g., `apt`, `yum`), and when reasonable via LINBIT's docker registry. Packages/images frome these sources are considered "official" builds.

These builds are available for the following distributions:

- Red Hat Enterprise Linux (RHEL), versions 6 and 7
- SUSE Linux Enterprise Server (SLES), versions 11SP4, and 12
- Debian GNU/Linux, 8 (jessie), and 9 (stretch)
- Ubuntu Server Edition LTS 14.04 (Trusty Tahr), LTS 16.04 (Xenial Xerus), and LTS 18.04 (Bionic Beaver).

Packages for some other distributions are built as well, but don't receive as much testing.

LINBIT releases binary builds in parallel with any new DRBD source release.

Package installation on RPM-based systems (SLES, RHEL) is done by simply invoking `yum install` (for new installations) or `yum update` (for upgrades).

For Debian-based systems (Debian GNU/Linux, Ubuntu) systems, `drbd-utils` and `drbd-dkms` packages are installed with `apt`, or similar tools like `aptitude` or `synaptic`, if available.

3.2. Docker images supplied by LINBIT

LINBIT provides a Docker registry for its commercial support customers. The registry is accessible via the host name 'drbd.io'. Before you can pull images, you have to log in to the registry:

```
# docker login drbd.io
```

After a successful login, you can pull images. To test your login and the registry, start by issuing the following command:

```
# docker pull drbd.io/alpine
# docker run -it --rm drbd.io/alpine # press CTRL-D to exit
```

3.3. Packages supplied by distribution vendors

A number of distributions provide DRBD, including pre-built binary packages. Support for these builds, if any, is being provided by the associated distribution vendor. Their release cycle may lag behind DRBD source releases.

3.3.1. SUSE Linux Enterprise Server

SLES High Availability Extension (HAE) includes DRBD.

On SLES, DRBD is normally installed via the software installation component of YaST2. It comes bundled with the High Availability package selection.

Users who prefer a command line install may simply issue:

```
# yast -i drbd
```

or

```
# zypper install drbd
```

3.3.2. CentOS

CentOS has had DRBD 8 since release 5; for DRBD 9 you'll need to look at EPEL and similar sources.

DRBD can be installed using `yum` (note that you will need a correct repository enabled for this to work):

```
# yum install drbd kmod-drbd
```

3.3.3. Ubuntu Linux

For Ubuntu LTS, LINBIT offers a PPA repository at https://launchpad.net/~linbit/+archive/ubuntu/linbit-drbd9-stack. See Adding Launchpad PPA Repositories for more information.

```
# apt install drbd-utils python-drbdmanage drbd-dkms
```

3.4. Compiling packages from source

Releases generated by git tags on github are snapshots of the git repository at the given time. You most likely do not want to use these. They might lack things such as generated man pages, the `configure` script, and other generated files. If you want to build from a tarball, use the ones provided by us.

All our projects contain standard build scripts (e.g., `Makefile`, `configure`). Maintaining specific information per distribution (e.g., documenting broken build macros) is too cumbersome, and historically the information provided in this section got outdated quickly. If you don't know how to build software the standard way, please consider using packages provided by LINBIT.

LINSTOR

Chapter 4. Basic administrative tasks / Setup

LINSTOR is a configuration management system for storage on Linux systems. It manages LVM logical volumes and/or ZFS ZVOLs on a cluster of nodes. It leverages DRBD for replication between different nodes and to provide block storage devices to users and applications. It manages snapshots, encryption and caching of HDD backed data in SSDs via bcache.

4.1. Concepts and Terms

This section goes over core concepts and terms that you will need to familiarize yourself with to understand how LINSTOR works and deploys storage. The section is laid out in a "ground up" approach.

4.1.1. Installable Components

linstor-controller

A LINSTOR setup requires at least one active controller and one or more satellites.

The *linstor-controller* contains the database that holds all configuration information for the whole cluster. It makes all decisions that need to have a view of the whole cluster. The controller is typically deployed as a HA service using Pacemaker and DRBD as it is a crucial part of the system. Multiple controllers can be used for LINSTOR but only one can be active.

linstor-satellite

The *linstor-satellite* runs on each node where LINSTOR consumes local storage or provides storage to services. It is stateless; it receives all the information it needs from the controller. It runs programs like `lvcreate` and `drbdadm`. It acts like a node agent.

linstor-client

The *linstor-client* is a command line utility that you use to issue commands to the system and to investigate the status of the system.

4.1.2. Objects

Objects are the end result which LINSTOR presents to the end-user or application, such as; Kubernettes/OpenShift, a replicated block device (DRBD), NVMeOF target, etc.

Node

Node's are a server or container that participate in a LINSTOR cluster. The *Node* attribute defines:

- Determines which LINSTOR cluster the node participates in
- Sets the role of the node: Controller, Satellite, Auxiliary
- **NetInterface** objects define the node's connectivity

NetInterface

As the name implies, this is how you define the interface/address of a node's network interface.

Definitions

Definitions define attributes of an object, they can be thought of as profile or template. Objects created will inherit the configuration defined in the definitions. A definition must be defined prior to creating the associated object. For

example; you must create a *ResourceDefinition* prior to creating the *Resource*

StoragePoolDefinition

- Defines the name of a storage pool

ResourceDefinition

Resource definitions define the following attributes of a resource:

- The name of a DRBD resource
- The TCP port for DRBD to use for the resource's connection

VolumeDefinition

Volume definitions define the following:

- A volume of a DRBD resource
- The size of the volume
- The volume number of the DRBD resource's volume
- The meta data properties of the volume
- The minor number to use for the DRBD device associated with the DRBD volume

StoragePool

The *StoragePool* identifies storage in the context of LINSTOR. It defines:

- The configuration of a storage pool on a specific node
- The storage back-end driver to use for the storage pool on the cluster node (LVM, ZFS, etc)
- The parameters and configuration to pass to the storage backed driver

Resource

LINSTOR has now has expanded its capabilities to manage a broader set of storage technologies outside of just DRBD. A *Resource*:

- Represents the placement of a DRBD resource, as defined within the *ResourceDefinition*
- Places a resource on a node in the cluster
- Defines the placement of a *ResourceDefinition* on a node

Volume

Volumes are a subset of a *Resource*. A *Resource* could have multiple volumes, for example you may wish to have your database stored on slower storage than your logs in your MySQL cluster. By keeping the *volumes* under a single *resource* you are essentially creating a consistency group. The *Volume* attribute can define also define attributes on a more granular level.

4.2. Broader Context

While LINSTOR might be used to make the management of DRBD more convenient, it is often integrated with software stacks higher up. Such integration exist already for Kubernetes, OpenStack, OpenNebula and Proxmox. Chapters specific to deploying LINSTOR in these environments are included in this guide.

The southbound drivers used by LINSTOR are LVM, thinLVM and ZFS with support for Swordfish in progress.

4.3. Packages

LINSTOR is packaged in both the .rpm and the .deb variants:

1. *linstor-client* contains the command line client program. It depends on python which is usually already installed. In RHEL 8 systems you will need to symlink python
2. *linstor-controller* and *linstor-satellite* Both contain systemd unit files for the services. They depend on Java runtime environment (JRE) version 1.8 (headless) or higher.

For further detail about these packages see the Installable Components section above.

If you have a support subscription to LINBIT, you will have access to our certified binaries via our repositories.

4.4. Installation

If you want to use LINSTOR in containers skip this Topic and use the "Containers" section below for the installation.

4.4.1. Ubuntu Linux

If you want to have the option of creating replicated storage using DRBD, you will need to install *drbd-dkms* and *drbd-utils*. These packages will need to be installed on all nodes. You will also need to choose a volume manager, either ZFS or LVM, in this instance we're using LVM.

```
# apt install -y drbd-dkms drbd-utils lvm2
```

Depending on whether your node is a LINSTOR controller, satellite, or both (Combined) will determine what packages are required on that node. For combined type nodes, we'll need both the controller and satellite LINSTOR package.

Combined node:

```
# apt install linstor-controller linstor-satellite  linstor-client
```

That will make our remaining nodes our Satellites, so we'll need to install the following packages on them:

```
# apt install linstor-satellite  linstor-client
```

4.4.2. SUSE Linux Enterprise Server

SLES High Availability Extension (HAE) includes DRBD.

On SLES, DRBD is normally installed via the software installation component of YaST2. It comes bundled with the High Availabilty package selection.

As we download DRBD's newest module we can check if the LVM-tools are up to date as well. User who prefer a command line install may simply issue the following command to get the newest DRBD and LVM version:

```
# zypper install drbd lvm2
```

Depending on whether your node is a LINSTOR controller, satellite, or both (Combined) will determine what packages

are required on that node. For combined type nodes, we'll need both the controller and satellite LINSTOR package.

Combined node:

```
# zypper install linstor-controller linstor-satellite  linstor-client
```

That will make our remaining nodes our Satellites, so we'll need to install the following packages on them:

```
# zypper install linstor-satellite  linstor-client
```

4.4.3. CentOS

CentOS has had DRBD 8 since release 5. For DRBD 9 you'll need to look at EPEL and similar sources. Alternatively, if you have an active support contract with LINBIT you can utilize our RHEL 8 repositories. DRBD can be installed using `yum`. We can also check for the newest version of the LVM-tools as well.

LINSTOR **requires** DRBD 9 if you wish to have replicated storage. This requires an external repository to be configured, either LINBIT's or a 3rd parties.

```
# yum install drbd kmod-drbd lvm2
```

Depending on whether your node is a LINSTOR controller, satellite, or both (Combined) will determine what packages are required on that node. For combined type nodes, we'll need both the controller and satellite LINSTOR package.

On RHEL 8 systems you will need to install python2 for the linstor-client to work.

Combined node:

```
# yum install linstor-controller linstor-satellite  linstor-client
```

That will make our remaining nodes our Satellites, so we'll need to install the following packages on them:

```
# yum install linstor-satellite  linstor-client
```

4.5. Containers

LINSTOR is also available as containers. The base images are available in LINBIT's container registry, `drbd.io`.

In order to access the images, you first have to login to the registry (reach out to sales@linbit.com for credentials):

```
# docker login drbd.io
```

The containers available in this repo are:

- drbd.io/drbd9:rhel8
- drbd.io/drbd9:rhel7
- drbd.io/drbd9:bionic

- drbd.io/linstor-csi
- drbd.io/linstor-controller
- drbd.io/linstor-satellite
- drbd.io/linstor-client

An up to date list of available images with versions can be retrieved by opening http://drbd.io in your browser. Make sure to access the host via "http", as the registry's images themselves are served via "https".

To load the kernel module, needed only for LINSTOR satellites, you'll need to run a `drbd9` container in privileged mode. The kernel module containers either retrieve an official LINBIT package from a customer repository, or they try to build the kernel modules from source. If you intend to build from source, you need to have the according kernel headers (e.g., `kernel-devel`) installed on the host. There are 3 ways to execute such a module load container:

- Specifying a LINBIT node hash and a distribution.
- Bind-mounting an existing repository configuration.
- Not doing one of the above to trigger a build from source.

Example using a hash and a distribution:

```
# docker run -it --rm --privileged -v /lib/modules:/lib/modules \
  -e LB_DIST=rhel7.7 -e LB_HASH=ThisIsMyNodeHash \
  drbd.io/drbd9:rhel7
```

Example using an existing repo config.

```
# docker run -it --rm --privileged -v /lib/modules:/lib/modules \
  -v /etc/yum.repos.d/linbit.repo:/etc/yum.repos.d/linbit.repo:ro \
  drbd.io/drbd9:rhel7
```

In both cases (hash + distribution, as well as bind-mounting) the hash or config has to be from a node that has a special property set. Feel free to contact our support, and we set this property.

Example building from shipped source (RHEL based):

```
# docker run -it --rm --privileged -v /lib/modules:/lib/modules \
  -v /usr/src:/usr/src:ro \
  drbd.io/drbd9:rhel7
```

Example building from shipped source (Debian based):

```
# docker run -it --rm --privileged -v /lib/modules:/lib/modules \
  -v /usr/src:/usr/src:ro -v /usr/lib:/usr/lib:ro \
  drbd.io/drbd9:bionic
```

Do **not** bind-mount `/usr/lib` on RHEL based systems! And do **not** forget to bind-mount it on Debian based ones.

For now (i.e., pre DRBD 9 version "9.0.17"), you must use the containerized DRBD kernel module, as opposed to loading a kernel module onto the host system. If you intend to use the containers you should not install the DRBD kernel module on your host systems. For DRBD version 9.0.17 or greater, you can install the kernel module as usual on the host system, but you need to make sure to load the module with the `usermode_helper=disabled` parameter (e.g., `modprobe drbd usermode_helper=disabled`).

Then run the LINSTOR satellite container, also privileged, as a daemon:

```
# docker run -d --name=linstor-satellite --net=host --privileged drbd.io/linstor-satellite
```

`net=host` is required for the containerized `drbd-utils` to be able to communicate with the host-kernel via netlink.

To run the LINSTOR controller container as a daemon, mapping ports `3370`, `3376` and `3377` on the host to the container:

```
# docker run -d --name=linstor-controller -p 3370:3370 -p 3376:3376 -p 3377:3377
drbd.io/linstor-controller
```

To interact with the containerized LINSTOR cluster, you can either use a LINSTOR client installed on a system via packages, or via the containerized LINSTOR client. To use the LINSTOR client container:

```
# docker run -it --rm -e LS_CONTROLLERS=<controller-host-IP-address> drbd.io/linstor-client node
list
```

From this point you would use the LINSTOR client to initialize your cluster and begin creating resources using the typical LINSTOR patterns.

To stop and remove a daemonized container and image:

```
# docker stop linstor-controller
# docker rm linstor-controller
```

4.6. Initializing your cluster

We assume that the following steps are accomplished on **all** cluster nodes:

1. The DRBD9 kernel module is installed and loaded
2. `drbd-utils` are installed
3. LVM tools are installed
4. `linstor-controller` and/or `linstor-satellite` its dependencies are installed
5. The `linstor-client` is installed on the `linstor-controller` node

Start and enable the linstor-controller service on the host where it has been installed:

```
# systemctl enable --now linstor-controller
```

If you are sure the linstor-controller service gets automatically enabled on installation you can use the following

command as well:

```
# systemctl start linstor-controller
```

4.7. Using the LINSTOR client

Whenever you run the LINSTOR command line client, it needs to know where your linstor-controller runs. If you do not specify it, it will try to reach a locally running linstor-controller listening on IP `127.0.0.1` port `3376`. Therefore we will use the `linstor-client` on the same host as the `linstor-controller`.

The `linstor-satellite` requires ports 3366 and 3367. The `linstor-controller` requires ports 3376 and 3377. Make sure you have these ports allowed on your firewall.

```
# linstor node list
```

should give you an empty list and not an error message.

You can use the `linstor` command on any other machine, but then you need to tell the client how to find the linstor-controller. As shown, this can be specified as a command line option, an environment variable, or in a global file:

```
# linstor --controllers=alice node list
# LS_CONTROLLERS=alice linstor node list
```

Alternatively you can create the `/etc/linstor/linstor-client.conf` file and populate it like below.

```
[global]
controllers=alice
```

If you have multiple linstor-controllers configured you can simply specify them all in a comma separated list. The linstor-client will simply try them in the order listed.

The linstor-client commands can also be used in a much faster and convenient way by only writing the starting letters of the parameters e.g.: `linstor node list` → `linstor n l`

4.8. Adding nodes to your cluster

The next step is to add nodes to your LINSTOR cluster. You need to provide:

1. A node name which **must** match the output of `uname -n`
2. The IP address of the node.

```
# linstor node create bravo 10.43.70.3
```

When you use `linstor node list` you will see that the new node is marked as offline. Now start and enable the linstor-satellite on that node so that the service comes up on reboot as well:

```
# systemctl enable --now  linstor-satellite
```

You can also use `systemctl start linstor-satellite` if you are sure that the service is already enabled as default and comes up on reboot.

About 10 seconds later you will see the status in `linstor node list` becoming online. Of course the satellite process may be started before the controller knows about the existence of the satellite node.

In case the node which hosts your controller should also contribute storage to the LINSTOR cluster, you have to add it as a node and start the linstor-satellite as well.

4.9. Storage pools

StoragePools identify storage in the context of LINSTOR. To group storage pools from multiple nodes, simply use the same name on each node. For example, one valid approach is to give all SSDs one name and all HDDs another.

On each host contributing storage, you need to create either an LVM VG or a ZFS zPool. The VGs and zPools identified with one LINSTOR storage pool name may have different VG or zPool names on the hosts, but do yourself a favor and use the same VG or zPool name on all nodes.

```
# vgcreate vg_ssd /dev/nvme0n1 /dev/nvme1n1 [...]
```

These then need to be registered with LINSTOR:

```
# linstor storage-pool create lvm alpha pool_ssd vg_ssd
# linstor storage-pool create lvm bravo pool_ssd vg_ssd
```

The storage pool name and common metadata is referred to as a *storage pool definition*. The listed commands create a storage pool definition implicitly. You can see that by using `linstor storage-pool-definition list`. Creating storage pool definitions explicitly is possible but not necessary.

To list your storage-pools you can use:

```
# linstor storage-pool list
```

or using the short version

```
# linstor sp l
```

In case anything goes wrong with the storage pool's VG/zPool, e.g. the VG having been renamed or somehow became invalid you can delete the storage pool in LINSTOR with the following command, given that only resources with all their volumes in the so-called 'lost' storage pool are attached. This feature is available since LINSTOR v0.9.13.

```
# linstor storage-pool lost alpha pool_ssd
```

or using the short version

```
# linstor sp lo alpha pool_ssd
```

Should the deletion of the storage pool be prevented due to attached resources or snapshots with some of its volumes in another still functional storage pool, hints will be given in the 'status' column of the corresponding list-command

(e.g. `linstor resource list`). After deleting the LINSTOR-objects in the lost storage pool manually, the lost-command can be executed again to ensure a complete deletion of the storage pool and its remaining objects.

4.9.1. A storage pool per backend device

In clusters where you have only one kind of storage and the capability to hot-repair storage devices, you may choose a model where you create one storage pool per physical backing device. The advantage of this model is to confine failure domains to a single storage device.

4.10. Cluster configuration

4.10.1. Available storage plugins

LINSTOR has the following supported storage plugins as of writing:

- Thick LVM
- Thin LVM with a single thin pool
- Thick ZFS
- Thin ZFS

4.11. Creating and deploying resources/volumes

In the following scenario we assume that the goal is to create a resource 'backups' with a size of '500 GB' that is replicated among three cluster nodes.

First, we create a new resource definition:

```
# linstor resource-definition create backups
```

Second, we create a new volume definition within that resource definition:

```
# linstor volume-definition create backups 500G
```

If you want to change the size of the volume-definition you can simply do that by:

```
# linstor volume-definition set-size backups 0 100G
```

The parameter `0` is the number of the volume in the resource `backups`. You have to provide this parameter , because resources can have multiple volumes and they are identified by a so called volume-number. This number can be found by listing the volume-definitions.

The size of a volume-definition can only be decreased if it has no resource. Despite of that the size can be increased even with an deployed resource.

So far we have only created objects in LINSTOR's database, not a single LV was created on the storage nodes. Now you have the choice of delegating the task of placement to LINSTOR or doing it yourself.

4.11.1. Manual placement

With the `resource create` command you may assign a resource definition to named nodes explicitly.

```
# linstor resource create alpha backups --storage-pool pool_hdd
# linstor resource create bravo backups --storage-pool pool_hdd
# linstor resource create charlie backups --storage-pool pool_hdd
```

4.11.2. Autoplace

The value after autoplace tells LINSTOR how many replicas you want to have. The storage-pool option should be obvious.

```
# linstor resource create backups --auto-place 3 --storage-pool pool_hdd
```

Maybe not so obvious is that you may omit the `--storage-pool` option, then LINSTOR may select a storage pool on its own. The selection follows these rules:

- Ignore all nodes and storage pools the current user has no access to
- Ignore all diskless storage pools
- Ignore all storage pools not having enough free space

From the remaining storage pools, LINSTOR currently chooses the one with the most available free space.

If everything went right the DRBD-resource has now been created by LINSTOR. This can be checked by looking for the DRBD block device with the `lsblk` command which should look like `drbd0000` or similar.

Now we should be able to mount the block device of our resource and start using LINSTOR.

Chapter 5. Further LINSTOR tasks

5.1. DRBD clients

By using the `--diskless` option instead of `--storage-pool` you can have a permanently diskless DRBD device on a node. This means that the resource will appear as Blockdevice and can be mounted to the filesystem without an existing storage-device. The data of the resource is accessed over the network on another nodes with the same resource.

```
# linstor resource create delta backups --diskless
```

5.2. LINSTOR – DRBD consistency group/multiple volumes

The so called consistency group is a feature from DRBD. It is mentioned in this user-guide, due to the fact that one of LINSTOR's main functions is to manage storage-clusters with DRBD. Multiple volumes in one resource are a consistency group.

This means that changes on different volumes from one resource are getting replicated in the same chronological order on the other Satellites.

Therefore you don't have to worry about the timing if you have interdependent data on different volumes in a resource.

To deploy more than one volume in a LINSTOR-resource you have to create two volume-definitions with the same name.

```
# linstor volume-definition create backups 500G
# linstor volume-definition create backups 100G
```

5.3. Volumes of one resource to different Storage-Pools

This can be achieved by setting the `StorPoolName` property to the volume definitions before the resource is deployed to the nodes:

```
# linstor resource-definition create backups
# linstor volume-definition create backups 500G
# linstor volume-definition create backups 100G
# linstor volume-definition set-property backups 0 StorPoolName pool_hdd
# linstor volume-definition set-property backups 1 StorPoolName pool_ssd
# linstor resource create alpha backups
# linstor resource create bravo backups
# linstor resource create charlie backups
```

Since the `volume-definition create` command is used without the `--vlmnr` option LINSTOR assigned the volume numbers starting at 0. In the following two lines the 0 and 1 refer to these automatically assigned volume numbers.

Here the 'resource create' commands do not need a `--storage-pool` option. In this case LINSTOR uses a 'fallback' storage pool. Finding that storage pool, LINSTOR queries the properties of the following objects in the following order:

- Volume definition

- Resource
- Resource definition
- Node

If none of those objects contain a `StorPoolName` property, the controller falls back to a hard-coded 'DfltStorPool' string as a storage pool.

This also means that if you forgot to define a storage pool prior deploying a resource, you will get an error message that LINSTOR could not find the storage pool named 'DfltStorPool'.

5.4. LINSTOR without DRBD

LINSTOR can be used without DRBD as well. Without DRBD, LINSTOR is able to provision volumes from LVM and ZFS backed storage pools, and create those volumes on individual nodes in your LINSTOR cluster.

Currently LINSTOR supports the creation of LVM and ZFS volumes with the option of layering some combinations of LUKS, DRBD, and/or NVMe-oF/NVMe-TCP on top of those volumes.

For example, assume we have a Thin LVM backed storage pool defined in our LINSTOR cluster named, `thin-lvm`:

```
# linstor --no-utf8 storage-pool list
+-------------------------------------------------------------+
| StoragePool | Node      | Driver   | PoolName          | ... |
|-------------------------------------------------------------|
| thin-lvm    | linstor-a | LVM_THIN | drbdpool/thinpool | ... |
| thin-lvm    | linstor-b | LVM_THIN | drbdpool/thinpool | ... |
| thin-lvm    | linstor-c | LVM_THIN | drbdpool/thinpool | ... |
| thin-lvm    | linstor-d | LVM_THIN | drbdpool/thinpool | ... |
+-------------------------------------------------------------+
```

We could use LINSTOR to create a Thin LVM on `linstor-d` that's 100GiB in size using the following commands:

```
# linstor resource-definition create rsc-1
# linstor volume-definition create rsc-1 100GiB
# linstor resource create --layer-list storage \
          --storage-pool thin-lvm linstor-d rsc-1
```

You should then see you have a new Thin LVM on `linstor-d`. You can extract the device path from LINSTOR by listing your linstor resources with the `--machine-readable` flag set:

```
# linstor --machine-readable resource list | grep device_path
            "device_path": "/dev/drbdpool/rsc-1_00000",
```

If you wanted to layer DRBD on top of this volume, which is the default `--layer-list` option in LINSTOR for ZFS or LVM backed volumes, you would use the following resource creation pattern instead:

```
# linstor resource-definition create rsc-1
# linstor volume-definition create rsc-1 100GiB
# linstor resource create --layer-list drbd,storage \
          --storage-pool thin-lvm linstor-d rsc-1
```

You would then see that you have a new Thin LVM backing a DRBD volume on `linstor-d`:

```
# linstor --machine-readable resource list | grep -e device_path -e backing_disk
            "device_path": "/dev/drbd1000",
            "backing_disk": "/dev/drbdpool/rsc-1_00000",
```

The complete list of currently supported `--layer-list` combinations are as follows:

- drbd,luks,storage
- drbd,storage
- luks,storage
- nvme,storage
- storage

For information about the prerequisites for the `luks` layer, refer to the Encrypted Volumes section of this User's Guide.

5.4.1. NVMe-oF/NVMe-TCP LINSTOR Layer

NVMe-oF/NVMe-TCP allows LINSTOR to connect diskless resources to a node with the same resource where the data is stored over NVMe fabrics. This leads to the advantage that resources can be mounted without using local storage by accessing the data over the network. LINSTOR is not using DRBD in this case, and therefore NVMe resources provisioned by LINSTOR are not replicated, the data is stored on one node.

NVMe-oF only works on RDMA-capable networks and NVMe-TCP on every network that can carry IP traffic. If you want to know more about NVMe-oF/NVMe-TCP visit https://www.linbit.com/en/nvme-linstor-swordfish/ for more information.

To use NVMe-oF/NVMe-TCP with LINSTOR the package `nvme-cli` needs to be installed on every Node which acts as a Satellite and will use NVMe-oF/NVMe-TCP for a resource:

If you are not using Ubuntu use the suitable command for installing packages on your OS - SLES: zypper - CentOS: yum

```
# apt install nvme-cli
```

To make a resource which uses NVMe-oF/NVMe-TCP an additional parameter has to be given as you create the resource-definiton:

```
# linstor resource-definition create nvmedata  -l nvme,storage
```

As default the -l (layer-stack) parameter is set to `drbd, storage` when DRBD is used. If you want to create LINSTOR resources with neither NVMe nor DBRD you have to set the `-l` parameter to only `storage`.

Create the volume-definition for our resource:

```
# linstor volume-definiton create nvmedata 500G
```

Before you create the resource on your nodes you have to know where the data will be stored locally and which node accesses it over the network.

First we create the resource on the node where our data will be stored:

```
# linstor resource create alpha nvmedata --storage-pool pool_ssd
```

On the nodes where the resource-data will be accessed over the network, the resource has to be defined as diskless:

```
# linstor resource create beta nvmedata -d
```

The `-d` parameter creates the resource on this node as diskless.

Now you can mount the resource `nvmedata` on one of your nodes.

If your nodes have more than one NIC you should force the route between them for NVMe-of/NVME-TCP, otherwise multiple NIC's could cause troubles.

5.5. Managing Network Interface Cards

LINSTOR can deal with multiple network interface cards (NICs) in a machine, they are called `netif` in LINSTOR speak.

When a satellite node is created a first `netif` gets created implicitly with the name `default`. Using the `--interface-name` option of the `node create` command you can give it a different name.

Additional NICs are created like this:

```
# linstor node interface create alpha 100G_nic 192.168.43.221
# linstor node interface create alpha 10G_nic 192.168.43.231
```

NICs are identified by the IP address only, the name is arbitrary and is **not** related to the interface name used by Linux. The NICs can be assigned to storage pools so that whenever a resource is created in such a storage pool, the DRBD traffic will be routed through the specified NIC.

```
# linstor storage-pool set-property alpha pool_hdd PrefNic 10G_nic
# linstor storage-pool set-property alpha pool_ssd PrefNic 100G_nic
```

FIXME describe how to route the controller <-> client communication through a specific `netif`.

5.6. Encrypted volumes

LINSTOR can handle transparent encryption of drbd volumes. dm-crypt is used to encrypt the provided storage from the storage device.

Basic steps to use encryption:

1. Disable user security on the controller (this will be obsolete once authentication works)
2. Create a master passphrase
3. Create a volume definition with the `--encrypt` option
4. Don't forget to re-enter the master passphrase after a controller restart.

5.6.1. Disable user security

Disabling the user security on the `Linstor` controller is a one time operation and is afterwards persisted.

1. Stop the running linstor-controller via systemd: `systemctl stop linstor-controller`
2. Start a linstor-controller in debug mode: `/usr/share/linstor-server/bin/Controller -c /etc/linstor -d`
3. In the debug console enter: `setSecLvl secLvl(NO_SECURITY)`
4. Stop linstor-controller with the debug shutdown command: `shutdown`
5. Start the controller again with systemd: `systemctl start linstor-controller`

5.6.2. Encrypt commands

Below are details about the commands.

Before LINSTOR can encrypt any volume a master passphrase needs to be created. This can be done with the linstor-client.

```
# linstor encryption create-passphrase
```

`crypt-create-passphrase` will wait for the user to input the initial master passphrase (as all other crypt commands will with no arguments).

If you ever want to change the master passphrase this can be done with:

```
# linstor encryption modify-passphrase
```

To mark which volumes should be encrypted you have to add a flag while creating a volume definition, the flag is is `--encrypt` e.g.:

```
# linstor volume-definition create crypt_rsc 1G --encrypt
```

To enter the master passphrase (after controller restart) use the following command:

```
# linstor encryption enter-passphrase
```

Whenever the linstor-controller is restarted, the user has to send the master passphrase to the controller, otherwise LINSTOR is unable to reopen or create encrypted volumes.

5.7. Checking the state of your cluster

LINSTOR provides various commands to check the state of your cluster. These commands start with a 'list-' prefix and provide various filtering and sorting options. The '--groupby' option can be used to group and sort the output in multiple dimensions.

```
# linstor node list
# linstor storage-pool list --groupby Size
```

5.8. Managing snapshots

Snapshots are supported with thin LVM and ZFS storage pools.

5.8.1. Creating a snapshot

Assuming a resource definition named 'resource1' which has been placed on some nodes, a snapshot can be created as follows:

```
# linstor snapshot create resource1 snap1
```

This will create snapshots on all nodes where the resource is present. LINSTOR will ensure that consistent snapshots are taken even when the resource is in active use.

5.8.2. Restoring a snapshot

The following steps restore a snapshot to a new resource. This is possible even when the original resource has been removed from the nodes where the snapshots were taken.

First define the new resource with volumes matching those from the snapshot:

```
# linstor resource-definition create resource2
# linstor snapshot volume-definition restore --from-resource resource1 --from-snapshot snap1
--to-resource resource2
```

At this point, additional configuration can be applied if necessary. Then, when ready, create resources based on the snapshots:

```
# linstor snapshot resource restore --from-resource resource1 --from-snapshot snap1 --to
-resource resource2
```

This will place the new resource on all nodes where the snapshot is present. The nodes on which to place the resource can also be selected explicitly; see the help (`linstor snapshot resource restore -h`).

5.8.3. Rolling back to a snapshot

LINSTOR can roll a resource back to a snapshot state. The resource must not be in use. That is, it may not be mounted on any nodes. If the resource is in use, consider whether you can achieve your goal by restoring the snapshot instead.

Rollback is performed as follows:

```
# linstor snapshot rollback resource1 snap1
```

A resource can only be rolled back to the most recent snapshot. To roll back to an older snapshot, first delete the intermediate snapshots.

5.8.4. Removing a snapshot

An existing snapshot can be removed as follows:

```
# linstor snapshot delete resource1 snap1
```

5.9. Setting options for resources

DRBD options are set using LINSTOR commands. Configuration in files such as `/etc/drbd.d/global_common.conf` that are not managed by LINSTOR will be ignored. The following commands show the usage and available options:

```
# linstor controller drbd-options -h
# linstor resource-definition drbd-options -h
# linstor volume-definition drbd-options -h
# linstor resource drbd-peer-options -h
```

For instance, it is easy to set the DRBD protocol for a resource named `backups`:

```
# linstor resource-definition drbd-options --protocol C backups
```

5.10. Adding and removing disks

LINSTOR can convert resources between diskless and having a disk. This is achieved with the `resource toggle-disk` command, which has syntax similar to `resource create`.

For instance, add a disk to the diskless resource `backups` on 'alpha':

```
# linstor resource toggle-disk alpha backups --storage-pool pool_ssd
```

Remove this disk again:

```
# linstor resource toggle-disk alpha backups --diskless
```

5.10.1. Migrating disks

In order to move a resource between nodes without reducing redundancy at any point, LINSTOR's disk migrate feature can be used. First create a diskless resource on the target node, and then add a disk using the `--migrate-from` option. This will wait until the data has been synced to the new disk and then remove the source disk.

For example, to migrate a resource `backups` from 'alpha' to 'bravo':

```
# linstor resource create bravo backups --diskless
# linstor resource toggle-disk bravo backups --storage-pool pool_ssd --migrate-from alpha
```

5.11. DRBD Proxy with LINSTOR

LINSTOR can be used to configure DRBD Proxy for long-distance replication. DRBD Proxy must first be installed and licensed as described in Using DRBD Proxy.

LINSTOR expects DRBD Proxy to be running on the nodes which are involved in the relevant connections. It does not currently support connections via DRBD Proxy on a separate node.

Suppose our cluster consists of nodes 'alpha' and 'bravo' in a local network and 'charlie' at a remote site, with a resource definition named `backups` deployed to each of the nodes. Then DRBD Proxy can be enabled for the connections to 'charlie' as follows:

```
# linstor drbd-proxy enable alpha charlie backups
# linstor drbd-proxy enable bravo charlie backups
```

The DRBD Proxy configuration can be tailored with commands such as:

```
# linstor drbd-proxy options backups --memlimit 100000000
# linstor drbd-proxy compression zlib backups --level 9
```

LINSTOR does not automatically optimize the DRBD configuration for long-distance replication, so you will probably want to set some configuration options such as the protocol:

```
# linstor resource-connection drbd-options alpha charlie backups --protocol A
# linstor resource-connection drbd-options bravo charlie backups --protocol A
```

Please contact LINBIT for assistance optimizing your configuration.

5.12. External database

It is possible to have LINSTOR working with an external database provider like Postgresql or MariaDB. To use an external database there are a few additional steps to configure.

1. The JDBC database driver for your database needs to be downloaded and installed to the LINSTOR library directory.
2. The `/etc/linstor/linstor.toml` configuration file needs to be editied for your database setup.

5.12.1. Postgresql

Postgresql JDBC driver can be downloaded here:

https://jdbc.postgresql.org/download.html

And afterwards copied to: `/usr/share/linstor-server/lib/`

A sample Postgresql `linstor.toml` looks like this:

```
[db]
user = "linstor"
password = "linstor"
connection_url = "jdbc:postgresql://localhost/linstor"
```

5.12.2. MariaDB/Mysql

MariaDB JDBC driver can be downloaded here:

https://downloads.mariadb.org/connector-java/

And afterwards copied to: `/usr/share/linstor-server/lib/`

A sample MariaDB `linstor.toml` looks like this:

```
[db]
user = "linstor"
password = "linstor"
connection_url = "jdbc:mariadb://localhost/LINSTOR?createDatabaseIfNotExist=true"
```

The LINSTOR schema/database is created as `LINSTOR` so make sure the mariadb connection string refers to the `LINSTOR` schema, as in the example above.

5.13. LINSTOR REST-API

To make LINSTOR's administrative tasks more accessable and also available for web-frontends a REST-API has been created. The REST-API is embedded in the linstor-controller and since LINSTOR 0.9.13 configured via the the `linstor.toml` configuration file.

```
[http]
  enabled = true
  port = 3370
  listen_addr = 127.0.0.1  # to disable remote access
```

If you want to use the REST-API the current documentation can be found on the following link: https://app.swaggerhub.com/apis-docs/Linstor/Linstor/

5.13.1. LINSTOR REST-API HTTPS

The HTTP REST-API can also run secured by HTTPS and is highly recommended if you use any features that require authorization. Todo so you have to create a java keystore file with a valid certificate that will be used to encrypt all HTTPS traffic.

Here is a simple example on how you can create a self signed certificate with the `keytool` that is included in the java runtime:

```
keytool -keyalg rsa -keysize 2048 -genkey -keystore ./keystore_linstor.jks\
 -alias linstor_controller\
 -dname "CN=localhost, OU=SecureUnit, O=ExampleOrg, L=Vienna, ST=Austria, C=AT"
```

`keytool` will ask for a password to secure the generated keystore file and is needed for the LINSTOR-controller configuration. In your `linstor.toml` file you have to add the following section:

```
[http]
  keystore = "/path/to/keystore_linstor.jks"
  keystore_password = "linstor"
```

Now (re)start the `linstor-controller` and the HTTPS REST-API should be available on port 3371.

More information on how to import other certificates can be found here: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/keytool.html

When HTTPS is enabled, all requests to the HTTP /v1/ REST-API will be redirected to the HTTPS redirect.

5.14. Getting help

5.14.1. From the command line

A quick way to list available commands on the command line is to type `linstor`.

Further information on subcommands (e.g., list-nodes) can be retrieved in two ways:

```
# linstor node list -h
# linstor help node list
```

Using the 'help' subcommand is especially helpful when LINSTOR is executed in interactive mode (`linstor interactive`).

One of the most helpful features of LINSTOR is its rich tab-completion, which can be used to complete basically every object LINSTOR knows about (e.g., node names, IP addresses, resource names, ...). In the following examples, we show some possible completions, and their results:

```
# linstor node create alpha 1<tab> # completes the IP address if hostname can be resolved
# linstor resource create b<tab> c<tab> # linstor assign-resource backups charlie
```

If tab-completion does not work out of the box, please try to source the appropriate file:

```
# source /etc/bash_completion.d/linstor # or
# source /usr/share/bash_completion/completions/linstor
```

For zsh shell users linstor-client can generate a zsh compilation file, that has basic support for command and argument completion.

```
# linstor gen-zsh-completer > /usr/share/zsh/functions/Completion/Linux/_linstor
```

5.14.2. From the community

For help from the community please subscribe to our mailing list located here: http://lists.linbit.com/listinfo/drbd-user

5.14.3. GitHub

To file bug or feature request please check out our GitHub page https://github.com/linbit

5.14.4. Paid support and development

Alternatively, if you wish to purchase remote installation services, 24/7 support, access to certified repositories, or feature development please contact us: +1-877-454-6248 (1-877-4LINBIT) , International: +43-1-8178292-0 | sales@linbit.com

Chapter 6. LINSTOR Volumes in Kubernetes

This chapter describes LINSTOR volumes in Kubernetes as managed by the LINSTOR CSI plugin

6.1. Kubernetes Overview

Kubernetes is a container orchestrator (CO) made by Google. Kubernetes defines the behavior of containers and related services via declarative specifications. In this guide, we'll focus on on using `kubectl` to manipulate `.yaml` files that define the specifications of Kubernetes objects.

6.2. LINSTOR CSI Plugin Deployment

Instructions for deploying the CSI plugin can be found on the project's github. This will result in a linstor-csi-controller *StatefulSet* and a linstor-csi-node *DaemonSet* running in the kube-system namespace.

```
NAME                        READY   STATUS    RESTARTS   AGE     IP              NODE
linstor-csi-controller-0    5/5     Running   0          3h10m   191.168.1.200   kubelet-a
linstor-csi-node-4fcnn      2/2     Running   0          3h10m   192.168.1.202   kubelet-c
linstor-csi-node-f2dr7      2/2     Running   0          3h10m   192.168.1.203   kubelet-d
linstor-csi-node-j66bc      2/2     Running   0          3h10m   192.168.1.201   kubelet-b
linstor-csi-node-qb7fw      2/2     Running   0          3h10m   192.168.1.200   kubelet-a
linstor-csi-node-zr75z      2/2     Running   0          3h10m   192.168.1.204   kubelet-e
```

6.3. Basic Configuration and Deployment

Once all linstor-csi *Pod*s are up and running, we can provision volumes using the usual Kubernetes workflows.

Configuring the behavior and properties of LINSTOR volumes deployed via Kubernetes is accomplished via the use of *StorageClass*es. Here below is the simplest practical *StorageClass* that can be used to deploy volumes:

Listing 1. linstor-basic-sc.yaml

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  # The name used to identify this StorageClass.
  name: linstor-basic-storage-class
  # The name used to match this StorageClass with a provisioner.
  # linstor.csi.linbit.com is the name that the LINSTOR CSI plugin uses to identify itself
provisioner: linstor.csi.linbit.com
parameters:
  # LINSTOR will provision volumes from the drbdpool storage pool configured
  # On the satellite nodes in the LINSTOR cluster specified in the plugin's deployment
  storagePool: "drbdpool"
```

We can create the *StorageClass* with the following command:

```
kubectl create -f linstor-basic-sc.yaml
```

Now that our *StorageClass* is created, we can now create a *PersistentVolumeClaim* which can be used to provision volumes known both to Kubernetes and LINSTOR:

Listing 2. my-first-linstor-volume-pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-first-linstor-volume
  annotations:
    # This line matches the PersistentVolumeClaim with our StorageClass
    # and therefore our provisioner.
    volume.beta.kubernetes.io/storage-class: linstor-basic-storage-class
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

We can create the *PersistentVolumeClaim* with the following command:

```
kubectl create -f my-first-linstor-volume-pvc.yaml
```

This will create a *PersistentVolumeClaim* known to Kubernetes, which will have a *PersistentVolume* bound to it, additionally LINSTOR will now create this volume according to the configuration defined in the `linstor-basic-storage-class` *StorageClass*. The LINSTOR volume's name will be a UUID prefixed with `csi-` This volume can be observed with the usual `linstor resource list`. Once that volume is created, we can attach it to a pod. The following *Pod* spec will spawn a Fedora container with our volume attached that busy waits so it is not unscheduled before we can interact with it:

Listing 3. my-first-linstor-volume-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: fedora
  namespace: default
spec:
  containers:
  - name: fedora
    image: fedora
    command: [/bin/bash]
    args: ["-c", "while true; do sleep 10; done"]
    volumeMounts:
    - name: my-first-linstor-volume
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: my-first-linstor-volume
    persistentVolumeClaim:
      claimName: "my-first-linstor-volume"
```

We can create the *Pod* with the following command:

```
kubectl create -f my-first-linstor-volume-pod.yaml
```

Running `kubectl describe pod fedora` can be used to confirm that *Pod* scheduling and volume attachment succeeded.

To remove a volume, please ensure that no pod is using it and then delete the *PersistentVolumeClaim* via `kubectl`. For example, to remove the volume that we just made, run the following two commands, noting that the *Pod* must be unscheduled before the *PersistentVolumeClaim* will be removed:

```
kubectl delete pod fedora # unschedule the pod.

kubectl get pod -w # wait for pod to be unscheduled

kubectl delete pvc my-first-linstor-volume # remove the PersistentVolumeClaim, the
PersistentVolume, and the LINSTOR Volume.
```

6.4. Snapshots

Creating snapshots and creating new volumes from snapshots is done via the use of *VolumeSnapshot*s, *VolumeSnapshotClass*es, and *PVC*s. First, you'll need to create a *VolumeSnapshotClass*:

Listing 4. my-first-linstor-snapshot-class.yaml

```
kind: VolumeSnapshotClass
apiVersion: snapshot.storage.k8s.io/v1alpha1
metadata:
  name: my-first-linstor-snapshot-class
  namespace: kube-system
snapshotter: io.drbd.linstor-csi
```

Create the *VolumeSnapshotClass* with `kubectl`:

```
kubectl create -f my-first-linstor-snapshot-class.yaml
```

Now we will create a volume snapshot for the volume that we created above. This is done with a *VolumeSnapshot*:

Listing 5. my-first-linstor-snapshot.yaml

```
apiVersion: snapshot.storage.k8s.io/v1alpha1
kind: VolumeSnapshot
metadata:
  name: my-first-linstor-snapshot
spec:
  snapshotClassName: my-first-linstor-snapshot-class
  source:
    name: my-first-linstor-volume
    kind: PersistentVolumeClaim
```

Create the *VolumeSnapshot* with `kubectl`:

```
kubectl create -f my-first-linstor-snapshot.yaml
```

Finally, we'll create a new volume from the snapshot with a *PVC*.

Listing 6. my-first-linstor-volume-from-snapshot.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-first-linstor-volume-from-snapshot
spec:
  storageClassName: linstor-basic-storage-class
  dataSource:
    name: my-first-linstor-snapshot
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

Create the *PVC* with `kubectl`:

```
kubectl create -f my-first-linstor-volume-from-snapshot.yaml
```

6.5. Volume Accessibility

LINSTOR volumes are typically accessible both locally and over the network.

By default, the CSI plugin will attach volumes directly if the *Pod* happens to be scheduled on a *kubelet* where its underlying storage is present. However, *Pod* scheduling does not currently take volume locality into account. The replicasOnSame parameter can be used to restrict where the underlying storage may be provisioned, if locally attached volumes are desired.

See localStoragePolicy to see how this default behavior can be modified.

6.6. Advanced Configuration

In general, all configuration for LINSTOR volumes in Kubernetes should be done via the *StorageClass* parameters, as seen with the *storagePool* in the basic example above. We'll give all the available options an in-depth treatment here.

6.6.1. nodeList

`nodeList` is a list of nodes for volumes to be assigned to. This will assign the volume to each node and it will be replicated among all of them. This can also be used to select a single node by hostname, but it's more flexible to use replicasOnSame to select a single node.

If you use this option, you must not use autoPlace.

This option determines on which LINSTOR nodes the underlying storage for volumes will be provisioned and is orthogonal from which *kubelets* these volumes will be accessible.

Example: `nodeList: "node-a node-b node-c"`

Example: `nodeList: "node-a"`

6.6.2. autoPlace

`autoPlace` is an integer that determines the amount of replicas a volume of this *StorageClass* will have. For instance, `autoPlace: 3` will produce volumes with three-way replication. If neither `autoPlace` nor `nodeList` are set, volumes will be automatically placed on one node.

If you use this option, you must not use nodeList.

This option (and all options which affect autoplacement behavior) modifies the number of LINSTOR nodes on which the underlying storage for volumes will be provisioned and is orthogonal to which *kubelets* those volumes will be accessible from.

Example: `autoPlace: 2`

Default: `autoPlace: 1`

6.6.3. replicasOnSame

`replicasOnSame` is a list of key=value pairs used as required autoplacement selection labels when autoplace is used to determine where to provision storage. These labels correspond to LINSTOR node aux props. Please note both the key and value names are user-defined and arbitrary. Let's explore this behavior with examples assuming a LINSTOR cluster such that `node-a` is configured with the following aux props `zone=z1` and `role=backups`, while `node-b` is configured with only `zone=z1`.

If we configure a *StorageClass* with `autoPlace: "1"` and `replicasOnSame: "zone=z1 role=backups"`, then all volumes created from that *StorageClass* will be provisioned on `node-a`, since that is the only node with all of the correct key=value pairs in the LINSTOR cluster. This is the most flexible way to select a single node for provisioning.

If we configure a *StorageClass* with `autoPlace: "1"` and `replicasOnSame: "zone=z1"`, then volumes will be provisioned on either `node-a` or `node-b` as they both have the `zone=z1` aux prop.

If we configure a *StorageClass* with `autoPlace: "2"` and `replicasOnSame: "zone=z1 role=backups"`, then provisioning will fail, as there are not two or more nodes that have the appropriate aux props.

If we configure a *StorageClass* with `autoPlace: "2"` and `replicasOnSame: "zone=z1"`, then volumes will be provisioned on both `node-a` and `node-b` as they both have the `zone=z1` aux prop.

Example: `replicasOnSame: "zone=z1 role=backups"`

6.6.4. replicasOnDifferent

`replicasOnDifferent` is a list of key=value pairs to avoid as autoplacement selection. It is the inverse of replicasOnSame.

Example: `replicasOnDifferent: "no-csi-volumes=true"`

6.6.5. localStoragePolicy

`localStoragePolicy` determines, via volume topology, which LINSTOR *Satellite*s volumes should be assigned and from where Kubernetes will access volumes. The behavior of each option is explained below in detail.

If you specify a nodeList, volumes will be created on those nodes, irrespective of the `localStoragePolicy`; however, the accessibility reporting will still be as described.

You must set `volumeBindingMode: WaitForFirstConsumer` in the *StorageClass* and the LINSTOR *Satellites* running on the *kubelets* must be able to support the diskfull placement of volumes as they are configured in that *StorageClass* for required or preferred to work properly.

Use `topologyKey: "linbit.com/hostname"` rather than `topologyKey: "kubernetes.io/hostname"` if you are setting `affinity` in your *Pod* or *StatefulSet* specs.

Example: `localStoragePolicy: required`

ignore (default)

When `localStoragePolicy` is set to `ignore`, regular autoplacement occurs based on autoplace, replicasOnSame, and replicasOnDifferent. Volume location will not affect *Pod* scheduling in Kubernetes and the volumes will be accessed over the network if they're not local to the *kubelet* where the *Pod* was scheduled.

required

When `localStoragePolicy` is set to `required`, Kubernetes will report a list of places that it wants to schedule a *Pod* in order of preference. The plugin will attempt to provision the volume(s) according to that preference. The number of volumes to be provisioned in total is based off of autoplace.

If all preferences have been attempted, but no volumes where successfully assigned volume creation will fail.

In case of multiple replicas when all preferences have been attempted, and at least one has succeeded, but there are still replicas remaining to be provisioned, autoplace behavior will apply for the remaining volumes.

With this option set, Kubernetes will consider volumes that are not locally present on a *kubelet* to be unaccessible from that *kubelet*.

preferred

When `localStoragePolicy` is set to `preferred`, volume placement behavior will be the same as when it's set to required with the exception that volume creation will not fail if no preference was able to be satisfied. Volume accessibility will be the same as when set to ignore.

6.6.6. storagePool

`storagePool` is the name of the LINSTOR storage pool that will be used to provide storage to the newly-created volumes.

Only nodes configured with this same *storage pool* with be considered for autoplacement. Likewise, for *StorageClasses* using nodeList all nodes specified in that list must have this *storage pool* configured on them.

Example: `storagePool: my-storage-pool`

6.6.7. disklessStoragePool

`disklessStoragePool` is an optional parameter that only effects LINSTOR volumes assigned disklessly to *kubelets* i.e., as clients. If you have a custom *diskless storage pool* defined in LINSTOR, you'll specify that here.

Example: `disklessStoragePool: my-custom-diskless-pool`

6.6.8. encryption

`encryption` is an optional parameter that determines whether to encrypt volumes. LINSTOR must be configured for

encryption for this to work properly.

Example: `encryption: "true"`

6.6.9. filesystem

`filesystem` is an option parameter to specify the filesystem for non raw block volumes. Currently supported options are `xfs` and `ext4`.

Example: `filesystem: "xfs"`

Default: `filesystem: "ext4"`

6.6.10. fsOpts

`fsOpts` is an optional parameter that passes options to the volume's filesystem at creation time.

Please note these values are specific to your chosen filesystem.

Example: `fsOpts: "-b 2048"`

6.6.11. mountOpts

`mountOpts` is an optional parameter that passes options to the volume's filesystem at mount time.

Example: `mountOpts: "sync,noatime"`

Chapter 7. LINSTOR Volumes in Proxmox VE

This chapter describes DRBD in Proxmox VE via the LINSTOR Proxmox Plugin.

7.1. Proxmox VE Overview

Proxmox VE is an easy to use, complete server virtualization environment with KVM, Linux Containers and HA.

'linstor-proxmox' is a Perl plugin for Proxmox that, in combination with LINSTOR, allows to replicate VM disks on several Proxmox VE nodes. This allows to live-migrate active VMs within a few seconds and with no downtime without needing a central SAN, as the data is already replicated to multiple nodes.

7.2. Proxmox Plugin Installation

LINBIT provides a dedicated public repository for Proxmox VE users. This repository not only contains the Proxmox plugin, but the whole DRBD SDS stack including a DRBD SDS kernel module and user space utilities.

The DRBD9 kernel module is installed as a `dkms` package (i.e., `drbd-dkms`), therefore you'll have to install `pve-headers` package, before you set up/install the software packages from LINBIT's repositories. Following that order, ensures that the kernel module will build properly for your kernel. If you don't plan to install the latest Proxmox kernel, you have to install kernel headers matching your current running kernel (e.g., `pve-headers-$(uname -r)`). If you missed this step, then still you can rebuild the dkms package against your current kernel, (kernel headers have to be installed in advance), by issuing `apt install --reinstall drbd-dkms` command.

LINBIT's repository can be enabled as follows, where "$PVERS" should be set to your Proxmox VE **major version** (e.g., "5", not "5.2"):

```
# wget -O- https://packages.linbit.com/package-signing-pubkey.asc | apt-key add -
# PVERS=5 && echo "deb http://packages.linbit.com/proxmox/ proxmox-$PVERS drbd-9.0" > \
    /etc/apt/sources.list.d/linbit.list
# apt update && apt install linstor-proxmox
```

7.3. LINSTOR Configuration

For the rest of this guide we assume that you have a LINSTOR cluster configured as described in Initializing your cluster. In the most simple case you will have one storage pool definition (e.g., "drbdpool") and the equivalent storage pools on each PVE node. If you have multiple pools, currently the plugin selects one of those, but as of now you can not control which one. But usually you have only one pool for your DRBD resources, so that shouldn't be a problem. Also make sure to setup each node as a "Combined" node. Start the "linstor-controller" on one node, and the "linstor-satellite" on all nodes.

7.4. Proxmox Plugin Configuration

The final step is to provide a configuration for Proxmox itself. This can be done by adding an entry in the `/etc/pve/storage.cfg` file, with the following content, assuming a three node cluster in this example:

```
drbd: drbdstorage
   content images,rootdir
   redundancy 3
   controller 10.11.12.13
```

The "drbd" entry is fixed and you are not allowed to modify it, as it tells to Proxmox to use DRBD as storage backend. The "drbdstorage" entry can be modified and is used as a friendly name that will be shown in the PVE web GUI to locate the DRBD storage. The "content" entry is also fixed, so do not change it. The "redundancy" parameter specifies how

many replicas of the data will be stored in the cluster. The recommendation is to set it to "3", assuming that you have a three node cluster as a minimum. The data is accessible from all nodes, even if some of them do not have local copies of the data. For example, in a 5 node cluster, all nodes will be able to access 3 copies of the data, no matter where they are stored in. The "controller" parameter must be set to the IP of the node that runs the LINSTOR controller service. Only one node can be set to run as LINSTOR controller at the same time. If that node fails, start the LINSTOR controller on another node and change that value to its IP address. There are more elegant ways to deal with this problem. For more, see later in this chapter how to setup a highly available LINSTOR controller VM in Proxmox.

Recent versions of the plugin allow to define multiple different storage pools. Such a configuration would look like this, where "storagepool" is set to the name of a LINSTOR storage pool definition:

```
drbd: drbdstorage
   content images,rootdir
   redundancy 3
   # Should be set, see below
   # storagepool drbdpool
   controller 10.11.12.13

drbd: fastdrbd
   content images,rootdir
   redundancy 3
   storagepool ssd
   controller 10.11.12.13

drbd: slowdrbd
   content images,rootdir
   redundancy 2
   storagepool rotatingrust
   controller 10.11.12.13
```

Note that if you do not set a "storagepool", as it is the case for "drbdstorage", one will be selected by an internal metric. We suggest that you either have one pool where it is optional to set "storagepool", or you explicitly set the storage pool name for all entries.

By now, you should be able to create VMs via Proxmox's web GUI by selecting "*drbdstorage*", or any other of the defined pools as storage location.

NOTE: DRBD supports only the **raw** *disk format at the moment.*

At this point you can try to live migrate the VM - as all data is accessible on all nodes (even on Diskless nodes) - it will take just a few seconds. The overall process might take a bit longer if the VM is under load and if there is a lot of RAM being dirtied all the time. But in any case, the downtime should be minimal and you will see no interruption at all.

7.5. Making the Controller Highly-Available

For the rest of this guide we assume that you installed LINSTOR and the Proxmox Plugin as described in LINSTOR Configuration.

The basic idea is to execute the LINSTOR controller within a VM that is controlled by Proxmox and its HA features, where the storage resides on DRBD managed by LINSTOR itself.

The first step is to allocate storage for the VM: Create a VM as usual and select "Do not use any media" on the "OS" section. The hard disk should of course reside on DRBD (e.g., "drbdstorage"). 2GB disk space should be enough, and for RAM we chose 1GB. These are the minimum requirements for the appliance LINBIT provides to its customers (see below). If you wish to set up your own controller VM, and you have enough hardware resources available, you can increase these minimum values. In the following use case, we assume that the controller VM was created with ID 100, but it is fine if this VM was created at a later time and has a different ID.

LINBIT provides an appliance for its customers that can be used to populate the created storage. For the appliance to

work, we first create a "Serial Port". First click on "Hardware" and then on "Add" and finally on "Serial Port":

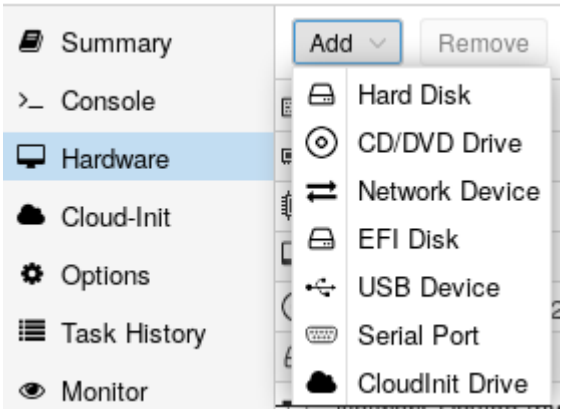


Figure 5. Adding a Serial Port

If everything worked as expected the VM definition should then look like this:

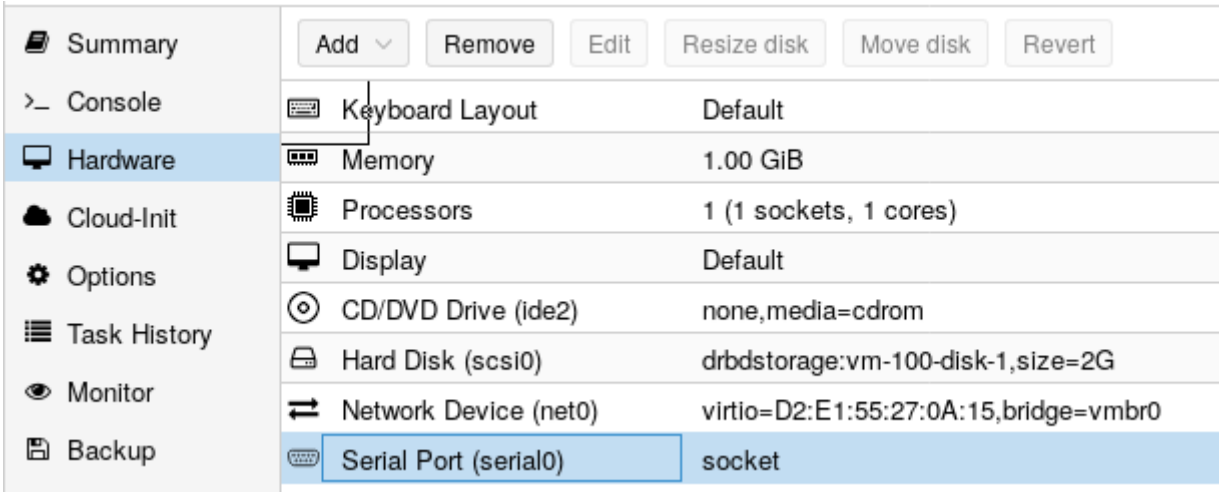


Figure 6. VM with Serial Port

The next step is to copy the VM appliance to the VM disk storage. This can be done with `qemu-img`.

Make sure to replace the VM ID with the correct one.

```
# qemu-img dd -O raw if=/tmp/linbit-linstor-controller-amd64.img \
  of=/dev/drbd/by-res/vm-100-disk-1/0
```

Once completed you can start the VM and connect to it via the Proxmox VNC viewer. The default user name and password are both "linbit". Note that we kept the default configuration for the ssh server, so you will not be able to log in to the VM via ssh and username/password. If you want to enable that (and/or "root" login), enable these settings in `/etc/ssh/sshd_config` and restart the ssh service. As this VM is based on "Ubuntu Bionic", you should change your network settings (e.g., static IP) in `/etc/netplan/config.yaml`. After that you should be able to ssh to the VM:

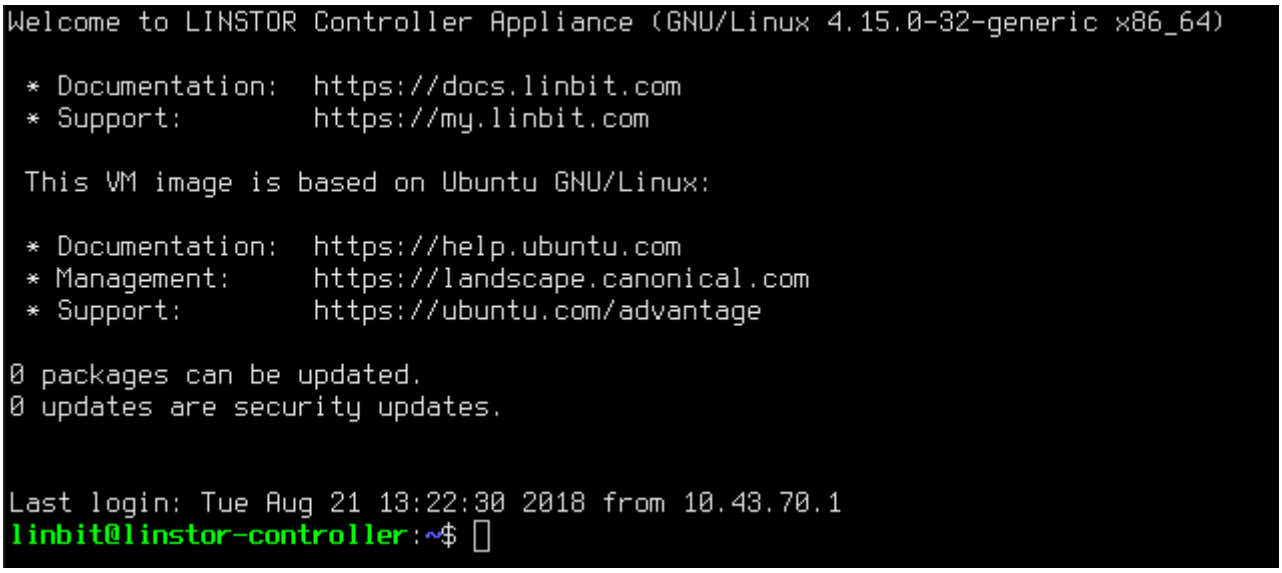


Figure 7. LINBIT LINSTOR Controller Appliance

In the next step you add the controller VM to the existing cluster:

```
# linstor node create --node-type Controller \
  linstor-controller 10.43.7.254
```

As the Controller VM will be handled in a special way by the Proxmox storage plugin (comparing to the rest of VMs), we must make sure all hosts have access to its backing storage, **before** PVE HA starts the VM, otherwise the VM will fail to start. See below for the details on how to achieve this.

In our test cluster the Controller VM disk was created in DRBD storage and it was initially assigned to one host (use `linstor resource list` to check the assignments). Then, we used `linstor resource create` command to create additional resource assignments to the other nodes of the cluster for this VM. In our lab consisting of four nodes, we created all resource assignments as diskful, but diskless assignments are fine as well. As a rule of thumb keep the redundancy count at "3" (more usually does not make sense), and assign the rest as diskless.

As the storage for the Controller VM must be made available on all PVE hosts in some way, we must make sure to enable the `drbd.service` on all hosts (given that it is not controlled by LINSTOR at this stage):

```
# systemctl enable drbd
# systemctl start drbd
```

At startup, the `linstor-satellite` service deletes all of its resource files (`.res`) and regenerates them. This conflicts with the drbd services that needs these resource files to start the controller VM. It is good enough to first bring up the resources via drbd.service and then start linstor-satellite.service. To make the necessary changes, you need to create a drop-in for the linstor-satellite.service via systemctl (do *not* edit the file directly).

```
systemctl edit linstor-satellite
[Unit]
After=drbd.service
```

Don't forget to restart the `linstor-satellite.service`.

After that, it is time for the final steps, namely switching from the existing controller (residing on the physical host) to

the new one in the VM. So let's stop the old controller service on the physical host, and copy the LINSTOR controller database to the VM host:

```
# systemctl stop linstor-controller
# systemctl disable linstor-controller
# scp /var/lib/linstor/* root@10.43.7.254:/var/lib/linstor/
```

Finally, we can enable the controller in the VM:

```
# systemctl start linstor-controller # in the VM
# systemctl enable linstor-controller # in the VM
```

To check if everything worked as expected, you can query the cluster nodes on a physical PVE host by asking the controller in the VM: `linstor --controllers=10.43.7.254 node list`. It is perfectly fine that the controller (which is just a Controller and not a "Combined" host) is shown as "OFFLINE". This might change in the future to something more reasonable.

As the last — but crucial — step, you need to add the "controlervm" option to `/etc/pve/storage.cfg`, and change the controller IP address to the IP address of the Controller VM:

```
drbd: drbdstorage
   content images,rootdir
   redundancy 3
   controller 10.43.7.254
   controllervm 100
```

Please note the additional setting "controllervm". This setting is very important, as it tells to PVE to handle the Controller VM differently than the rest of VMs stored in the DRBD storage. In specific, it will instruct PVE to NOT use LINSTOR storage plugin for handling the Controller VM, but to use other methods instead. The reason for this, is that simply LINSTOR backend is not available at this stage. Once the Controller VM is up and running (and the associated LINSTOR controller service inside the VM), then the PVE hosts will be able to start the rest of virtual machines which are stored in the DRBD storage by using LINSTOR storage plugin. Please make sure to set the correct VM ID in the "controllervm" setting. In this case is set to "100", which represents the ID assigned to our Controller VM.

It is very important to make sure that the Controller VM is up and running at all times and that you are backing it up at regular times(mostly when you do modifications to the LINSTOR cluster). Once the VM is gone, and there are no backups, the LINSTOR cluster must be recreated from scratch.

To prevent accidental deletion of the VM, you can go to the "Options" tab of the VM, in the PVE GUI and enable the "Protection" option. If however you accidentally deleted the VM, such requests are ignored by our storage plugin, so the VM disk will NOT be deleted from the LINSTOR cluster. Therefore, it is possible to recreate the VM with the same ID as before(simply recreate the VM configuration file in PVE and assign the same DRBD storage device used by the old VM). The plugin will just return "OK", and the old VM with the old data can be used again. In general, be careful to not delete the controller VM and "protect" it accordingly.

Currently, we have the controller executed as VM, but we should make sure that one instance of the VM is started at all times. For that we use Proxmox's HA feature. Click on the VM, then on "More", and then on "Manage HA". We set the following parameters for our controller VM:

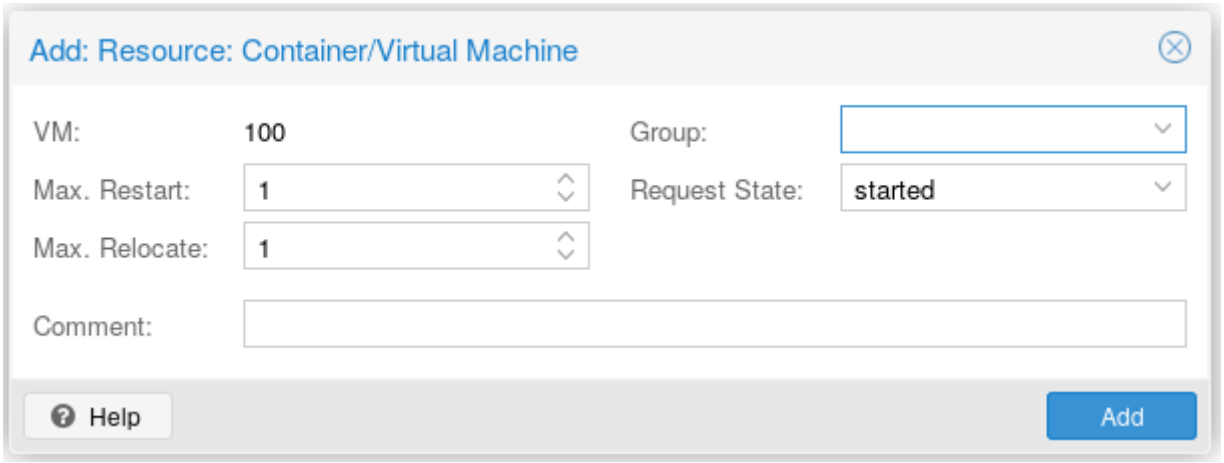


Figure 8. HA settings for the controller VM

As long as there are surviving nodes in your Proxmox cluster, everything should be fine and in case the node hosting the Controller VM is shut down or lost, Proxmox HA will make sure the controller is started on another host. Obviously the IP of the controller VM should not change. It is up to you as an administrator to make sure this is the case (e.g., setting a static IP, or always providing the same IP via dhcp on the bridged interface).

It is important to mention at this point that in the case that you are using a dedicated network for the LINSTOR cluster, you must make sure that the network interfaces configured for the cluster traffic, are configured as bridges (i.e vmb1,vmbr2 etc) on the PVE hosts. If they are setup as direct interfaces (i.e eth0,eth1 etc), then you will not be able to setup the Controller VM vNIC to communicate with the rest of LINSTOR nodes in the cluster, as you cannot assign direct network interfaces to the VM, but only bridged interfaces.

One limitation that is not fully handled with this setup is a total cluster outage (e.g., common power supply failure) with a restart of all cluster nodes. Proxmox is unfortunately pretty limited in this regard. You can enable the "HA Feature" for a VM, and you can define "Start and Shutdown Order" constraints. But both are completely separated from each other. Therefore it is hard/impossible to guarantee that the Controller VM will be up and running, before all other VMs are started.

It might be possible to work around that by delaying VM startup in the Proxmox plugin itself until the controller VM is up (i.e., if the plugin is asked to start the controller VM it does it, otherwise it waits and pings the controller). While a nice idea, this would horribly fail in a serialized, non-concurrent VM start/plugin call event stream where some VM should be started (which then are blocked) before the Controller VM is scheduled to be started. That would obviously result in a deadlock.

We will discuss these options with Proxmox, but we think the current solution is valuable in most typical use cases, as is. Especially, compared to the complexity of a pacemaker setup. Use cases where one can expect that not the whole cluster goes down at the same time are covered. And even if that is the case, only automatic startup of the VMs would not work when the whole cluster is started. In such a scenario the admin just has to wait until the Proxmox HA service starts the controller VM. After that all VMs can be started manually/scripted on the command line.

Chapter 8. LINSTOR Volumes in OpenNebula

This chapter describes DRBD in OpenNebula via the usage of the LINSTOR storage driver addon.

Detailed installation and configuration instructions and be found in the README.md file of the driver's source.

8.1. OpenNebula Overview

OpenNebula is a flexible and open source cloud management platform which allows its functionality to be extended via the use of addons.

The LINSTOR addon allows the deployment of virtual machines with highly available images backed by DRBD and attached across the network via DRBD's own transport protocol.

8.2. OpenNebula addon Installation

Installation of the LINSTOR storage addon for OpenNebula requires a working OpenNebula cluster as well as a working LINSTOR cluster.

With access to LINBIT's customer repositories you can install the linstor-opennebula with

```
# apt install linstor-opennebula
```

or

```
# yum install linstor-opennebula
```

Without access to LINBIT's prepared packages you need to fall back to instructions on it's GitHub page.

A DRBD cluster with LINSTOR can be installed and configured by following the instructions in this guide, see Initializing your cluster.

The OpenNebula and DRBD clusters can be somewhat independent of one another with the following exception: OpenNebula's Front-End and Host nodes must be included in both clusters.

Host nodes do not need a local LINSTOR storage pools, as virtual machine images are attached to them across the network [3: If a host is also a storage node, it will use a local copy of an image if that is available].

8.3. Deployment Options

Resources may be automatically placed or assigned to specific nodes. Generally, automatic assignment is preferred as LINSTOR will automatically scale as more nodes are added to the cluster, provided they have the storage pool that matches the one that the driver is configured to use.

The driver also allows for the selection of storage pools on a per-datastore basis. If no storage pool is selected, the default storage pool will be used.

8.4. Live Migration

Live migration is supported even with the use of the ssh system datastore, as well as the nfs shared system datastore.

8.5. Free Space Reporting

Free space is calculated differently depending on whether resources are deployed automatically or on a per node basis.

For datastores which place per node, free space is reported based on the most restrictive storage pools from all nodes where resources are being deployed. For example, the capacity of the node with the smallest amount of total storage space is used to determine the total size of the datastore and the node with the least free space is used to determine the remaining space in the datastore.

For a datastore which uses automatic placement, size and remaining space are determined based on the aggregate storage pool used by the datastore as reported by LINSTOR.

Chapter 9. LINSTOR volumes in Openstack

This chapter describes DRBD in Openstack for persistent, replicated, and high-performance block storage with LINSTOR Driver.

9.1. Openstack Overview

Openstack consists of a wide range of individual services; the two that are mostly relevant to DRBD are Cinder and Nova. **Cinder** is the block storage service, while **Nova** is the compute node service that's responsible for making the volumes available for the VMs.

The LINSTOR driver for OpenStack manages DRBD/LINSTOR clusters and makes them available within the OpenStack environment, especially within Nova compute instances. LINSTOR-backed Cinder volumes will seamlessly provide all the features of DRBD/LINSTOR while allowing OpenStack to manage all their deployment and management. The driver will allow OpenStack to create and delete persistent LINSTOR volumes as well as managing and deploying volume snapshots and raw volume images.

Aside from using the kernel-native DRBD protocols for replication, the LINSTOR driver also allows using iSCSI with LINSTOR cluster(s) to provide maximum compatibility. For more information on these two options, please see Choosing the Transport Protocol.

9.2. LINSTOR for Openstack Installation

An initial installation and configuration of DRBD and LINSTOR must be completed prior to installing OpenStack driver. Each LINSTOR node in a cluster should also have a Storage Pool defined as well. Details about LINSTOR installation can be found here.

9.2.1. Here's a synopsis on quickly setting up a LINSTOR cluster on Ubuntu:

Install DRBD and LINSTOR on Cinder node as a LINSTOR Controller node:

```
# First, set up LINBIT repository per support contract

# Install DRBD and LINSTOR packages
sudo apt update
sudo apt install -y drbd-dkms lvm2
sudo apt install -y linstor-controller linstor-satellite linstor-client
sudo apt install -y drbdtop

# Start both LINSTOR Controller and Satellite Services
systemctl enable linstor-controller.service
systemctl start linstor-controller.service
systemctl enable linstor-satellite.service
systemctl start linstor-satellite.service

# For Diskless Controller, skip the following two 'sudo' commands

# For Diskful Controller, create backend storage for DRBD/LINSTOR by creating
# a Volume Group 'drbdpool' and specify appropriate volume location (/dev/vdb)
sudo vgcreate drbdpool /dev/vdb

# Create a Logical Volume 'thinpool' within 'drbdpool'
# Specify appropriate thin volume size (64G)
sudo lvcreate -L 64G -T drbdpool/thinpool
```

OpenStack measures storage size in GiBs.

Install DRBD and LINSTOR on other node(s) on the LINSTOR cluster:

```
# First, set up LINBIT repository per support contract

# Install DRBD and LINSTOR packages
sudo apt update
sudo apt install -y drbd-dkms lvm2
sudo apt install -y linstor-satellite
sudo apt install -y drbdtop

# Start only the LINSTOR Satellite service
systemctl enable linstor-satellite.service
systemctl start linstor-satellite.service

# Create backend storage for DRBD/LINSTOR by creating a Volume Group 'drbdpool'
# Specify appropriate volume location (/dev/vdb)
sudo vgcreate drbdpool /dev/vdb

# Create a Logical Volume 'thinpool' within 'drbdpool'
# Specify appropriate thin volume size (64G)
sudo lvcreate -L 64G -T drbdpool/thinpool
```

Lastly, from the Cinder node, create LINSTOR Satellite Node(s) and Storage Pool(s)

```
# Create a LINSTOR cluster, including the Cinder node as one of the nodes
# For each node, specify node name, its IP address, volume type (diskless) and
# volume location (drbdpool/thinpool)

# Create the controller node as combined controller and satellite node
linstor node create cinder-node-name 192.168.1.100 --node-type Combined

# Create the satellite node(s)
linstor node create another-node-name 192.168.1.101
# repeat to add more satellite nodes in the LINSTOR cluster

# Create LINSTOR Storage Pool on each nodes
# For each node, specify node name, its IP address,
# storage pool name (DfltStorPool),
# volume type (diskless / lvmthin) and node type (Combined)

# Create diskless Controller node on the Cinder controller
linstor storage-pool create diskless cinder-node-name DfltStorPool

# Create diskful Satellite nodes
linstor storage-pool create lvmthin another-node-name DfltStorPool drbdpool/thinpool
# repeat to add a storage pool to each node in the LINSTOR cluster
```

9.2.2. Install the LINSTOR driver file

The *linstor driver* will be officially available starting OpenStack Stein release. The latest release is located at LINBIT OpenStack Repo. It is a single Python file called **linstordrv.py**. Depending on your OpenStack installation, its

destination may vary.

Place the driver (linstordrv.py) in an appropriate location within your OpenStack Cinder node.

For Devstack:

```
/opt/stack/cinder/cinder/volume/drivers/linstordrv.py
```

For Ubuntu:

```
/usr/lib/python2.7/dist-packages/cinder/volume/drivers/linstordrv.py
```

For RDO Packstack:

```
/usr/lib/python2.7/site-packages/cinder/volume/drivers/linstordrv.py
```

9.3. Cinder Configuration for LINSTOR

9.3.1. Edit Cinder configuration file cinder.conf in /etc/cinder/ as follows:

Enable LINSTOR driver by adding 'linstor' to enabled_backends

```
[DEFAULT]
...
enabled_backends=lvm, linstor
...
```

Add the following configuration options at the end of the cinder.conf

```
[linstor]
volume_backend_name = linstor
volume_driver = cinder.volume.drivers.linstordrv.LinstorDrbdDriver
linstor_default_volume_group_name=drbdpool
linstor_default_uri=linstor://localhost
linstor_default_storage_pool_name=DfltStorPool
linstor_default_resource_size=1
linstor_volume_downsize_factor=4096
```

9.3.2. Update Python python libraries for the driver

```
sudo pip install google --upgrade
sudo pip install protobuf --upgrade
sudo pip install eventlet --upgrade
```

9.3.3. Create a new backend type for LINSTOR

Run these commands from the Cinder node once environment variables are configured for OpenStack command line operation.

```
cinder type-create linstor
cinder type-key linstor set volume_backend_name=linstor
```

9.3.4. Restart the Cinder services to finalize

For Devstack:

```
sudo systemctl restart devstack@c-vol.service
sudo systemctl restart devstack@c-api.service
sudo systemctl restart devstack@c-sch.service
```

For RDO Packstack:

```
sudo systemctl restart openstack-cinder-volume.service
sudo systemctl restart openstack-cinder-api.service
sudo systemctl restart openstack-cinder-scheduler.service
```

For full OpenStack:

```
sudo systemctl restart cinder-volume.service
sudo systemctl restart cinder-api.service
sudo systemctl restart cinder-scheduler.service
```

9.3.5. Verify proper installation:

Once the Cinder service is restarted, a new Cinder volume with LINSTOR backend may be created using the Horizon GUI or command line. Use following as a guide for creating a volume with the command line.

```
# Check to see if there are any recurring errors with the driver.
# Occasional 'ERROR' keyword associated with the database is normal.
# Use Ctrl-C to stop the log output to move on.
sudo systemctl -f -u devstack@c-* | grep error

# Create a LINSTOR test volume.  Once the volume is created, volume list
# command should show one new Cinder volume.  The 'linstor' command then
# should list actual resource nodes within the LINSTOR cluster backing that
# Cinder volume.
openstack volume create --type linstor --size 1 --availability-zone nova linstor-test-vol
openstack volume list
linstor resource list
```

9.3.6. Additional Configuration

More to come

9.4. Choosing the Transport Protocol

There are two main ways to run DRBD/LINSTOR with Cinder:

- accessing storage via iSCSI exports, and
- using the DRBD transport protocol on the wire with LINSTOR.

These are not exclusive; you can define multiple backends, have some of them use iSCSI, and others the DRBD protocol.

9.4.1. iSCSI Transport

The default way to export Cinder volumes is via iSCSI. This brings the advantage of maximum compatibility - iSCSI can be used with every hypervisor, be it VMWare, Xen, HyperV, or KVM.

The drawback is that all data has to be sent to a Cinder node, to be processed by an (userspace) iSCSI daemon; that means that the data needs to pass the kernel/userspace border, and these transitions will cost some performance.

9.4.2. DRBD/LINSTOR Transport

The alternative is to get the data to the VMs by using DRBD as the transport protocol. This means that DRBD 9 [4: LINSTOR must be installed on Cinder node. Please see the note at [s-openstack-linstor-drbd-external-NOTE].] needs to be installed on the Cinder node as well.

Since OpenStack only functions in Linux, using DRBD/LINSTOR Transport restricts deployment only on Linux hosts with KVM at the moment.

One advantage of that solution is that the storage access requests of the VMs can be sent via the DRBD kernel module to the storage nodes, which can then directly access the allocated LVs; this means no Kernel/Userspace transitions on the data path, and consequently better performance. Combined with RDMA capable hardware you should get about the same performance as with VMs accessing a FC backend directly.

Another advantage is that you will be implicitly benefitting from the HA background of DRBD: using multiple storage nodes, possibly available over different network connections, means redundancy and avoiding a single point of failure.

Default configuration options for Cinder driver assumes the Cinder node to be a **Diskless** LINSTOR node. If the node is a **Diskful** node, please change the 'linstor_controller_diskless=True' to 'linstor_controller_diskless=False' and restart the Cinder services.

9.4.3. Configuring the Transport Protocol

In the LINSTOR section in `cinder.conf` you can define which transport protocol to use. The initial setup described at the beginning of this chapter is set to use DRBD transport. You can configure as necessary as shown below. Then Horizon [5: The OpenStack GUI] should offer these storage backends at volume creation time.

- To use iSCSI with LINSTOR:

```
volume_driver=cinder.volume.drivers.drbdmanagedrv.DrbdManageIscsiDriver
```

- To use DRBD Kernel Module with LINSTOR:

```
volume_driver=cinder.volume.drivers.drbdmanagedrv.DrbdManageDrbdDriver
```

The old class name "DrbdManageDriver" is being kept for the time because of compatibility reasons; it's just an alias to

the iSCSI driver.

To summarize:

- You'll need the LINSTOR Cinder driver 0.1.0 or later, and LINSTOR 0.6.5 or later.
- The DRBD transport protocol should be preferred whenever possible; iSCSI won't offer any locality benefits.
- Take care to not run out of disk space, especially with thin volumes.

Chapter 10. LINSTOR Volumes in Docker

This chapter describes LINSTOR volumes in Docker as managed by the LINSTOR Docker Volume Plugin.

10.1. Docker Overview

Docker is a platform for developing, shipping, and running applications in the form of Linux containers. For stateful applications that require data persistence, Docker supports the use of persistent *volumes* and *volume_drivers*.

The LINSTOR Docker Volume Plugin is a volume driver that provisions persistent volumes from a LINSTOR cluster for Docker containers.

10.2. LINSTOR Plugin for Docker Installation

To install the `linstor-docker-volume` package provided by LINBIT, you'll need to have LINBIT's `drbd-9.0` repository enabled. Once enabled, use your package manager to install the `linstor-docker-volume` package:

```
# yum install linstor-docker-volume

  -- OR --

# apt install linstor-docker-volume
```

Alternatively, you can build and install from source:

```
# git clone https://github.com/LINBIT/linstor-docker-volume
# cd ./linstor-docker-volume
# make
  ..snip..
# make doc
  ..snip..
# make install
  ..snip..
```

Once installed, *enable* and *start* the `linstor-docker-volume.socket` and `linstor-docker-volume.service` via `systemd`:

```
# systemctl enable linstor-docker-volume.socket linstor-docker-volume.service --now
```

10.3. LINSTOR Plugin for Docker Configuration

As the plugin has to communicate to the LINSTOR controller via the LINSTOR python library, we must tell the plugin where to find the LINSTOR Controller node in it's configuration file:

```
# cat /etc/linstor/docker-volume.conf
[global]
controllers = linstor://hostnameofcontroller
```

It is also possible to set all of the command line options found in `man linstor-docker-volume` in the `docker-volume.conf`. For example:

```
# cat /etc/linstor/docker-volume.conf
[global]
storagepool = thin-lvm
fs = ext4
fsopts = -E discard
size = 100MB
replicas = 2
nodes = alpha,bravo,charlie
```

If neither *replicas* or *nodes* is defined, the default of *replicas* will be used, which is two diskful replicas. If both *replicas* and *nodes* are defined, *nodes* will be used and *replicas* will be ignored.

10.4. Example Usage

The following are some examples of how you might use the LINSTOR Docker Volume Plugin. In the following we expect a cluster consisting of three nodes (alpha, bravo, and charlie).

10.4.1. Example 1 – typical docker pattern

On node alpha:

```
$ docker volume create -d linstor \
             --opt fs=xfs --opt size=200 lsvol
$ docker run -it --rm --name=cont \
             -v lsvol:/data --volume-driver=linstor busybox sh
$ root@cont: echo "foo" > /data/test.txt
$ root@cont: exit
```

On node bravo:

```
$ docker run -it --rm --name=cont \
             -v lsvol:/data --volume-driver=linstor busybox sh
$ root@cont: cat /data/test.txt
  foo
$ root@cont: exit
$ docker volume rm lsvol
```

10.4.2. Example 2 – one diskfull assignment by name, two nodes diskless

```
$ docker volume create -d linstor --opt hosts=bravo lsvol
```

10.4.3. Example 3 – one diskfull assignment, no matter where, two nodes diskless

```
$ docker volume create -d linstor --opt replicas=1 lsvol
```

10.4.4. Example 4 – two diskfull assignments by name, charlie diskless

```
$ docker volume create -d linstor --opt hosts=alpha,bravo lsvol
```

10.4.5. Example 5 – two diskfull assignments, no matter where, one node diskless

```
$ docker volume create -d linstor --opt replicas=2 lsvol
```

10.4.6. Example 6 – using curl without docker, useful for developers

```
$ curl --unix-socket /run/docker/plugins/linstor.sock \
          -X POST -H "Content-Type: application/json"
          -d '{"Name":"linstorvol"}' http://localhost/VolumeDriver.List
```

Working with DRBD

Chapter 11. Common administrative tasks

This chapter outlines typical administrative tasks encountered during day-to-day operations. It does not cover troubleshooting tasks, these are covered in detail in Troubleshooting and error recovery.

11.1. Configuring DRBD

11.1.1. Preparing your lower-level storage

After you have installed DRBD, you must set aside a roughly identically sized storage area on both cluster nodes. This will become the *lower-level device* for your DRBD resource. You may use any type of block device found on your system for this purpose. Typical examples include:

- A hard drive partition (or a full physical hard drive),
- a software RAID device,
- an LVM Logical Volume or any other block device configured by the Linux device-mapper infrastructure,
- any other block device type found on your system.

You may also use *resource stacking*, meaning you can use one DRBD device as a lower-level device for another. Some specific considerations apply to stacked resources; their configuration is covered in detail in Creating a stacked three-node setup.

While it is possible to use loop devices as lower-level devices for DRBD, doing so is not recommended due to deadlock issues.

It is *not* necessary for this storage area to be empty before you create a DRBD resource from it. In fact it is a common use case to create a two-node cluster from a previously non-redundant single-server system using DRBD (some caveats apply — please refer to DRBD meta data if you are planning to do this).

For the purposes of this guide, we assume a very simple setup:

- Both hosts have a free (currently unused) partition named `/dev/sda7`.
- We are using internal meta data.

11.1.2. Preparing your network configuration

It is recommended, though not strictly required, that you run your DRBD replication over a dedicated connection. At the time of this writing, the most reasonable choice for this is a direct, back-to-back, Gigabit Ethernet connection. When DRBD is run over switches, use of redundant components and the *bonding* driver (in *active-backup* mode) is recommended.

It is generally not recommended to run DRBD replication via routers, for reasons of fairly obvious performance drawbacks (adversely affecting both throughput and latency).

In terms of local firewall considerations, it is important to understand that DRBD (by convention) uses TCP ports from 7788 upwards, with every resource listening on a separate port. DRBD uses *two* TCP connections for every resource configured. For proper DRBD functionality, it is required that these connections are allowed by your firewall configuration.

Security considerations other than firewalling may also apply if a Mandatory Access Control (MAC) scheme such as SELinux or AppArmor is enabled. You may have to adjust your local security policy so it does not keep DRBD from functioning properly.

You must, of course, also ensure that the TCP ports for DRBD are not already used by another application.

It is not (yet) possible to configure a DRBD resource to support more than one TCP connection pair. If you want to

provide for DRBD connection load-balancing or redundancy, you can easily do so at the Ethernet level (again, using the *bonding* driver).

For the purposes of this guide, we assume a very simple setup:

- Our two DRBD hosts each have a currently unused network interface, `eth1`, with IP addresses `10.1.1.31` and `10.1.1.32` assigned to it, respectively.
- No other services are using TCP ports 7788 through 7799 on either host.
- The local firewall configuration allows both inbound and outbound TCP connections between the hosts over these ports.

11.1.3. Configuring your resource

All aspects of DRBD are controlled in its configuration file, `/etc/drbd.conf`. Normally, this configuration file is just a skeleton with the following contents:

```
include "/etc/drbd.d/global_common.conf";
include "/etc/drbd.d/*.res";
```

By convention, `/etc/drbd.d/global_common.conf` contains the `global` and `common` sections of the DRBD configuration, whereas the `.res` files contain one `resource` section each.

It is also possible to use `drbd.conf` as a flat configuration file without any `include` statements at all. Such a configuration, however, quickly becomes cluttered and hard to manage, which is why the multiple-file approach is the preferred one.

Regardless of which approach you employ, you should always make sure that `drbd.conf`, and any other files it includes, are *exactly identical* on all participating cluster nodes.

The DRBD source tarball contains an example configuration file in the `scripts` subdirectory. Binary installation packages will either install this example configuration directly in `/etc`, or in a package-specific documentation directory such as `/usr/share/doc/packages/drbd`.

This section describes only those few aspects of the configuration file which are absolutely necessary to understand in order to get DRBD up and running. The configuration file's syntax and contents are documented in great detail in the man page of `drbd.conf`.

Example configuration

For the purposes of this guide, we assume a minimal setup in line with the examples given in the previous sections:

Listing 7. Simple DRBD configuration (`/etc/drbd.d/global_common.conf`)

```
global {
  usage-count yes;
}
common {
  net {
    protocol C;
  }
}
```

Listing 8. Simple DRBD resource configuration (`/etc/drbd.d/r0.res`)

```
resource r0 {
  on alice {
    device    /dev/drbd1;
    disk      /dev/sda7;
    address   10.1.1.31:7789;
    meta-disk internal;
  }
  on bob {
    device    /dev/drbd1;
    disk      /dev/sda7;
    address   10.1.1.32:7789;
    meta-disk internal;
  }
}
```

This example configures DRBD in the following fashion:

- You "opt in" to be included in DRBD's usage statistics (see `usage-count`).
- Resources are configured to use fully synchronous replication (Protocol C) unless explicitly specified otherwise.
- Our cluster consists of two nodes, 'alice' and 'bob'.
- We have a resource arbitrarily named `r0` which uses `/dev/sda7` as the lower-level device, and is configured with internal meta data.
- The resource uses TCP port 7789 for its network connections, and binds to the IP addresses 10.1.1.31 and 10.1.1.32, respectively. (This implicitly defines the network connections that are used.)

The configuration above implicitly creates one volume in the resource, numbered zero (`0`). For multiple volumes in one resource, modify the syntax as follows (assuming that the same lower-level storage block devices are used on both nodes):

Listing 9. Multi-volume DRBD resource configuration (`/etc/drbd.d/r0.res`)

```
resource r0 {
  volume 0 {
    device    /dev/drbd1;
    disk      /dev/sda7;
    meta-disk internal;
  }
  volume 1 {
    device    /dev/drbd2;
    disk      /dev/sda8;
    meta-disk internal;
  }
  on alice {
    address   10.1.1.31:7789;
  }
  on bob {
    address   10.1.1.32:7789;
  }
}
```

Volumes may also be added to existing resources on the fly. For an example see Adding a new DRBD volume to an existing Volume Group.

The `global` section

This section is allowed only once in the configuration. It is normally in the `/etc/drbd.d/global_common.conf` file. In a single-file configuration, it should go to the very top of the configuration file. Of the few options available in this section, only one is of relevance to most users:

`usage-count`
The DRBD project keeps statistics about the usage of various DRBD versions. This is done by contacting an HTTP server every time a new DRBD version is installed on a system. This can be disabled by setting `usage-count no;`. The default is `usage-count ask;` which will prompt you every time you upgrade DRBD.

DRBD's usage statistics are, of course, publicly available: see http://usage.drbd.org.

The `common` section

This section provides a shorthand method to define configuration settings inherited by every resource. It is normally found in `/etc/drbd.d/global_common.conf`. You may define any option you can also define on a per-resource basis.

Including a `common` section is not strictly required, but strongly recommended if you are using more than one resource. Otherwise, the configuration quickly becomes convoluted by repeatedly-used options.

In the example above, we included `net { protocol C; }` in the `common` section, so every resource configured (including `r0`) inherits this option unless it has another `protocol` option configured explicitly. For other synchronization protocols available, see Replication modes.

The `resource` sections

A per-resource configuration file is usually named `/etc/drbd.d/`*`resource`*`.res`. Any DRBD resource you define must be named by specifying a resource name in the configuration. The convention is to use only letters, digits, and the underscore; while it is technically possible to use other characters as well, you won't like the result if you ever happen stumble to need the more specific *`peer@resource/volume`* syntax.

Every resource configuration must also have at least two `on` *`host`* sub-sections, one for every cluster node. All other configuration settings are either inherited from the `common` section (if it exists), or derived from DRBD's default settings.

In addition, options with equal values on all hosts can be specified directly in the `resource` section. Thus, we can further condense our example configuration as follows:

```
resource r0 {
  device    /dev/drbd1;
  disk      /dev/sda7;
  meta-disk internal;
  on alice {
    address   10.1.1.31:7789;
  }
  on bob {
    address   10.1.1.32:7789;
  }
}
```

11.1.4. Defining network connections

Currently the communication links in DRBD 9 must build a full mesh, ie. in every resource every node must have a direct connection to every other node (excluding itself, of course).

For the simple case of two hosts `drbdadm` will insert the (single) network connection by itself, for ease of use and backwards compatibility.

The net effect of this is a quadratic number of network connections over hosts. For the "traditional" two nodes one connection is needed; for three hosts there are three node pairs; for four, six pairs; 5 hosts: 10 connections, and so on. For (the current) maximum of 16 nodes there'll be 120 host pairs to connect.

$$C = \frac{N(N-1)}{2}$$

Figure 9. Number of connections for N hosts

An example configuration file for three hosts would be this:

```
resource r0 {
  device    /dev/drbd1;
  disk      /dev/sda7;
  meta-disk internal;
  on alice {
    address   10.1.1.31:7000;
    node-id   0;
  }
  on bob {
    address   10.1.1.32:7001;
    node-id   1;
  }
  on charlie {
    address   10.1.1.33:7002;
    node-id   2;
  }
  connection {
    host alice   port 7010;
    host bob     port 7001;
  }
  connection {
    host alice   port 7020;
    host charlie port 7002;
  }
  connection {
    host bob     port 7012;
    host charlie port 7021;
  }

}
```

The port for the `address` value within the `on` *`host`* sections is optional; but then you have to specify distinct port numbers for each connection.

For this pre-release the whole connection mesh must be defined.

In the final release it will be sufficient to give each node a single port, and DRBD will figure out during the handshake which peer node it is talking to.

Alternatively, using the `connection-mesh` option, the same three node configuration:

```
resource r0 {
  device    /dev/drbd1;
  disk      /dev/sda7;
  meta-disk internal;
  on alice {
    address   10.1.1.31:7000;
    node-id   0;
  }
  on bob {
    address   10.1.1.32:7001;
    node-id   1;
  }
  on charlie {
    address   10.1.1.33:7002;
    node-id   2;
  }
  connection-mesh {
    hosts alice bob charlie;
    net {
        use-rle no;
    }
  }

}
```

If you have got enough network cards in your servers, you can create direct cross-over links between server pairs. A single four-port ethernet card allows to have a single management interface, and to connect 3 other servers, to get a full mesh for 4 cluster nodes.

In this case you can specify a different IP address to use the direct link:

```
resource r0 {
  ...
  connection {
    host alice   address 10.1.2.1 port 7010;
    host bob     address 10.1.2.2 port 7001;
  }
  connection {
    host alice   address 10.1.3.1 port 7020;
    host charlie address 10.1.3.2 port 7002;
  }
  connection {
    host bob     address 10.1.4.1 port 7021;
    host charlie address 10.1.4.2 port 7012;
  }
}
```

For easier maintenance and debugging it's recommend to have different ports for each endpoint - looking at a `tcpdump` trace the packets can be associated easily.

The examples below will still be using two servers only; please see Example configuration for four nodes for a four-node example.

11.1.5. Configuring transport implementations

DRBD supports multiple network transports. A transport implementation can be configured for each connection of a resource.

TCP/IP

```
resource <resource> {
  net {
    transport "tcp";
  }
  ...
}
```

`tcp` is the default transport. I.e. each connection that lacks a transport option uses the `tcp` transport.

The `tcp` transport can be configured with the net options: `sndbuf-size`, `rcvbuf-size`, `connect-int`, `sock-check-timeo`, `ping-timeo`, `timeout`.

RDMA

```
resource <resource> {
  net {
    transport "rdma";
  }
  ...
}
```

The `rdma` transport can be configured with the net options: `sndbuf-size`, `rcvbuf-size`, `max_buffers`, `connect-int`, `sock-check-timeo`, `ping-timeo`, `timeout`.

The `rdma` transport is a zero-copy-receive transport. One implication of that is that the `max_buffers` configuration options must be set to a value big enough to hold all `rcvbuf-size`.

`rcvbuf-size` is configured in bytes, while `max_buffers` is configured in pages. For optimal performance `max_buffers` should be big enough to hold all of `rcvbuf-size` and the amount of data that might be in flight to the backend device at any point in time.

In case you are using InfiniBand HCAs with the `rdma` transport, you need to configure IPoIB as well. The IP address is not used for data transfer, but it is used to find the right adapters and ports while establishing the connection.

The configuration options `sndbuf-size`, `rcvbuf-size` are only considered at the time a connection is established. I.e. you can change them while the connection is established. They will take effect in the moment the connection is re-established.

Performance considerations for RDMA

By looking at the pseudo file */sys/kernel/debug/drbd/<resource>/connections/<peer>/transport*, the counts of available receive descriptors (rx_desc) and transmit descriptors (tx_desc) can be monitored. In case one of the descriptor kinds becomes depleted you should increase `sndbuf-size` or `rcvbuf-size`.

11.1.6. Enabling your resource for the first time

After you have completed initial resource configuration as outlined in the previous sections, you can bring up your resource.

Each of the following steps must be completed on both nodes.

Please note that with our example config snippets (`resource r0 { … }`), `<resource>` would be `r0`.

Create device metadata

This step must be completed only on initial device creation. It initializes DRBD's metadata:

```
# drbdadm create-md <resource>
v09 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block sucessfully created.
```

Please note that the number of bitmap slots that are allocated in the meta-data depends on the number of hosts for this resource; per default the hosts in the resource configuration are counted. If all hosts are specified *before* creating the meta-data, this will "just work"; adding bitmap slots for further nodes is possible later, but incurs some manual work.

Enable the resource

This step associates the resource with its backing device (or devices, in case of a multi-volume resource), sets replication parameters, and connects the resource to its peer:

```
# drbdadm up <resource>
```

Observe the status via `drbdadm status`

`drbdsetup`'s status output should now contain information similar to the following:

```
# drbdadm status r0
r0 role:Secondary
  disk:Inconsistent
  bob role:Secondary
    disk:Inconsistent
```

The *Inconsistent/Inconsistent* disk state is expected at this point.

By now, DRBD has successfully allocated both disk and network resources and is ready for operation. What it does not know yet is which of your nodes should be used as the source of the initial device synchronization.

11.1.7. The initial device synchronization

There are two more steps required for DRBD to become fully operational:

Select an initial sync source

If you are dealing with newly-initialized, empty disks, this choice is entirely arbitrary. If one of your nodes already has valuable data that you need to preserve, however, *it is of crucial importance* that you select that node as your synchronization source. If you do initial device synchronization in the wrong direction, you will lose that data. Exercise caution.

Start the initial full synchronization

This step must be performed on only one node, only on initial resource configuration, and only on the node you selected as the synchronization source. To perform this step, issue this command:

```
# drbdadm primary --force <resource>
```

After issuing this command, the initial full synchronization will commence. You will be able to monitor its progress via `drbdadm status`. It may take some time depending on the size of the device.

By now, your DRBD device is fully operational, even before the initial synchronization has completed (albeit with slightly reduced performance). If you started with empty disks you may now already create a filesystem on the device, use it as a raw block device, mount it, and perform any other operation you would with an accessible block device.

You will now probably want to continue with Working with DRBD, which describes common administrative tasks to perform on your resource.

11.1.8. Using truck based replication

In order to preseed a remote node with data which is then to be kept synchronized, and to skip the initial full device synchronization, follow these steps.

This assumes that your local node has a configured, but disconnected DRBD resource in the *Primary* role. That is to say, device configuration is completed, identical `drbd.conf` copies exist on both nodes, and you have issued the commands for initial resource promotion on your local node — but the remote node is not connected yet.

- On the local node, issue the following command:

  ```
  # drbdadm new-current-uuid --clear-bitmap <resource>/<volume>
  ```

  or

  ```
  # drbdsetup new-current-uuid --clear-bitmap <minor>
  ```

- Create a consistent, verbatim copy of the resource's data *and its metadata*. You may do so, for example, by removing a hot-swappable drive from a RAID-1 mirror. You would, of course, replace it with a fresh drive, and rebuild the RAID set, to ensure continued redundancy. But the removed drive is a verbatim copy that can now be shipped off site. If your local block device supports snapshot copies (such as when using DRBD on top of LVM), you may also create a bitwise copy of that snapshot using `dd`.
- On the local node, issue:

  ```
  # drbdadm new-current-uuid <resource>
  ```

  or the matching `drbdsetup` command.

  Note the absence of the `--clear-bitmap` option in this second invocation.

- Physically transport the copies to the remote peer location.
- Add the copies to the remote node. This may again be a matter of plugging a physical disk, or grafting a bitwise copy of your shipped data onto existing storage on the remote node. Be sure to restore or copy not only your replicated data, but also the associated DRBD metadata. If you fail to do so, the disk shipping process is moot.
- On the new node we need to fix the node ID in the meta data, and exchange the peer-node info for the two nodes. Please see the following lines as example for changing node id from `2` to `1` on a resource `r0` volume `0`.

This must be done while the volume is not in use.

```
V=r0/0
NODE_FROM=2
NODE_TO=1

drbdadm -- --force dump-md $V > /tmp/md_orig.txt
sed -e "s/node-id $NODE_FROM/node-id $NODE_TO/" \
    -e "s/^peer.$NODE_FROM. /peer-NEW /" \
    -e "s/^peer.$NODE_TO. /peer[$NODE_FROM] /" \
    -e "s/^peer-NEW /peer[$NODE_TO] /" \
    < /tmp/md_orig.txt > /tmp/md.txt

drbdmeta --force $(drbdadm sh-minor $V) v09 $(drbdadm sh-ll-dev $V) internal restore-md
/tmp/md.txt
```

NOTE

`drbdmeta` before 8.9.7 cannot cope with out-of-order `peer` sections; you'll need to exchange the blocks via an editor.

- Bring up the resource on the remote node:

```
# drbdadm up <resource>
```

After the two peers connect, they will not initiate a full device synchronization. Instead, the automatic synchronization that now commences only covers those blocks that changed since the invocation of `drbdadm --clear-bitmap new-current-uuid`.

Even if there were *no* changes whatsoever since then, there may still be a brief synchronization period due to areas covered by the Activity Log being rolled back on the new Secondary. This may be mitigated by the use of checksum-based synchronization.

You may use this same procedure regardless of whether the resource is a regular DRBD resource, or a stacked resource. For stacked resources, simply add the `-S` or `--stacked` option to `drbdadm`.

11.1.9. Example configuration for four nodes

Here is an example for a four-node cluster.

Please note that the `connection` sections (and distinct ports) won't be necessary for the DRBD 9.0.0 release; we will find some nice short-hand syntax.

```
resource r0 {
  device      /dev/drbd0;
  disk        /dev/vg/r0;
  meta-disk   internal;

  on store1 {
    address   10.1.10.1:7100;
    node-id   1;
  }
  on store2 {
    address   10.1.10.2:7100;
    node-id   2;
  }
  on store3 {
    address   10.1.10.3:7100;
    node-id   3;
  }
  on store4 {
    address   10.1.10.4:7100;
    node-id   4;
  }

  # All connections involving store1
  connection {
    host store1  port 7012;
    host store2  port 7021;
  }
  connection {
    host store1  port 7013;
    host store3  port 7031;
  }
  connection {
    host store1  port 7014;
    host store4  port 7041;
  }

  # All remaining connections involving store2
  connection {
    host store2  port 7023;
    host store3  port 7032;
  }
  connection {
    host store2  port 7024;
    host store4  port 7042;
  }

  # All remaining connections involving store3
  connection {
    host store3  port 7034;
    host store4  port 7043;
  }

  # store4 already done.
}
```

In contrast, the same configuration will be written like this:

```
resource r0 {
  device      /dev/drbd0;
  disk        /dev/vg/r0;
  meta-disk   internal;

  on store1 {
    address   10.1.10.1:7100;
    node-id   1;
  }
  on store2 {
    address   10.1.10.2:7100;
    node-id   2;
  }
  on store3 {
    address   10.1.10.3:7100;
    node-id   3;
  }
  on store4 {
    address   10.1.10.4:7100;
    node-id   4;
  }

  connection-mesh {
    hosts     store1 store2 store3 store4;
  }
}
```

In case you want to see the `connection-mesh` configuration expanded, try `drbdadm dump <resource> -v`.

As another example, if the four nodes have enough interfaces to provide a complete mesh via direct links [6: ie. three crossover and at least one outgoing/management interface], you can specify the IP addresses of the interfaces:

```
resource r0 {
  ...

  # store1 has crossover links like 10.99.1x.y
  connection {
    host store1  address 10.99.12.1 port 7012;
    host store2  address 10.99.12.2 port 7021;
  }
  connection {
    host store1  address 10.99.13.1  port 7013;
    host store3  address 10.99.13.3  port 7031;
  }
  connection {
    host store1  address 10.99.14.1  port 7014;
    host store4  address 10.99.14.4  port 7041;
  }

  # store2 has crossover links like 10.99.2x.y
  connection {
    host store2  address 10.99.23.2  port 7023;
    host store3  address 10.99.23.3  port 7032;
  }
  connection {
    host store2  address 10.99.24.2  port 7024;
    host store4  address 10.99.24.4  port 7042;
  }

  # store3 has crossover links like 10.99.3x.y
  connection {
    host store3  address 10.99.34.3  port 7034;
    host store4  address 10.99.34.4  port 7043;
  }
}
```

Please note the numbering scheme used for the IP addresses and ports. Another resource could use the same IP addresses, but ports 71*xy*, the next one 72*xy*, and so on.

11.2. Checking DRBD status

11.2.1. Retrieving status with drbdmon

One convenient way to look at DRBD's status is the `drbdmon` utility. It updates the state of DRBD resources in realtime.

11.2.2. Retrieving status and interacting with DRBD via drbdtop

As its name suggests, `drbdtop` shares similarities with tools like `htop`. On one hand it allows monitoring of DRBD resources as well as interacting (e.g., switching them to *Primary*, or even resolving split-brains). A complete overview can be found here[https://linbit.github.io/drbdtop/].

11.2.3. Status information in `/proc/drbd`

`/proc/drbd` is deprecated. While it won't be removed in the 8.4 series, we recommend to switch to other means, like Status information via `drbdadm`; or, for monitoring even more convenient, One-shot or realtime monitoring via `drbdsetup events2`.

`/proc/drbd` is a virtual file displaying basic information about the DRBD module. It was used extensively up to DRBD 8.4, but couldn't keep up with the amount of information provided by DRBD 9.

```
$ cat /proc/drbd
version: 9.0.0 (api:1/proto:86-110) FIXME
GIT-hash: XXX build by linbit@buildsystem.linbit, 2011-10-12 09:07:35
```

The first line, prefixed with `version:`, shows the DRBD version used on your system. The second line contains information about this specific build.

11.2.4. Status information via `drbdadm`

In its simplest invocation, we just ask for the status of a single resource.

```
# drbdadm status home
home role:Secondary
  disk:UpToDate
  nina role:Secondary
    disk:UpToDate
  nino role:Secondary
    disk:UpToDate
  nono connection:Connecting
```

This here just says that the resource *home* is locally, on 'nina', and on 'nino' *UpToDate* and *Secondary*; so the three nodes have the same data on their storage devices, and nobody is using the device currently.

The node 'nono' is not connected, its state is reported as *Connecting*; please see Connection states below for more details.

You can get more information by passing the `--verbose` and/or `--statistics` arguments to `drbdsetup` (lines broken for readability):

```
# drbdsetup status home --verbose --statistics
home node-id:1 role:Secondary suspended:no
    write-ordering:none
  volume:0 minor:0 disk:UpToDate
      size:1048412 read:0 written:1048412 al-writes:0 bm-writes:48 upper-pending:0
                                          lower-pending:0 al-suspended:no blocked:no
  nina local:ipv4:10.9.9.111:7001 peer:ipv4:10.9.9.103:7010 node-id:0
                                                connection:Connected role:Secondary
      congested:no
    volume:0 replication:Connected disk:UpToDate resync-suspended:no
        received:1048412 sent:0 out-of-sync:0 pending:0 unacked:0
  nino local:ipv4:10.9.9.111:7021 peer:ipv4:10.9.9.129:7012 node-id:2
                                                connection:Connected role:Secondary
      congested:no
    volume:0 replication:Connected disk:UpToDate resync-suspended:no
        received:0 sent:0 out-of-sync:0 pending:0 unacked:0
  nono local:ipv4:10.9.9.111:7013 peer:ipv4:10.9.9.138:7031 node-id:3
                                                          connection:Connecting
```

Every few lines in this example form a block that is repeated for every node used in this resource, with small format exceptions for the local node - see below for more details.

The first line in each block shows the `node-id` (for the current resource; a host can have different `node-id`s in different resources). Furthermore the `role` (see Resource roles) is shown.

The next important line begins with the `volume` specification; normally these are numbered starting by zero, but the configuration may specify other IDs as well. This line shows the connection state in the `replication` item (see Connection states for details) and the remote disk state in `disk` (see Disk states). Then there's a line for this volume giving a bit of statistics - data `received`, `sent`, `out-of-sync`, etc; please see Performance indicators and Connection information data for more information.

For the local node the first line shows the resource name, *home*, in our example. As the first block always describes the local node, there is no *Connection* or address information.

please see the `drbd.conf` manual page for more information.

The other four lines in this example form a block that is repeated for every DRBD device configured, prefixed by the device minor number. In this case, this is `0`, corresponding to the device `/dev/drbd0`.

The resource-specific output contains various pieces of information about the resource:

11.2.5. One-shot or realtime monitoring via `drbdsetup events2`

This is available only with userspace versions 8.9.3 and up.

This is a low-level mechanism to get information out of DRBD, suitable for use in automated tools, like monitoring.

In its simplest invocation, showing only the current status, the output looks like this (but, when running on a terminal, will include colors):

```
# drbdsetup events2 --now r0
exists resource name:r0 role:Secondary suspended:no
exists connection name:r0 peer-node-id:1 conn-name:remote-host connection:Connected
role:Secondary
exists device name:r0 volume:0 minor:7 disk:UpToDate
exists device name:r0 volume:1 minor:8 disk:UpToDate
exists peer-device name:r0 peer-node-id:1 conn-name:remote-host volume:0
    replication:Established peer-disk:UpToDate resync-suspended:no
exists peer-device name:r0 peer-node-id:1 conn-name:remote-host volume:1
    replication:Established peer-disk:UpToDate resync-suspended:no
exists -
```

Without the "--now", the process will keep running, and send continuous updates like this:

```
# drbdsetup events2 r0
...
change connection name:r0 peer-node-id:1 conn-name:remote-host connection:StandAlone
change connection name:r0 peer-node-id:1 conn-name:remote-host connection:Unconnected
change connection name:r0 peer-node-id:1 conn-name:remote-host connection:Connecting
```

Then, for monitoring purposes, there's another argument "--statistics", that will produce some performance counters and other facts:

'drbdsetup' **verbose output** (lines broken for readability):

```
# drbdsetup events2 --statistics --now r0
exists resource name:r0 role:Secondary suspended:no write-ordering:drain
exists connection name:r0 peer-node-id:1 conn-name:remote-host connection:Connected
                                                    role:Secondary congested:no
exists device name:r0 volume:0 minor:7 disk:UpToDate size:6291228 read:6397188
           written:131844 al-writes:34 bm-writes:0 upper-pending:0 lower-pending:0
                                                     al-suspended:no blocked:no
exists device name:r0 volume:1 minor:8 disk:UpToDate size:104854364 read:5910680
         written:6634548 al-writes:417 bm-writes:0 upper-pending:0 lower-pending:0
                                                     al-suspended:no blocked:no
exists peer-device name:r0 peer-node-id:1 conn-name:remote-host volume:0
         replication:Established peer-disk:UpToDate resync-suspended:no received:0
                                    sent:131844 out-of-sync:0 pending:0 unacked:0
exists peer-device name:r0 peer-node-id:1 conn-name:remote-host volume:1
         replication:Established peer-disk:UpToDate resync-suspended:no received:0
                                   sent:6634548 out-of-sync:0 pending:0 unacked:0
exists -
```

You might also like the "--timestamp" parameter.

11.2.6. Connection states

A resource's connection state can be observed either by issuing the `drbdadm cstate` command:

```
# drbdadm cstate <resource>
Connected
Connected
StandAlone
```

If you are interested in only a single connection of a resource, specify the connection name, too:

The default is the peer's hostname as given in the configuration file.

```
# drbdadm cstate <peer>:<resource>
Connected
```

A resource may have one of the following connection states:

StandAlone
No network configuration available. The resource has not yet been connected, or has been administratively disconnected (using `drbdadm disconnect`), or has dropped its connection due to failed authentication or split brain.

Disconnecting
Temporary state during disconnection. The next state is *StandAlone*.

Unconnected
Temporary state, prior to a connection attempt. Possible next states: *Connecting*.

Timeout
Temporary state following a timeout in the communication with the peer. Next state: *Unconnected*.

BrokenPipe
Temporary state after the connection to the peer was lost. Next state: *Unconnected*.

NetworkFailure
Temporary state after the connection to the partner was lost. Next state: *Unconnected*.

ProtocolError
Temporary state after the connection to the partner was lost. Next state: *Unconnected*.

TearDown
Temporary state. The peer is closing the connection. Next state: *Unconnected*.

Connecting
This node is waiting until the peer node becomes visible on the network.

Connected
A DRBD connection has been established, data mirroring is now active. This is the normal state.

11.2.7. Replication states

Each volume has over each connection a replication state. The possible replication states are:

Off
The volume is not replicated over this connection, since the connection is not *Connected*.

Established

All writes to that volume are replicated online. This is the normal state.

StartingSyncS

Full synchronization, initiated by the administrator, is just starting. The next possible states are: *SyncSource* or *PausedSyncS*.

StartingSyncT

Full synchronization, initiated by the administrator, is just starting. Next state: *WFSyncUUID*.

WFBitMapS

Partial synchronization is just starting. Next possible states: *SyncSource* or *PausedSyncS*.

WFBitMapT

Partial synchronization is just starting. Next possible state: *WFSyncUUID*.

WFSyncUUID

Synchronization is about to begin. Next possible states: *SyncTarget* or *PausedSyncT*.

SyncSource

Synchronization is currently running, with the local node being the source of synchronization.

SyncTarget

Synchronization is currently running, with the local node being the target of synchronization.

PausedSyncS

The local node is the source of an ongoing synchronization, but synchronization is currently paused. This may be due to a dependency on the completion of another synchronization process, or due to synchronization having been manually interrupted by `drbdadm pause-sync`.

PausedSyncT

The local node is the target of an ongoing synchronization, but synchronization is currently paused. This may be due to a dependency on the completion of another synchronization process, or due to synchronization having been manually interrupted by `drbdadm pause-sync`.

VerifyS

On-line device verification is currently running, with the local node being the source of verification.

VerifyT

On-line device verification is currently running, with the local node being the target of verification.

Ahead

Data replication was suspended, since the link can not cope with the load. This state is enabled by the configuration `on-congestion` optione (see Configuring congestion policies and suspended replication).

Behind

Data replication was suspended by the peer, since the link can not cope with the load. This state is enabled by the configuration `on-congestion` option on the peer node (see Configuring congestion policies and suspended replication).

11.2.8. Resource roles

A resource's role can be observed by issuing the `drbdadm role` command:

```
# drbdadm role <resource>
Primary
```

You may see one of the following resource roles:

Primary
The resource is currently in the primary role, and may be read from and written to. This role only occurs on one of the two nodes, unless dual-primary mode is enabled.

Secondary
The resource is currently in the secondary role. It normally receives updates from its peer (unless running in disconnected mode), but may neither be read from nor written to. This role may occur on one or both nodes.

Unknown
The resource's role is currently unknown. The local resource role never has this status. It is only displayed for the peer's resource role, and only in disconnected mode.

11.2.9. Disk states

A resource's disk state can be observed either by issuing the `drbdadm dstate` command:

```
# drbdadm dstate <resource>
UpToDate
```

The disk state may be one of the following:

Diskless
No local block device has been assigned to the DRBD driver. This may mean that the resource has never attached to its backing device, that it has been manually detached using `drbdadm detach`, or that it automatically detached after a lower-level I/O error.

Attaching
Transient state while reading meta data.

Detaching
Transient state while detaching and waiting for ongoing IOs to complete.

Failed
Transient state following an I/O failure report by the local block device. Next state: *Diskless*.

Negotiating
Transient state when an *Attach* is carried out on an already-*Connected* DRBD device.

Inconsistent
The data is inconsistent. This status occurs immediately upon creation of a new resource, on both nodes (before the initial full sync). Also, this status is found in one node (the synchronization target) during synchronization.

Outdated
Resource data is consistent, but outdated.

DUnknown
This state is used for the peer disk if no network connection is available.

Consistent
Consistent data of a node without connection. When the connection is established, it is decided whether the data is *UpToDate* or *Outdated*.

UpToDate
Consistent, up-to-date state of the data. This is the normal state.

11.2.10. Connection information data

local
Shows the network family, the local address and port that is used to accept connections from the peer.

peer
Shows the network family, the peer address and port that is used to connect.

congested
This flag tells whether the TCP send buffer of the data connection is more than 80% filled.

11.2.11. Performance indicators

One line of `drbdadm status`-output includes the following counters and gauges:

send (network send)
Volume of net data sent to the partner via the network connection; in Kibyte.

receive (network receive)
Volume of net data received by the partner via the network connection; in Kibyte.

read (disk write)
Net data written on local hard disk; in Kibyte.

written (disk read)
Net data read from local hard disk; in Kibyte.

al-writes (activity log)
Number of updates of the activity log area of the meta data.

bm-writes (bit map)
Number of updates of the bitmap area of the meta data.

lower-pending (local count)
Number of open requests to the local I/O sub-system issued by DRBD.

pending
Number of requests sent to the partner, but that have not yet been answered by the latter.

unacked (unacknowledged)
Number of requests received by the partner via the network connection, but that have not yet been answered.

upper-pending (application pending)
Number of block I/O requests forwarded to DRBD, but not yet answered by DRBD.

write-ordering (write order)
Currently used write ordering method: *b*(barrier), *f*(flush), *d*(drain) or *n*(none).

out-of-sync
Amount of storage currently out of sync; in Kibibytes.

resync-suspended
Whether the resynchronization is currently suspended or not. Possible values are *no*, *user*, *peer*, *dependency*.

blocked
Shows local I/O congestion.

- *no*: No congestion.
- *upper*: I/O *above* the DRBD device is blocked, ie. to the filesystem. Typical causes are
  - I/O suspension by the administrator, see the `suspend-io` command in `drbdadm`.
  - transient blocks, eg. during attach/detach
  - buffers depleted, see Optimizing DRBD performance
  - Waiting for bitmap IO
- *lower*: Backing device is congested.

It's possible to see a value of *upper,lower*, too.

11.3. Enabling and disabling resources

11.3.1. Enabling resources

Normally, all configured DRBD resources are automatically enabled

- by a cluster resource management application at its discretion, based on your cluster configuration, or
- by the `/etc/init.d/drbd` init script on system startup.

If, however, you need to enable resources manually for any reason, you may do so by issuing the command

```
# drbdadm up <resource>
```

As always, you may use the keyword `all` instead of a specific resource name if you want to enable all resources configured in `/etc/drbd.conf` at once.

11.3.2. Disabling resources

You may temporarily disable specific resources by issuing the command

```
# drbdadm down <resource>
```

Here, too, you may use the keyword `all` in place of a resource name if you wish to temporarily disable all resources listed in `/etc/drbd.conf` at once.

11.4. Reconfiguring resources

DRBD allows you to reconfigure resources while they are operational. To that end,

- make any necessary changes to the resource configuration in `/etc/drbd.conf`,
- synchronize your `/etc/drbd.conf` file between both nodes,
- issue the `drbdadm adjust <resource>` command on both nodes.

`drbdadm adjust` then hands off to `drbdsetup` to make the necessary adjustments to the configuration. As always, you are able to review the pending `drbdsetup` invocations by running `drbdadm` with the `-d` (dry-run) option.

When making changes to the `common` section in `/etc/drbd.conf`, you can adjust the configuration for all resources in one run, by issuing `drbdadm adjust all`.

11.5. Promoting and demoting resources

Manually switching a resource's role from secondary to primary (promotion) or vice versa (demotion) is done using one of the following commands:

```
# drbdadm primary <resource>
# drbdadm secondary <resource>
```

In single-primary mode (DRBD's default), any resource can be in the primary role on only one node at any given time while the connection state is *Connected*. Thus, issuing `drbdadm primary <resource>` on one node while the specified resource is still in the primary role on another node will result in an error.

A resource configured to allow dual-primary mode can be switched to the primary role on two nodes; this is eg. needed for online migration of virtual machines.

11.6. Basic Manual Fail-over

If not using Pacemaker and looking to handle fail-overs manually in a passive/active configuration, the process is as follows.

On the current primary node stop any applications or services using the DRBD device, unmount the DRBD device, and demote the resource to secondary.

```
# umount /dev/drbd/by-res/<resource>/<vol-nr>
# drbdadm secondary <resource>
```

Now on the node we wish to make primary promote the resource and mount the device.

```
# drbdadm primary <resource>
# mount /dev/drbd/by-res/<resource>/<vol-nr> <mountpoint>
```

If you're using the `auto-promote` feature, you don't need to change the roles (*Primary*/*Secondary*) manually; only stopping of the services and umounting, respectively mounting, is necessary.

11.7. Upgrading DRBD

Upgrading DRBD is a fairly simple process. This section will cover the process of upgrading from 8.4.x to 9.0.x in great detail; for within-9 upgrades it gets to be much easier, see the short version below.

11.7.1. General overview

The general process for upgrading 8.4 to 9.0 is as follows:

- Configure the new repositories (if using packages from LINBIT)
- Make sure that the current situation is okay
- Pause any cluster manager
- Get new versions installed
- If wanting to move to more than 2 nodes, then you'll need to resize the lower-level storage to provide room for the additional meta-data; this topic is discussed in the LVM Chapter.
- Deconfigure resources, unload DRBD 8.4, and load the v9 kernel module

- Convert DRBD meta-data to format `v09`, perhaps changing the number of bitmaps in the same step
- Take the DRBD resources up, and be happy

11.7.2. Updating your repository

Due to the number of changes between the 8.4 and 9.0 branches we have created separate repositories for each. Perform this repository update on both servers.

RHEL/CentOS systems

Edit your `/etc/yum.repos.d/linbit.repo` file to reflect the following changes.

```
[drbd-9.0]
name=DRBD 9.0
baseurl=http://packages.linbit.com/<hash>/yum/rhel7/drbd-9.0/<arch>
gpgcheck=0
```

You will have to populate the <hash> and <arch> variables. The <hash> is provided by LINBIT support services.

Debian/Ubuntu systems

Edit `/etc/apt/sources.list` (or a file in `/etc/apt/sources.d/`) to reflect the following changes.

```
deb http://packages.linbit.com/<hash>/ stretch drbd-9.0
```

In case you're not using the `stretch` release, but some other, you'll need to change that line.

You will have to populate the <hash> variable. The <hash> is provided by LINBIT support services.

Next you will want to add the DRBD signing key to your trusted keys.

```
# gpg --keyserver subkeys.pgp.net --recv-keys  0x282B6E23
# gpg --export -a 282B6E23 | apt-key add -
```

Lastly perform an `apt update` so Debian recognizes the updated repository.

```
# apt update
```

11.7.3. Checking the DRBD state

Before you begin make sure your resources are in sync. The output of `cat /proc/drbd` (which is only available *before* 9.0) should show *UpToDate/UpToDate*.

```
bob# cat /proc/drbd

version: 8.4.9-1 (api:1/proto:86-101)
GIT-hash: e081fb0570183db40caa29b26cb8ee907e9a7db3 build by linbit@buildsystem, 2016-11-18
14:49:21

 0: cs:Connected ro:Secondary/Secondary ds:UpToDate/UpToDate C r-----
    ns:0 nr:211852 dw:211852 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:d oos:0
```

11.7.4. Pausing the cluster

Now that you know the resources are in sync, start by upgrading the secondary node. This can be done manually or if you're using Pacemaker put the node in standby mode. Both processes are covered below. If you're running Pacemaker do not use the manual method.

- Manual Method

```
bob# /etc/init.d/drbd stop
```

- Pacemaker

Put the secondary node into standby mode. In this example 'bob' is secondary.

```
bob# crm node standby bob
```

You can watch the status of your cluster using `crm_mon -rf` or watch `cat /proc/drbd` until it shows *Unconfigured* for your resources.

11.7.5. Upgrading the packages

Now update your packages with either yum or apt.

```
bob# yum upgrade
```

```
bob# apt upgrade
```

Once the upgrade is finished will now have the latest DRBD 9.0 kernel module and drbd-utils installed on your secondary node, 'bob'.

But the kernel module is not active yet.

11.7.6. Loading the new Kernel module

By now the DRBD module should not be in use anymore, so we unload it via

```
bob# rmmod drbd
```

If there's a message like `ERROR: Module drbd is in use`, then not all resources have been correctly stopped

yet.
Retry Upgrading DRBD, and/or run the command `drbdadm down all` to find out which resources are still active.

Typical issues that might prevent unloading are these:

- NFS export on a DRBD-backed filesystem (see `exportfs -v` output)
- Filesystem still mounted - check `grep drbd /proc/mounts`
- Loopback device active (`losetup -l`)
- Device mapper using DRBD, directly or indirectly (`dmsetup ls --tree`)
- LVM with a DRBD-PV (`pvs`)

Please note that this list isn't complete - these are just the most common examples.

Now we can load the new DRBD module:

```
bob# modprobe drbd
```

Now you should check the contents of `/proc/drbd` and verify that the correct (new) version is loaded; if the installed packages is for the wrong kernel version, the `modprobe` would be successful, but you'd be left with the old version being active again.

The output of `cat /proc/drbd` should now show 9.0.x and look similar to this.

```
version: 9.0.0 (api:2/proto:86-110)
GIT-hash: 768965a7f158d966bd3bd4ff1014af7b3d9ff10c build by root@bob, 2015-09-03 13:58:02
Transports (api:10): tcp (1.0.0)
```

On the primary node, alice, 'cat /proc/drbd' will still show the prior version, until you upgrade it.

11.7.7. Migrating your configuration files

DRBD 9.0 is backward compatible with the 8.4 configuration files; however, some syntax has changed. See Changes to the configuration syntax for a full list of changes. In the meantime you can port your old configs fairly easily by using 'drbdadm dump all' command. This will output both a new global config followed by the new resource config files. Take this output and make changes accordingly.

11.7.8. Changing the meta-data

Now you need to convert the on-disk metadata to the new version; this is really easy, it's just running one command and acknowledging two questions.

If you want to change the number of nodes, you should already have increased the size of the lower level device, so that there is enough space to store the additional bitmaps; in that case, you'd run the command below with an additional argument `--max-peers=<N>`. When determining the number of (possible) peers please take setups like the DRBD client into account.

Upgrading the DRBD metadata is as easy as running one command, and acknowledging the two questions:

```
# drbdadm create-md <resource>
You want me to create a v09 style flexible-size internal meta data block.
There appears to be a v08 flexible-size internal meta data block
already in place on <disk> at byte offset <offset>

Valid v08 meta-data found, convert to v09?
[need to type 'yes' to confirm] yes

md_offset <offsets...>
al_offset <offsets...>
bm_offset <offsets...>

Found some data

 ==> This might destroy existing data! <==

Do you want to proceed?
[need to type 'yes' to confirm] yes

Writing meta data...
New drbd meta data block successfully created.
success
```

Of course, you can pass `all` for the resource names, too; and if you feel really lucky, you can avoid the questions via a commandline like this here, too. (Yes, the order is important.)

```
drbdadm -v --max-peers=<N>  -- --force create-md <resources>
```

11.7.9. Starting DRBD again

Now, the only thing left to do is to get the DRBD devices up and running again - a simple `drbdadm up all` should do the trick.

Now, depending on whether you've got a cluster manager or keep track of your resources manually, there are two different ways again.

- Manually

  ```
  bob# /etc/init.d/drbd start
  ```

- Pacemaker

  ```
  # crm node online bob
  ```

  This should make DRBD connect to the other node, and the resynchronization process will start.

When the two nodes are *UpToDate* on all resources again, you can move your applications to the already upgraded node (here 'bob'), and then follow the same steps on the cluster node still running 8.4.

11.7.10. From DRBD 9 to DRBD 9

If you are already running 9.0, it is sufficient to install new package versions, make the cluster node *standby*, unload/reload the kernel module, start the resources, and make the cluster node *online* again [7: At least that's the state at the time of writing - that's how it has been in the past, and we want to keep it that easy. But who knows? Who can tell? ;)].

These individual steps have been detailed above, so we won't repeat them here.

11.8. Enabling dual-primary mode

Dual-primary mode allows a resource to assume the primary role simultaneously on more than one node. Doing so is possible on either a permanent or a temporary basis.

Dual-primary mode requires that the resource is configured to replicate synchronously (protocol C). Because of this it is latency sensitive, and ill suited for WAN environments.

Additionally, as both resources are always primary, any interruption in the network between nodes will result in a split-brain.

In DRBD 9.0.x Dual-Primary mode is limited to exactly 2 Primaries for the use in live migration.

11.8.1. Permanent dual-primary mode

To enable dual-primary mode, set the `allow-two-primaries` option to `yes` in the `net` section of your resource configuration:

```
resource <resource>
  net {
    protocol C;
    allow-two-primaries yes;
    fencing resource-and-stonith;
  }
  handlers {
    fence-peer "...";
    unfence-peer "...";
  }
  ...
}
```

After that, do not forget to synchronize the configuration between nodes. Run `drbdadm adjust <resource>` on both nodes.

You can now change both nodes to role primary at the same time with `drbdadm primary <resource>`.

You should always implement suitable fencing policies. Using 'allow-two-primaries' without fencing is a very bad idea, even worse than using single-primary without fencing.

11.8.2. Temporary dual-primary mode

To temporarily enable dual-primary mode for a resource normally running in a single-primary configuration, issue the following command:

```
# drbdadm net-options --protocol=C --allow-two-primaries <resource>
```

To end temporary dual-primary mode, run the same command as above but with `--allow-two-primaries=no` (and your desired replication protocol, if applicable).

11.9. Using on-line device verification

11.9.1. Enabling on-line verification

On-line device verification for resources is not enabled by default. To enable it, add the following lines to your resource configuration in `/etc/drbd.conf`:

```
resource <resource>
  net {
    verify-alg <algorithm>;
  }
  ...
}
```

<algorithm> may be any message digest algorithm supported by the kernel crypto API in your system's kernel configuration. Normally, you should be able to choose at least from `sha1`, `md5`, and `crc32c`.

If you make this change to an existing resource, as always, synchronize your `drbd.conf` to the peer, and run `drbdadm adjust <resource>` on both nodes.

11.9.2. Invoking on-line verification

After you have enabled on-line verification, you will be able to initiate a verification run using the following command:

```
# drbdadm verify <resource>
```

When you do so, DRBD starts an online verification run for *<resource>*, and if it detects any blocks that are not in sync, will mark those blocks as such and write a message to the kernel log. Any applications using the device at that time can continue to do so unimpeded, and you may also switch resource roles at will.

If out-of-sync blocks were detected during the verification run, you may resynchronize them using the following commands after verification has completed:

```
# drbdadm disconnect <resource>
# drbdadm connect <resource>
```

11.9.3. Automating on-line verification

Most users will want to automate on-line device verification. This can be easily accomplished. Create a file with the following contents, named `/etc/cron.d/drbd-verify` on **one** of your nodes:

```
42 0 * * 0    root    /sbin/drbdadm verify <resource>
```

This will have `cron` invoke a device verification every Sunday at 42 minutes past midnight; so, if you come into the

office on Monday morning, a quick look at the resource's status would show the result. If your devices are very big, and the \~32 hours were not enough, then you'll notice *VerifyS* or *VerifyT* as connection state, meaning that the `verify` is still in progress.

If you have enabled on-line verification for all your resources (for example, by adding `verify-alg <algorithm>` to the `common` section in `/etc/drbd.d/global_common.conf`), you may also use:

```
42 0 * * 0    root    /sbin/drbdadm verify all
```

11.10. Configuring the rate of synchronization

Normally, one tries to ensure that background synchronization (which makes the data on the synchronization target temporarily inconsistent) completes as quickly as possible. However, it is also necessary to keep background synchronization from hogging all bandwidth otherwise available for foreground replication, which would be detrimental to application performance. Thus, you must configure the synchronization bandwidth to match your hardware — which you may do in a permanent fashion or on-the-fly.

It does not make sense to set a synchronization rate that is higher than the maximum write throughput on your secondary node. You must not expect your secondary node to miraculously be able to write faster than its I/O subsystem allows, just because it happens to be the target of an ongoing device synchronization.

Likewise, and for the same reasons, it does not make sense to set a synchronization rate that is higher than the bandwidth available on the replication network.

11.10.1. Estimating a synchronization speed

A good rule of thumb for this value is to use about 30% of the available replication bandwidth. Thus, if you had an I/O subsystem capable of sustaining write throughput of 400MB/s, and a Gigabit Ethernet network capable of sustaining 110 MB/s network throughput (the network being the bottleneck), you would calculate:

$$110\ MB/s * 0.3 = 33\ MB/s$$

Figure 10. Syncer rate example, 110MB/s effective available bandwidth

Thus, the recommended value for the `rate` option would be `33M`.

By contrast, if you had an I/O subsystem with a maximum throughput of 80MB/s and a Gigabit Ethernet connection (the I/O subsystem being the bottleneck), you would calculate:

$$80\ MB/s * 0.3 = 24\ MB/s$$

Figure 11. Syncer rate example, 80MB/s effective available bandwidth

In this case, the recommended value for the `rate` option would be `24M`.

Similarly, for a storage speed of 800MB/s and a 10Gbe network connection, you would shoot for \~240MB/s synchronization rate.

11.10.2. Variable sync rate configuration

When multiple DRBD resources share a single replication/synchronization network, synchronization with a fixed rate may not be an optimal approach. So, in DRBD 8.4.0 the variable-rate synchronization was enabled by default. In this mode, DRBD uses an automated control loop algorithm to determine, and adjust, the synchronization rate. This algorithm ensures that there is always sufficient bandwidth available for foreground replication, greatly mitigating the impact that background synchronization has on foreground I/O.

The optimal configuration for variable-rate synchronization may vary greatly depending on the available network bandwidth, application I/O pattern and link congestion. Ideal configuration settings also depend on whether DRBD Proxy is in use or not. It may be wise to engage professional consultancy in order to optimally configure this DRBD feature. An *example* configuration (which assumes a deployment in conjunction with DRBD Proxy) is provided below:

```
resource <resource> {
  disk {
    c-plan-ahead 5;
    c-max-rate 10M;
    c-fill-target 2M;
  }
}
```

A good starting value for `c-fill-target` is *BDP * 2*, where *BDP* is your bandwidth-delay-product on the replication link.

For example, when using a 1GBit/s crossover connection, you'll end up with about 200µs latency [8: The rule-of-thumb is using the time reported by `ping`.].
1GBit/s means about 120MB/s; times 200*10^{-6} seconds gives 24000 Byte. Just round that value up to the next MB, and you're good to go.

Another example: a 100MBit WAN connection with 200ms latency means 12MB/s times 0.2s, or about 2.5MB "on the wire". Here a good starting value for `c-fill-target` would be 3MB.

Please see the `drbd.conf` manual page for more details on the other configuration items.

11.10.3. Permanent fixed sync rate configuration

In a few, very restricted situations [9: Like benchmarking.], it might make sense to just use some fixed synchronization rate. In this case, first of all you need to turn the dynamic sync rate controller off, by using `c-plan-ahead 0;`.

Then, the maximum bandwidth a resource uses for background re-synchronization is determined by the `resync-rate` option for a resource. This must be included in the resource's `disk` section in `/etc/drbd.conf`:

```
resource <resource>
  disk {
    resync-rate 40M;
    ...
  }
  ...
}
```

Note that the rate setting is given in *bytes*, not *bits* per second; the default unit is *Kibibyte*, so a value of `4096` would be interpreted as `4MiB`.

This just defines a rate that DRBD tries to achieve. If there is a bottleneck with lower throughput (network, storage speed), the defined speed (aka the "wished-for" performance ;) won't be reached.

11.10.4. Some more hints about synchronization

When some amount of the to-be-synchronized data isn't really in use anymore (eg. because files got deleted while one node wasn't connected), you might benefit from the Trim/Discard support.

Furthermore, `c-min-rate` is easy to misunderstand - it doesn't define a **minimum** synchronization speed, but rather

a limit below which DRBD will not slow down further **on purpose**.
Whether you manage to **reach** that synchronization rate depends on your network and storage speed, network latency (which might be highly variable for shared links), and application IO (which not might be able to do anything about).

11.11. Configuring checksum-based synchronization

Checksum-based synchronization is not enabled for resources by default. To enable it, add the following lines to your resource configuration in `/etc/drbd.conf`:

```
resource <resource>
  net {
    csums-alg <algorithm>;
  }
  ...
}
```

<algorithm> may be any message digest algorithm supported by the kernel crypto API in your system's kernel configuration. Normally, you should be able to choose at least from `sha1`, `md5`, and `crc32c`.

If you make this change to an existing resource, as always, synchronize your `drbd.conf` to the peer, and run `drbdadm adjust <resource>` on both nodes.

11.12. Configuring congestion policies and suspended replication

In an environment where the replication bandwidth is highly variable (as would be typical in WAN replication setups), the replication link may occasionally become congested. In a default configuration, this would cause I/O on the primary node to block, which is sometimes undesirable.

Instead, you may configure DRBD to *suspend* the ongoing replication in this case, causing the Primary's data set to *pull ahead* of the Secondary. In this mode, DRBD keeps the replication channel open — it never switches to disconnected mode — but does not actually replicate until sufficient bandwidth becomes available again.

The following example is for a DRBD Proxy configuration:

```
resource <resource> {
  net {
    on-congestion pull-ahead;
    congestion-fill 2G;
    congestion-extents 2000;
    ...
  }
  ...
}
```

It is usually wise to set both `congestion-fill` and `congestion-extents` together with the `pull-ahead` option.

A good value for `congestion-fill` is 90%

- of the allocated DRBD proxy buffer memory, when replicating over DRBD Proxy, or
- of the TCP network send buffer, in non-DRBD Proxy setups.

A good value for `congestion-extents` is 90% of your configured `al-extents` for the affected resources.

11.13. Configuring I/O error handling strategies

DRBD's strategy for handling lower-level I/O errors is determined by the `on-io-error` option, included in the resource `disk` configuration in `/etc/drbd.conf`:

```
resource <resource> {
  disk {
    on-io-error <strategy>;
    ...
  }
  ...
}
```

You may, of course, set this in the `common` section too, if you want to define a global I/O error handling policy for all resources.

<strategy> may be one of the following options:

`detach`

This is the default and recommended option. On the occurrence of a lower-level I/O error, the node drops its backing device, and continues in diskless mode.

`pass-on`

This causes DRBD to report the I/O error to the upper layers. On the primary node, it is reported to the mounted file system. On the secondary node, it is ignored (because the secondary has no upper layer to report to).

`call-local-io-error`

Invokes the command defined as the local I/O error handler. This requires that a corresponding `local-io-error` command invocation is defined in the resource's `handlers` section. It is entirely left to the administrator's discretion to implement I/O error handling using the command (or script) invoked by `local-io-error`.

Early DRBD versions (prior to 8.0) included another option, `panic`, which would forcibly remove the node from the cluster by way of a kernel panic, whenever a local I/O error occurred. While that option is no longer available, the same behavior may be mimicked via the `local-io-error /call-local-io-error` interface. You should do so only if you fully understand the implications of such behavior.

You may reconfigure a running resource's I/O error handling strategy by following this process:

- Edit the resource configuration in `/etc/drbd.d/<resource>.res`.
- Copy the configuration to the peer node.
- Issue `drbdadm adjust <resource>` on both nodes.

11.14. Configuring replication traffic integrity checking

Replication traffic integrity checking is not enabled for resources by default. To enable it, add the following lines to your resource configuration in `/etc/drbd.conf`:

```
resource <resource>
  net {
    data-integrity-alg <algorithm>;
  }
  ...
}
```

<algorithm> may be any message digest algorithm supported by the kernel crypto API in your system's kernel configuration. Normally, you should be able to choose at least from `sha1`, `md5`, and `crc32c`.

If you make this change to an existing resource, as always, synchronize your `drbd.conf` to the peer, and run `drbdadm adjust <resource>` on both nodes.

This feature is not intended for production use. Enable only if you need to diagnose data corruption problems, and want to see whether the transport path (network hardware, drivers, switches) might be at fault!

11.15. Resizing resources

11.15.1. Growing on-line

If the backing block devices can be grown while in operation (online), it is also possible to increase the size of a DRBD device based on these devices during operation. To do so, two criteria must be fulfilled:

1. The affected resource's backing device must be one managed by a logical volume management subsystem, such as LVM.
2. The resource must currently be in the *Connected* connection state.

Having grown the backing block devices on all nodes, ensure that only one node is in primary state. Then enter on one node:

```
# drbdadm resize <resource>
```

This triggers a synchronization of the new section. The synchronization is done from the primary node to the secondary node.

If the space you're adding is clean, you can skip syncing the additional space by using the `--assume-clean` option.

```
# drbdadm -- --assume-clean resize <resource>
```

11.15.2. Growing off-line

When the backing block devices on both nodes are grown while DRBD is inactive, and the DRBD resource is using external meta data, then the new size is recognized automatically. No administrative intervention is necessary. The DRBD device will have the new size after the next activation of DRBD on both nodes and a successful establishment of a network connection.

If however the DRBD resource is configured to use internal meta data, then this meta data must be moved to the end of the grown device before the new size becomes available. To do so, complete the following steps:

This is an advanced procedure. Use at your own discretion.

- Unconfigure your DRBD resource:

```
# drbdadm down <resource>
```

- Save the meta data in a text file prior to resizing:

```
# drbdadm dump-md <resource> > /tmp/metadata
```

You must do this on both nodes, using a separate dump file for every node. **Do not** dump the meta data on one node, and simply copy the dump file to the peer. **This. will. not. work.**

- Grow the backing block device on both nodes.
- Adjust the size information (`la-size-sect`) in the file `/tmp/metadata` accordingly, on both nodes. Remember that `la-size-sect` must be specified in sectors.
- Re-initialize the metadata area:

```
# drbdadm create-md <resource>
```

- Re-import the corrected meta data, on both nodes:

```
# drbdmeta_cmd=$(drbdadm -d dump-md <resource>)
# ${drbdmeta_cmd/dump-md/restore-md} /tmp/metadata
Valid meta-data in place, overwrite? [need to type 'yes' to confirm]
yes
Successfully restored meta data
```

This example uses `bash` parameter substitution. It may or may not work in other shells. Check your `SHELL` environment variable if you are unsure which shell you are currently using.

- Re-enable your DRBD resource:

```
# drbdadm up <resource>
```

- On one node, promote the DRBD resource:

```
# drbdadm primary <resource>
```

- Finally, grow the file system so it fills the extended size of the DRBD device.

11.15.3. Shrinking on-line

Online shrinking is only supported with external metadata.

Before shrinking a DRBD device, you *must* shrink the layers above DRBD, i.e. usually the file system. Since DRBD cannot ask the file system how much space it actually uses, you have to be careful in order not to cause data loss.

Whether or not the *filesystem* can be shrunk on-line depends on the filesystem being used. Most filesystems do not support on-line shrinking. XFS does not support shrinking at all.

To shrink DRBD on-line, issue the following command *after* you have shrunk the file system residing on top of it:

```
# drbdadm resize --size=<new-size> <resource>
```

You may use the usual multiplier suffixes for *<new-size>* (K, M, G etc.). After you have shrunk DRBD, you may also

shrink the containing block device (if it supports shrinking).

It might be a good idea to issue `drbdadm resize <resource>` after resizing the lower level device, so that the DRBD metadata **really** gets written into the expected space at the end of the volume.

11.15.4. Shrinking off-line

If you were to shrink a backing block device while DRBD is inactive, DRBD would refuse to attach to this block device during the next attach attempt, since it is now too small (in case external meta data is used), or it would be unable to find its meta data (in case internal meta data is used). To work around these issues, use this procedure (if you cannot use on-line shrinking):

This is an advanced procedure. Use at your own discretion.

- Shrink the file system from one node, while DRBD is still configured.
- Unconfigure your DRBD resource:

```
# drbdadm down <resource>
```

- Save the meta data in a text file prior to shrinking:

```
# drbdadm dump-md <resource> > /tmp/metadata
```

You must do this on both nodes, using a separate dump file for every node. **Do not** dump the meta data on one node, and simply copy the dump file to the peer. **This. will. not. work.**

- Shrink the backing block device on both nodes.
- Adjust the size information (`la-size-sect`) in the file `/tmp/metadata` accordingly, on both nodes. Remember that `la-size-sect` must be specified in sectors.
- **Only if you are using internal metadata** (which at this time have probably been lost due to the shrinking process), re-initialize the metadata area:

```
# drbdadm create-md <resource>
```

- Re-import the corrected meta data, on both nodes:

```
# drbdmeta_cmd=$(drbdadm -d dump-md <resource>)
# ${drbdmeta_cmd/dump-md/restore-md} /tmp/metadata
Valid meta-data in place, overwrite? [need to type 'yes' to confirm]
yes
Successfully restored meta data
```

This example uses `bash` parameter substitution. It may or may not work in other shells. Check your `SHELL` environment variable if you are unsure which shell you are currently using.

- Re-enable your DRBD resource:

```
# drbdadm up <resource>
```

11.16. Disabling backing device flushes

You should only disable device flushes when running DRBD on devices with a battery-backed write cache (BBWC). Most storage controllers allow to automatically disable the write cache when the battery is depleted, switching to write-through mode when the battery dies. It is strongly recommended to enable such a feature.

Disabling DRBD's flushes when running without BBWC, or on BBWC with a depleted battery, is *likely to cause data loss* and should not be attempted.

DRBD allows you to enable and disable backing device flushes separately for the replicated data set and DRBD's own meta data. Both of these options are enabled by default. If you wish to disable either (or both), you would set this in the `disk` section for the DRBD configuration file, `/etc/drbd.conf`.

To disable disk flushes for the replicated data set, include the following line in your configuration:

```
resource <resource>
  disk {
    disk-flushes no;
    ...
  }
  ...
}
```

To disable disk flushes on DRBD's meta data, include the following line:

```
resource <resource>
  disk {
    md-flushes no;
    ...
  }
  ...
}
```

After you have modified your resource configuration (and synchronized your `/etc/drbd.conf` between nodes, of course), you may enable these settings by issuing this command on both nodes:

```
# drbdadm adjust <resource>
```

In case only one of the serves has a BBWC [10: For example, in the DR site you might be using different hardware, right?], you should move the setting into a host section, like this:

```
resource <resource> {
  disk {
    ... common settings ...
  }

  on host-1 {
    disk {
      md-flushes no;
    }
    ...
  }
  ...
}
```

11.17. Configuring split brain behavior

11.17.1. Split brain notification

DRBD invokes the `split-brain` handler, if configured, at any time split brain is *detected*. To configure this handler, add the following item to your resource configuration:

```
resource <resource>
  handlers {
    split-brain <handler>;
    ...
  }
  ...
}
```

<*handler*> may be any executable present on the system.

The DRBD distribution contains a split brain handler script that installs as `/usr/lib/drbd/notify-split-brain.sh`. It simply sends a notification e-mail message to a specified address. To configure the handler to send a message to `root@localhost` (which is expected to be an email address that forwards the notification to a real system administrator), configure the `split-brain handler` as follows:

```
resource <resource>
  handlers {
    split-brain "/usr/lib/drbd/notify-split-brain.sh root";
    ...
  }
  ...
}
```

After you have made this modification on a running resource (and synchronized the configuration file between nodes), no additional intervention is needed to enable the handler. DRBD will simply invoke the newly-configured handler on the next occurrence of split brain.

11.17.2. Automatic split brain recovery policies

Configuring DRBD to automatically resolve data divergence siutaions resulting from split-brain (or other) scenarios is configuring for potential **automatic data loss**. Understand the implications, and don't do it if you don't mean to.

You rather want to look into fencing policies, quorum settings, cluster manager integration, and redundant cluster manager communication links to **avoid** data divergence in the first place.

In order to be able to enable and configure DRBD's automatic split brain recovery policies, you must understand that DRBD offers several configuration options for this purpose. DRBD applies its split brain recovery procedures based on the number of nodes in the Primary role at the time the split brain is detected. To that end, DRBD examines the following keywords, all found in the resource's `net` configuration section:

`after-sb-0pri`

Split brain has just been detected, but at this time the resource is not in the Primary role on any host. For this option, DRBD understands the following keywords:

- `disconnect`: Do not recover automatically, simply invoke the `split-brain` handler script (if configured), drop the connection and continue in disconnected mode.
- `discard-younger-primary`: Discard and roll back the modifications made on the host which assumed the Primary role last.
- `discard-least-changes`: Discard and roll back the modifications on the host where fewer changes occurred.
- `discard-zero-changes`: If there is any host on which no changes occurred at all, simply apply all modifications made on the other and continue.

`after-sb-1pri`

Split brain has just been detected, and at this time the resource is in the Primary role on one host. For this option, DRBD understands the following keywords:

- `disconnect`: As with `after-sb-0pri`, simply invoke the `split-brain` handler script (if configured), drop the connection and continue in disconnected mode.
- `consensus`: Apply the same recovery policies as specified in `after-sb-0pri`. If a split brain victim can be selected after applying these policies, automatically resolve. Otherwise, behave exactly as if `disconnect` were specified.
- `call-pri-lost-after-sb`: Apply the recovery policies as specified in `after-sb-0pri`. If a split brain victim can be selected after applying these policies, invoke the `pri-lost-after-sb` handler on the victim node. This handler must be configured in the `handlers` section and is expected to forcibly remove the node from the cluster.
- `discard-secondary`: Whichever host is currently in the Secondary role, make that host the split brain victim.

`after-sb-2pri`

Split brain has just been detected, and at this time the resource is in the Primary role on both hosts. This option accepts the same keywords as `after-sb-1pri` except `discard-secondary` and `consensus`.

DRBD understands additional keywords for these three options, which have been omitted here because they are very rarely used. Refer to the man page of `drbd.conf` for details on split brain recovery keywords not discussed here.

For example, a resource which serves as the block device for a GFS or OCFS2 file system in dual-Primary mode may have its recovery policy defined as follows:

```
resource <resource> {
  handlers {
    split-brain "/usr/lib/drbd/notify-split-brain.sh root"
    ...
  }
  net {
    after-sb-0pri discard-zero-changes;
    after-sb-1pri discard-secondary;
    after-sb-2pri disconnect;
    ...
  }
  ...
}
```

11.18. Creating a stacked three-node setup

A three-node setup involves one DRBD device *stacked* atop another.

Stacking is deprecated in DRBD version 9.x, as more nodes can be implemented on a single level. See Defining network connections for details.

11.18.1. Device stacking considerations

The following considerations apply to this type of setup:

- The stacked device is the active one. Assume you have configured one DRBD device `/dev/drbd0`, and the stacked device atop it is `/dev/drbd10`, then `/dev/drbd10` will be the device that you mount and use.
- Device meta data will be stored twice, on the underlying DRBD device *and* the stacked DRBD device. On the stacked device, you must always use internal meta data. This means that the effectively available storage area on a stacked device is slightly smaller, compared to an unstacked device.
- To get the stacked upper level device running, the underlying device must be in the primary role.
- To be able to synchronize the backup node, the stacked device on the active node must be up and in the primary role.

11.18.2. Configuring a stacked resource

In the following example, nodes are named 'alice', 'bob', and 'charlie', with 'alice' and 'bob' forming a two-node cluster, and 'charlie' being the backup node.

```
resource r0 {
  protocol C;
  device    /dev/drbd0;
  disk      /dev/sda6;
  meta-disk internal;

  on alice {
    address    10.0.0.1:7788;
  }

  on bob {
    address   10.0.0.2:7788;
  }
}

resource r0-U {
  protocol A;

  stacked-on-top-of r0 {
    device     /dev/drbd10;
    address    192.168.42.1:7789;
  }

  on charlie {
    device     /dev/drbd10;
    disk       /dev/hda6;
    address    192.168.42.2:7789; # Public IP of the backup node
    meta-disk  internal;
  }
}
```

As with any `drbd.conf` configuration file, this must be distributed across all nodes in the cluster — in this case, three nodes. Notice the following extra keyword not found in an unstacked resource configuration:

`stacked-on-top-of`

This option informs DRBD that the resource which contains it is a stacked resource. It replaces one of the `on` sections normally found in any resource configuration. Do not use `stacked-on-top-of` in an lower-level resource.

It is not a requirement to use Protocol A for stacked resources. You may select any of DRBD's replication protocols depending on your application.

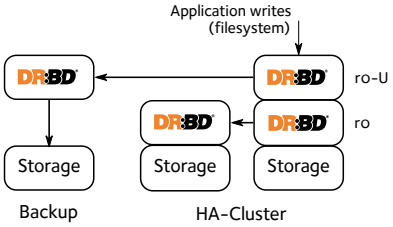


Figure 12. Single stacked setup

11.18.3. Enabling stacked resources

To enable a stacked resource, you first enable its lower-level resource and promote it:

```
drbdadm up r0
drbdadm primary r0
```

As with unstacked resources, you must create DRBD meta data on the stacked resources. This is done using the following command:

```
# drbdadm create-md --stacked r0-U
```

Then, you may enable the stacked resource:

```
# drbdadm up --stacked r0-U
# drbdadm primary --stacked r0-U
```

After this, you may bring up the resource on the backup node, enabling three-node replication:

```
# drbdadm create-md r0-U
# drbdadm up r0-U
```

In order to automate stacked resource management, you may integrate stacked resources in your cluster manager configuration. See Using stacked DRBD resources in Pacemaker clusters for information on doing this in a cluster managed by the Pacemaker cluster management framework.

11.19. Permanently diskless nodes

A node might be permanently diskless in DRBD. Here is a configuration example showing a resource with 3 diskfull nodes (servers) and one permanently diskless node (client).

```
resource kvm-mail {
  device      /dev/drbd6;
  disk        /dev/vg/kvm-mail;
  meta-disk   internal;

  on store1 {
    address   10.1.10.1:7006;
    node-id   0;
  }
  on store2 {
    address   10.1.10.2:7006;
    node-id   1;
  }
  on store3 {
    address   10.1.10.3:7006;
    node-id   2;
  }

  on for-later-rebalancing {
    address   10.1.10.4:7006;
    node-id   3;
  }

  # DRBD "client"
  floating 10.1.11.6:8006 {
    disk      none;
    node-id   4;
  }

  # rest omitted for brevity
  ...
}
```

For permanently diskless nodes no bitmap slot gets allocated. For such nodes the diskless status is displayed in green color since it is not an error or unexpected state.

This *DRBD client* is an easy way to get data over the wire, but it doesn't have any of the advanced iSCSI features like *Persistent Reservations*.
If your setup has only basic I/O needs, like *read*, *write*, *trim/discard* and perhaps *resize* (eg. for a virtual machine), you should be fine.

11.20. Data rebalancing

Given the (example) policy that data needs to be available on 3 nodes, you need at least 3 servers for your setup.

Now, as your storage demands grow, you will encounter the need for additional servers. Rather than having to buy 3 more servers at the same time, you can *rebalance* your data across a single additional node.

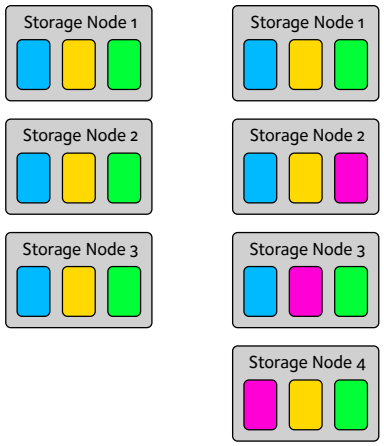


Figure 13. DRBD data rebalancing

In the figure above you can see the *before* and *after* states: from 3 nodes with three 25TiB volumes each (for a net 75TiB), to 4 nodes, with net 100TiB.

To redistribute the data across your cluster you have to choose a *new* node, and one where you want to remove this DRBD resource.
Please note that removing the resource from a currently *active* node (ie. where DRBD is *Primary*) will involve either migrating the service or running this resource on this node as a DRBD client; it's easier to choose a node in *Secondary* role. (Of course, that might not always be possible.)

11.20.1. Prepare a bitmap slot

You will need to have a free bitmap slot for temporary use, on each of the nodes that have the resource that is to be moved.

You can allocate one more at `drbdadm create-md` time, or simply put a placeholder in your configuration, so that `drbdadm` sees that it should reserve one more slot:

```
resource r0 {
  ...
  on for-later-rebalancing {
    address   10.254.254.254:65533;
    node-id   3;
  }
}
```

If you need to make that slot available during live use, you will have to

1. dump the metadata,
2. enlarge the metadata space,
3. edit the dumpfile,
4. load the changed metadata.

In a future version `drbdadm` will have a shortcut for you; most probably you'll be able to say `drbdadm resize --peers N` and have the kernel rewrite the metadata for you.

11.20.2. Preparing and activating the new node

First of all you have to create the underlying storage volume on the new node (using eg. `lvcreate`). Then the placeholder in the configuration can be filled with the correct host name, address, and storage path. Now copy the resource configuration to all relevant nodes.

On the new node initialize the meta-data (once) by doing

```
# drbdadm create-md <resource>
v09 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block sucessfully created.
```

11.20.3. Starting the initial sync

Now the new node needs to get the data.

This is done by defining the network connection on the existing nodes via

```
# drbdadm adjust <resource>
```

and starting the DRBD device on the new node via

```
# drbdadm up <resource>
```

11.20.4. Check connectivity

At this time please do a

```
# drbdadm status <resource>
```

on the new node, and check that *all* other nodes are connected.

11.20.5. After the initial sync

As soon as the new host is *UpToDate*, one of the other nodes in the configuration can be renamed to `for-later-rebalancing`, and kept for another migration.

Perhaps you want to comment the section; although that has the risk that doing a `drbdadm create-md` for a new node has too few bitmap slots for the next rebalancing.
It might be easier to use a reserved (unused) IP address and host name.

Copy the changed configuration around again, and use it by running

```
# drbdadm adjust <resource>
```

on all nodes.

11.20.6. Cleaning up

On the one node that had the data up to now, but isn't used anymore for this resource, you can now take the DRBD device down by starting

```
# drbdadm down <resource>
```

Now the lower level storage device isn't used anymore, and can either be re-used for other purposes or, if it is a logical volume, its space can be returned to the volume group via `lvremove`.

11.20.7. Conclusion and further steps

One of the resources has been migrated to the new node. The same could be done for one or more other resources, to free space on two or three nodes in the existing cluster.

Then new resources can be configured, as there're enough nodes with free space to achieve 3-way redundancy again.

Still, you might want to compare the steps above with the procedure when using DRBD Manage: Rebalancing data with DRBD Manage...

11.21. Configuring quorum

In order to avoid split brain or diverging data of replicas one has to configure fencing. All the options for fencing rely on redundant communication in the end. That might be in the form of a management network that connects the nodes to the IPMI network interfaces of the peer machines. In case of the crm-fence-peer script it is necessary that Pacemakers communication stays available when DRBD's network link breaks.

The quorum mechanism on the other hand takes a completely difference approach. The basic idea is that a cluster partition may only modify the replicated data set if the number of nodes that can communicate is greater then the half of the overall number of nodes. A node of such a partition *has quorum*. On the other hand a node does not have quorum needs to guarantee that the replicated data set it not touched, that it does not create a diverging data set.

The quorum implementation in DRBD gets enabled by setting the `quorum` resource option to `majority`, `all` or a numeric value. Where `majority` selects the behavior that was described in the previous paragraph.

11.21.1. Guaranteed minimal redundancy

By default every node with a disk gets a vote in the quorum election. In other words only diskless nodes do not count. So a partition with two *Inconsistent* disks gets quorum, while a partition with one *UpToDate* node will have quorum in a 3 node cluster. By configuring `quorum-minimum-redundancy` this behavior can be changed so that only nodes that are `UpToDate` have a vote in the quorum election. The option takes the same arguments as the `quorum` option.

With this option you express that you rather want to wait until eventually necessary resync operations finish before any services start. So in a way you prefer that the minimal redundancy of your data is guaranteed over the availability of your service. Financial data and services is an example that comes to mind.

Consider following example for a 5 node cluster. It requires a partitions to have at least 3 nodes, and two of them must be `UpToDate`:

```
resource quorum-demo {
  quorum majority;
  quorum-minimum-redundancy 2;
  ...
}
```

11.21.2. Actions on loss of quorum

When a node that is running the service loses quorum it needs to cease write-operations on the data set immediately. That means that IO immediately starts to complete all IO requests with errors. Usually that means that a graceful shutdown is not possible, since that would require more modifications to the data set. The IO errors propagate from the block level to the file system and from the file system to the user space application(s).

Ideally the application simply terminates in case of IO errors. This allows then Pacemaker to unmount the filesystem and to demote the DRBD resource to secondary role. If that is true you should set the `on-no-quorum` resource option to `io-error`. Here is an example:

```
resource quorum-demo {
  quorum majority;
  on-no-quorum io-error;
  ...
}
```

If you application does not terminate on the first IO error, you can choose to freeze IO instead and to reboot the node. Here is a configuration example:

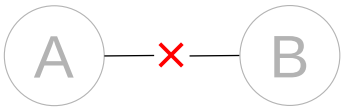
```
resource quorum-demo {
  quorum majority;
  on-no-quorum suspend-io;
  ...

  handlers {
    quorum-lost "echo b > /proc/sysrq-trigger";
  }
  ...
}
```

11.21.3. Using a diskless node as a tiebreaker

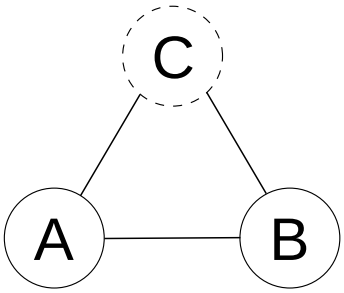
A diskless node with connections to all nodes in a cluster can be utilized to break ties in the quorum negotiation process.

Consider the following two-node cluster, where node A is the primary and node B is a secondary:

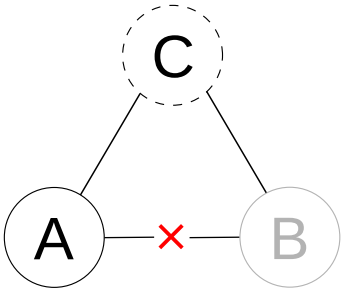
As soon as the connection between the two nodes is interrupted, they lose quorum and the application on top of the cluster cannot write data anymore.



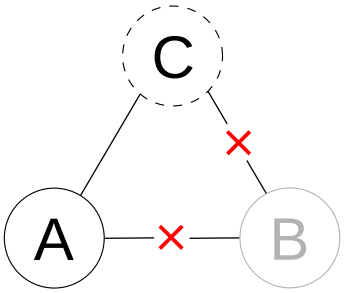
Now if we add a third node, C, to the cluster and configure it as diskless, we can take advantage of the tiebreaker mechanism.



In this case, when the primary and secondary nodes lose connection, they can still "see" the diskless tiebreaker. Because of this, the primary can continue working, while the secondary demotes its disk to Outdated and the service can therefore not be migrated there.

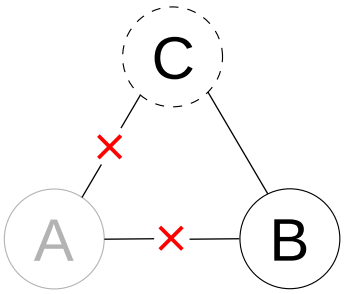


There are a few special cases in case two connections fail. Consider the following scenario:



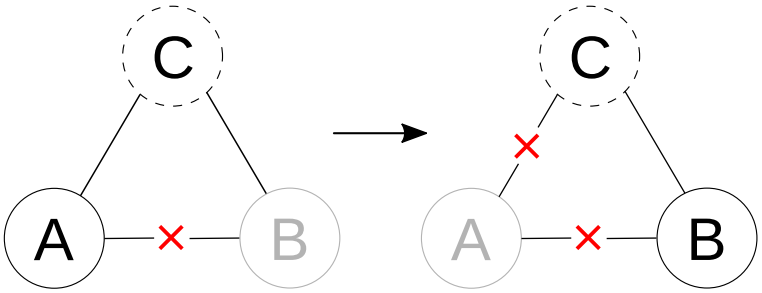
In this case, the tiebreaker node forms a partition with the primary node. The primary therefore keeps quorum, while the secondary becomes outdated. Note that the secondarys disk state will be "UpToDate", but regardless it cannot be promoted to primary because it lacks quorum.

Let's consider the possibility of the primary losing connection to the tiebreaker:



In this case, the primary becomes unusable and goes into a "quorum suspended" state. This effectively results in the application on top of DRBD receiving I/O errors. A cluster manager could then promote node B to primary and keep the service running there instead.

In case the diskless tiebreaker "switches sides", we need to avoid data divergence. Consider this scenario:

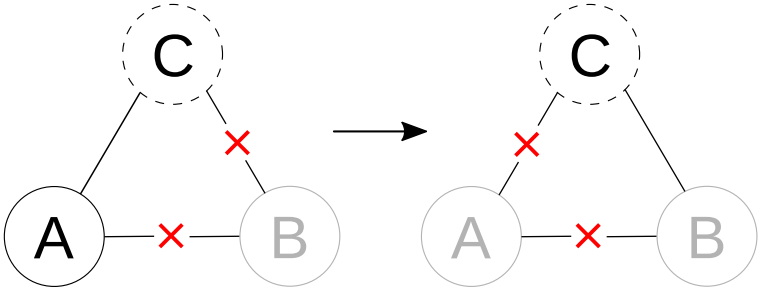


The connection between the primary and secondary has failed, and the application is continuing to run on the primary, when the primary suddenly loses its connection to the diskless node.

In this case, no node can be promoted to primary and the cluster cannot continue to operate.

Protecting from data divergence always takes priority over ensuring service availability.

Let's look at another scenario:



Here, the application is running on the primary, while the secondary is unavailable. Then, the tiebreaker first loses connection to the primary, and then reconnects to the secondary. It is important to note here that **a node that has lost quorum cannot regain quorum by connecting to a diskless node**. Thus, in this case, no node has quorum and the cluster halts.

Chapter 12. Using DRBD Proxy

12.1. DRBD Proxy deployment considerations

The DRBD Proxy processes can either be located directly on the machines where DRBD is set up, or they can be placed on distinct dedicated servers. A DRBD Proxy instance can serve as a proxy for multiple DRBD devices distributed across multiple nodes.

DRBD Proxy is completely transparent to DRBD. Typically you will expect a high number of data packets in flight, therefore the activity log should be reasonably large. Since this may cause longer re-sync runs after the crash of a primary node, it is recommended to enable DRBD's `csums-alg` setting.

For more information about the rationale for the DRBD Proxy, please see the feature explanation Long-distance replication via DRBD Proxy.

The DRBD Proxy 3 uses several kernel features that are only available since 2.6.26, so running it on older systems (eg. RHEL 5) is not possible; here we can still provide DRBD Proxy 1 packages, though [11: The v1 uses a different scheduling model and will therefore not reach the same performance as v3; so even if your production setup is still RHEL 5, perhaps you can run one RHEL 6/7 VM in each data center?].

12.2. Installation

To obtain DRBD Proxy, please contact your Linbit sales representative. Unless instructed otherwise, please always use the most recent DRBD Proxy release.

To install DRBD Proxy on Debian and Debian-based systems, use the dpkg tool as follows (replace version with your DRBD Proxy version, and architecture with your target architecture):

```
# dpkg -i drbd-proxy_3.2.2_amd64.deb
```

To install DRBD Proxy on RPM based systems (like SLES or RHEL) use the rpm tool as follows (replace version with your DRBD Proxy version, and architecture with your target architecture):

```
# rpm -i drbd-proxy-3.2.2-1.x86_64.rpm
```

Also install the DRBD administration program drbdadm since it is required to configure DRBD Proxy.

This will install the DRBD proxy binaries as well as an init script which usually goes into `/etc/init.d`. Please always use the init script to start/stop DRBD proxy since it also configures DRBD Proxy using the `drbdadm` tool.

12.3. License file

When obtaining a license from Linbit, you will be sent a DRBD Proxy license file which is required to run DRBD Proxy. The file is called `drbd-proxy.license`, it must be copied into the `/etc` directory of the target machines, and be owned by the user/group `drbdpxy`.

```
# cp drbd-proxy.license /etc/
```

12.4. Configuration using LINSTOR

DRBD Proxy can be configured using LINSTOR as described in DRBD Proxy with LINSTOR.

12.5. Configuration using resource files

DRBD Proxy can also be configured by editing resource files. It is configured by an additional section called `proxy` and additional `proxy on` sections within the host sections.

Below is a DRBD configuration example for proxies running directly on the DRBD nodes:

```
resource r0 {
    protocol A;
    device     /dev/drbd15;
    disk       /dev/VG/r0;
    meta-disk  internal;

    proxy {
        memlimit 512M;
        plugin {
            zlib level 9;
        }
    }

    on alice {
        address 127.0.0.1:7915;
        proxy on alice {
            inside 127.0.0.1:7815;
            outside 192.168.23.1:7715;
        }
    }

    on bob {
        address 127.0.0.1:7915;
        proxy on bob {
            inside 127.0.0.1:7815;
            outside 192.168.23.2:7715;
        }
    }
}
```

The `inside` IP address is used for communication between DRBD and the DRBD Proxy, whereas the `outside` IP address is used for communication between the proxies. The latter channel might have to be allowed in your firewall setup.

12.6. Controlling DRBD Proxy

`drbdadm` offers the `proxy-up` and `proxy-down` subcommands to configure or delete the connection to the local DRBD Proxy process of the named DRBD resource(s). These commands are used by the `start` and `stop` actions which `/etc/init.d/drbdproxy` implements.

The DRBD Proxy has a low level configuration tool, called `drbd-proxy-ctl`. When called without any option it operates in interactive mode.

To pass a command directly, avoiding interactive mode, use the `-c` parameter followed by the command.

To display the available commands use:

```
# drbd-proxy-ctl -c "help"
```

Note the double quotes around the command being passed.

Here is a list of commands; while the first few ones are typically only used indirectly (via `drbdadm proxy-up` resp. `drbdadm proxy-down`), the latter ones give various status informations.

`add connection <name> lots of arguments`
Creates a communication path. As this is run via `drbdadm proxy-up` the long list of arguments is omitted here.

`del connection <name>`
Removes a communication path.

`set memlimit <name> <memlimit-in-bytes>`
Sets the memory limit for a connection; this can only be done when setting it up afresh, changing it during runtime is not possible.
This command understands the usual units `k`, `M`, and `G`.

`show`
Shows currently configured communication paths.

`show memusage`
Shows memory usage of each connection.

As an example,

```
# watch -n 1 'drbd-proxy-ctl -c "show memusage"'
```

monitors memory usage. Please note that the quotes are required as listed above.

`show [h]subconnections`
Shows currently established individual connections together with some stats. With `h` outputs bytes in human readable format.

`show [h]connections`
Shows currently configured connections and their states With `h` outputs bytes in human readable format.

The column `Status` will show one of these states:

- *Off*: No communication to the remote DRBD Proxy process.
- *Half-up*: The connection to the remote DRBD Proxy could be established; the Proxy ⇒ DRBD paths are not up yet.
- *DRBD-conn*: The first few packets are being pushed across the connection; but still eg. a Split-Brain situation might sever it again.
- *Up*: The DRBD connection is fully established.

`shutdown`
Shuts down the `drbd-proxy` program. Attention: this unconditionally terminates any DRBD connections using the DRBD proxy.

`quit`
Exits the client program (closes the control connection), but leaves the DRBD proxy running.

`print statistics`

This prints detailed statistics for the currently active connections, in an easily parseable format. Use this for integration to your monitoring solution!

While the commands above are only accepted from UID 0 (ie., the `root` user), this one can be used by any user (provided that unix permissions allow access on the proxy socket at `/var/run/drbd-proxy/drbd-proxy-ctl.socket`); see the init script at `/etc/init.d/drbdproxy` about setting the rights.

12.7. About DRBD Proxy plugins

Since DRBD Proxy version 3 the proxy allows to enable a few specific plugins for the WAN connection.
The currently available plugins are `lz4`, `zlib` and `lzma` (all software compression), and `aha` (hardware compression support, see http://www.aha.com/data-compression/).

`lz4` is a very fast compression algorithm; the data typically gets compressed down by 1:2 to 1:4, half- to two-thirds of the bandwidth can be saved.

The `zlib` plugin uses the GZIP algorithm for compression; it uses a bit more CPU than `lz4`, but gives a ratio of 1:3 to 1:5.

The `lzma` plugin uses the `liblzma2` library. It can use dictionaries of several hundred MiB; these allow for very efficient delta-compression of repeated data, even for small changes. `lzma` needs much more CPU and memory, but results in much better compression than `zlib` — real-world tests with a VM sitting on top of DRBD gave ratios of 1:10 to 1:40. The `lzma` plugin has to be enabled in your license.

`aha` uses hardware compression cards, like the AHA367PCIe (10Gbit/sec) or AHA372 (20GBit/sec); this is the fastest compression for contemporary hardware.
You will need a special flag in your license file to enable this plugin.

Please contact LINBIT to find the best settings for your environment - it depends on the CPU (speed, number of threads), available memory, input and available output bandwidth, and expected IO spikes. Having a week of `sysstat` data already available helps in determining the configuration, too.

Please note that the older `compression on` in the `proxy` section is deprecated, and will be removed in a future release.
Currently it is treated as `zlib level 9`.

12.7.1. Using a WAN Side Bandwidth Limit

The experimental `bwlimit` option of DRBD Proxy is broken. Do not use it, as it may cause applications on DRBD to block on IO. It will be removed.

Instead use the Linux kernel's traffic control framework to limit bandwidth consumed by proxy on the WAN side.

In the following example you would need to replace the interface name, the source port and the ip address of the peer.

```
# tc qdisc add dev eth0 root handle 1: htb default 1
# tc class add dev eth0 parent 1: classid 1:1 htb rate 1gbit
# tc class add dev eth0 parent 1:1 classid 1:10 htb rate 500kbit
# tc filter add dev eth0 parent 1: protocol ip prio 16 u32 \
        match ip sport 7000 0xffff \
        match ip dst 192.168.47.11 flowid 1:10
# tc filter add dev eth0 parent 1: protocol ip prio 16 u32 \
        match ip dport 7000 0xffff \
        match ip dst 192.168.47.11 flowid 1:10
```

You can remove this bandwidth limitation with

```
# tc qdisc del dev eth0 root handle 1
```

12.8. Troubleshooting

DRBD proxy logs via syslog using the LOG_DAEMON facility. Usually you will find DRBD Proxy messages in `/var/log/daemon.log`.

Enabling debug mode in DRBD Proxy can be done with the following command.

```
# drbd-proxy-ctl -c 'set loglevel debug'
```

For example, if proxy fails to connect it will log something like `Rejecting connection because I can't connect on the other side`. In that case, please check if DRBD is running (not in StandAlone mode) on both nodes and if both proxies are running. Also double-check your configuration.

Chapter 13. Troubleshooting and error recovery

This chapter describes tasks to be performed in the event of hardware or system failures.

13.1. Dealing with hard drive failure

How to deal with hard drive failure depends on the way DRBD is configured to handle disk I/O errors (see Disk error handling strategies), and on the type of meta data configured (see DRBD meta data).

For the most part, the steps described here apply only if you run DRBD directly on top of physical hard drives. They generally do not apply in case you are running DRBD layered on top of

- an MD software RAID set (in this case, use `mdadm` to manage drive replacement),
- device-mapper RAID (use `dmraid`),
- a hardware RAID appliance (follow the vendor's instructions on how to deal with failed drives),
- some non-standard device-mapper virtual block devices (see the device mapper documentation).

13.1.1. Manually detaching DRBD from your hard drive

If DRBD is configured to pass on I/O errors (not recommended), you must first detach the DRBD resource, that is, disassociate it from its backing storage:

```
# drbdadm detach <resource>
```

By running the `drbdadm status` or the `drbdadm dstate` command, you will now be able to verify that the resource is now in *diskless mode*:

```
# drbdadm status <resource>
<resource> role:Primary
  volume:0 disk:Diskless
  <peer> role:Secondary
    volume:0 peer-disk:UpToDate
# drbdadm dstate <resource>
Diskless/UpToDate
```

If the disk failure has occured on your primary node, you may combine this step with a switch-over operation.

13.1.2. Automatic detach on I/O error

If DRBD is configured to automatically detach upon I/O error (the recommended option), DRBD should have automatically detached the resource from its backing storage already, without manual intervention. You may still use the `drbdadm status` command to verify that the resource is in fact running in diskless mode.

13.1.3. Replacing a failed disk when using internal meta data

If using internal meta data, it is sufficient to bind the DRBD device to the new hard disk. If the new hard disk has to be addressed by another Linux device name than the defective disk, the DRBD configuration file has to be modified accordingly.

This process involves creating a new meta data set, then re-attaching the resource:

```
# drbdadm create-md <resource>
v08 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block sucessfully created.

# drbdadm attach <resource>
```

Full synchronization of the new hard disk starts instantaneously and automatically. You will be able to monitor the synchronization's progress via `drbdadm status --verbose`, as with any background synchronization.

13.1.4. Replacing a failed disk when using external meta data

When using external meta data, the procedure is basically the same. However, DRBD is not able to recognize independently that the hard drive was swapped, thus an additional step is required.

```
# drbdadm create-md <resource>
v08 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block sucessfully created.

# drbdadm attach <resource>
# drbdadm invalidate <resource>
```

Make sure to run `drbdadm invalidate` on the node **without** good data; this command will cause the local contents to be overwritten with data from the peers, so running this command on the wrong node might lose data!

Here, the `drbdadm invalidate` command triggers synchronization. Again, sync progress may be observed via `drbdadm status --verbose`.

13.2. Dealing with node failure

When DRBD detects that its peer node is down (either by true hardware failure or manual intervention), DRBD changes its connection state from *Connected* to *Connecting* and waits for the peer node to re-appear. The DRBD resource is then said to operate in *disconnected mode*. In disconnected mode, the resource and its associated block device are fully usable, and may be promoted and demoted as necessary, but no block modifications are being replicated to the peer node. Instead, DRBD stores which blocks are being modified while disconnected, on a per-peer basis.

13.2.1. Dealing with temporary secondary node failure

If a node that currently has a resource in the secondary role fails temporarily (due to, for example, a memory problem that is subsequently rectified by replacing RAM), no further intervention is necessary — besides the obvious necessity to repair the failed node and bring it back online. When that happens, the two nodes will simply re-establish connectivity upon system start-up. After this, DRBD synchronizes all modifications made on the primary node in the meantime to the secondary node.

At this point, due to the nature of DRBD's re-synchronization algorithm, the resource is briefly inconsistent on the secondary node. During that short time window, the secondary node can not switch to the Primary role if the peer is unavailable. Thus, the period in which your cluster is not redundant consists of the actual secondary node down time, plus the subsequent re-synchronization.

Please note that with DRBD 9 more than two nodes can be connected for each resource, so for eg. 4 nodes a single failing secondary still leaves two other secondaries available for failover.

13.2.2. Dealing with temporary primary node failure

From DRBD's standpoint, failure of the primary node is almost identical to a failure of the secondary node. The surviving node detects the peer node's failure, and switches to disconnected mode. DRBD does *not* promote the surviving node to the primary role; it is the cluster management application's responsibility to do so.

When the failed node is repaired and returns to the cluster, it does so in the secondary role, thus, as outlined in the previous section, no further manual intervention is necessary. Again, DRBD does not change the resource role back, it is up to the cluster manager to do so (if so configured).

DRBD ensures block device consistency in case of a primary node failure by way of a special mechanism. For a detailed discussion, refer to The Activity Log.

13.2.3. Dealing with permanent node failure

If a node suffers an unrecoverable problem or permanent destruction, you must follow the following steps:

- Replace the failed hardware with one with similar performance and disk capacity.

Replacing a failed node with one with worse performance characteristics is possible, but not recommended. Replacing a failed node with one with less disk capacity is not supported, and will cause DRBD to refuse to connect to the replaced node [13: It couldn't replicate the data, anyway!].

- Install the base system and applications.
- Install DRBD and copy `/etc/drbd.conf` and all of `/etc/drbd.d/` from one of the surviving nodes.
- Follow the steps outlined in Configuring DRBD, but stop short of The initial device synchronization.

Manually starting a full device synchronization is not necessary at this point, it will commence automatically upon connection to the surviving primary and/or secondary node(s).

13.3. Manual split brain recovery

DRBD detects split brain at the time connectivity becomes available again and the peer nodes exchange the initial DRBD protocol handshake. If DRBD detects that both nodes are (or were at some point, while disconnected) in the primary role, it immediately tears down the replication connection. The tell-tale sign of this is a message like the following appearing in the system log:

```
Split-Brain detected, dropping connection!
```

After split brain has been detected, one node will always have the resource in a *StandAlone* connection state. The other might either also be in the *StandAlone* state (if both nodes detected the split brain simultaneously), or in *Connecting* (if the peer tore down the connection before the other node had a chance to detect split brain).

At this point, unless you configured DRBD to automatically recover from split brain, you must manually intervene by selecting one node whose modifications will be discarded (this node is referred to as the *split brain victim*). This intervention is made with the following commands:

This is still *work in progress*.

Expect rough edges and changes.

```
# drbdadm disconnect <resource>
# drbdadm secondary <resource>
# drbdadm connect --discard-my-data <resource>
```

On the other node (the *split brain survivor*), if its connection state is also *StandAlone*, you would enter:

```
# drbdadm disconnect <resource>
# drbdadm connect <resource>
```

You may omit this step if the node is already in the *Connecting* state; it will then reconnect automatically.

With the DRBD 9 pre-release you might get a slightly misleading error message

```
Failure: (102) Local address(port) already in use.
```

when in *StandAlone* and trying to reconnect, because the kernel module still has active connection data.

In this case just drop the network setup with `drbdadm disconnect`, and continue with a `drbdadm connect` as usual.

If the resource affected by the split brain is a stacked resource, use `drbdadm --stacked` instead of just `drbdadm`.

Upon connection, your split brain victim immediately changes its connection state to *SyncTarget*, and gets its modifications overwritten by the other node(s).

The split brain victim is not subjected to a full device synchronization. Instead, it has its local modifications rolled back, and any modifications made on the split brain survivor(s) propagate to the victim.

After re-synchronization has completed, the split brain is considered resolved and the nodes form a fully consistent, redundant replicated storage system again.

DRBD-enabled applications

Chapter 14. Integrating DRBD with Pacemaker clusters

Using DRBD in conjunction with the Pacemaker cluster stack is arguably DRBD's most frequently found use case. Pacemaker is also one of the applications that make DRBD extremely powerful in a wide variety of usage scenarios.

DRBD can be used in Pacemaker clusters in two ways: * DRBD running as a background-service, used like a SAN; or * DRBD completely managed by Pacemaker.

Both have a few advantages and disadvantages, these will be discussed below.

It's recommended to have some fencing configured. If your cluster has communication issues (eg. network switch loses power) and gets split, the parts might start the services (failover) and cause a **Split-Brain** when the communication resumes again.

14.1. Pacemaker primer

Pacemaker is a sophisticated, feature-rich, and widely deployed cluster resource manager for the Linux platform. It comes with a rich set of documentation. In order to understand this chapter, reading the following documents is highly recommended:

- Clusters From Scratch, a step-by-step guide to configuring high availability clusters;
- CRM CLI (command line interface) tool, a manual for the CRM shell, a simple and intuitive command line interface bundled with Pacemaker;
- Pacemaker Configuration Explained, a reference document explaining the concept and design behind Pacemaker.

14.2. Using DRBD as a background service in a pacemaker cluster

In this section you will see that using autonomous DRBD storage can look like local storage; so integrating in a Pacemaker cluster is done by pointing your mount points at DRBD.

First of all, we will use the `auto-promote` feature of DRBD, so that DRBD automatically sets itself *Primary* when needed. This will probably apply to all of your resources, so setting that a default in the `common` section makes sense:

```
common {
  options {
    auto-promote yes;
    ...
  }
}
```

Now you just need to use your storage, eg. via a filesystem:

Listing 10. Pacemaker configuration for DRBD-backed MySQL service, using `auto-promote`

```
crm configure
crm(live)configure# primitive fs_mysql ocf:heartbeat:Filesystem \
                    params device="/dev/drbd/by-res/mysql/0" \
                      directory="/var/lib/mysql" fstype="ext3"
crm(live)configure# primitive ip_mysql ocf:heartbeat:IPaddr2 \
                    params ip="10.9.42.1" nic="eth0"
crm(live)configure# primitive mysqld lsb:mysqld
crm(live)configure# group mysql fs_mysql ip_mysql mysqld
crm(live)configure# commit
crm(live)configure# exit
bye
```

Essentially all that is needed is a mountpoint (`/var/lib/mysql` in this example) where the DRBD resource gets mounted.

As long as Pacemaker has control, it will only allow a single instance of that mount across your cluster.

14.3. Adding a DRBD-backed service to the cluster configuration, including a master-slave resource

This section explains how to enable a DRBD-backed service in a Pacemaker cluster.

If you are employing the DRBD OCF resource agent, it is recommended that you defer DRBD startup, shutdown, promotion, and demotion *exclusively* to the OCF resource agent. That means that you should disable the DRBD init script:

```
chkconfig drbd off
```

The `ocf:linbit:drbd` OCF resource agent provides Master/Slave capability, allowing Pacemaker to start and monitor the DRBD resource on multiple nodes and promoting and demoting as needed. You must, however, understand that the DRBD RA disconnects and detaches all DRBD resources it manages on Pacemaker shutdown, and also upon enabling standby mode for a node.

The OCF resource agent which ships with DRBD belongs to the `linbit` provider, and hence installs as `/usr/lib/ocf/resource.d/linbit/drbd`. There is a legacy resource agent that ships as part of the OCF resource agents package, which uses the `heartbeat` provider and installs into `/usr/lib/ocf/resource.d/heartbeat/drbd`. The legacy OCF RA is deprecated and should no longer be used.

In order to enable a DRBD-backed configuration for a MySQL database in a Pacemaker CRM cluster with the `drbd` OCF resource agent, you must create both the necessary resources, and Pacemaker constraints to ensure your service only starts on a previously promoted DRBD resource. You may do so using the `crm` shell, as outlined in the following example:

Listing 11. Pacemaker configuration for DRBD-backed MySQL service, using a `master-slave` *resource*

```
crm configure
crm(live)configure# primitive drbd_mysql ocf:linbit:drbd \
                    params drbd_resource="mysql" \
                    op monitor interval="29s" role="Master" \
                    op monitor interval="31s" role="Slave"
crm(live)configure# ms ms_drbd_mysql drbd_mysql \
                    meta master-max="1" master-node-max="1" \
                         clone-max="2" clone-node-max="1" \
                         notify="true"
crm(live)configure# primitive fs_mysql ocf:heartbeat:Filesystem \
                    params device="/dev/drbd/by-res/mysql/0" \
                      directory="/var/lib/mysql" fstype="ext3"
crm(live)configure# primitive ip_mysql ocf:heartbeat:IPaddr2 \
                    params ip="10.9.42.1" nic="eth0"
crm(live)configure# primitive mysqld lsb:mysqld
crm(live)configure# group mysql fs_mysql ip_mysql mysqld
crm(live)configure# colocation mysql_on_drbd \
                      inf: mysql ms_drbd_mysql:Master
crm(live)configure# order mysql_after_drbd \
                      inf: ms_drbd_mysql:promote mysql:start
crm(live)configure# commit
crm(live)configure# exit
bye
```

After this, your configuration should be enabled. Pacemaker now selects a node on which it promotes the DRBD resource, and then starts the DRBD-backed resource group on that same node.

14.4. Using resource-level fencing in Pacemaker clusters

This section outlines the steps necessary to prevent Pacemaker from promoting a DRBD Master/Slave resource when its DRBD replication link has been interrupted. This keeps Pacemaker from starting a service with outdated data and causing an unwanted "time warp" in the process.

In order to enable any resource-level fencing for DRBD, you must add the following lines to your resource configuration:

```
resource <resource> {
  net {
    fencing resource-only;
    ...
  }
}
```

You will also have to make changes to the `handlers` section depending on the cluster infrastructure being used:

- Heartbeat-based Pacemaker clusters can employ the configuration outlined in Resource-level fencing with dopd.
- Both Corosync- and Heartbeat-based clusters can use the functionality explained in Resource-level fencing using the Cluster Information Base (CIB).

It is absolutely vital to configure at least two independent cluster communications channels for this functionality to work correctly. Heartbeat-based Pacemaker clusters should define at least two cluster communication links in their `ha.cf` configuration files. Corosync clusters should list at least two redundant rings in `corosync.conf`.

14.4.1. Resource-level fencing with dopd

In Heartbeat-based Pacemaker clusters, DRBD can use a resources-level fencing facility named the *DRBD outdate-peer daemon*, or dopd for short.

Heartbeat configuration for dopd

To enable dopd, you must add these lines to your `/etc/ha.d/ha.cf` file:

```
respawn hacluster /usr/lib/heartbeat/dopd
apiauth dopd gid=haclient uid=hacluster
```

You may have to adjust dopd's path according to your preferred distribution. On some distributions and architectures, the correct path is `/usr/lib64/heartbeat/dopd`.

After you have made this change and copied `ha.cf` to the peer node, put Pacemaker in maintenance mode and run `/etc/init.d/heartbeat reload` to have Heartbeat re-read its configuration file. Afterwards, you should be able to verify that you now have a running dopd process.

You can check for this process either by running `ps ax | grep dopd` or by issuing `killall -0 dopd`.

DRBD Configuration for dopd

Once dopd is running, add these items to your DRBD resource configuration:

```
resource <resource> {
    handlers {
        fence-peer "/usr/lib/heartbeat/drbd-peer-outdater -t 5";
        ...
    }
    net {
        fencing resource-only;
        ...
    }
    ...
}
```

As with dopd, your distribution may place the `drbd-peer-outdater` binary in `/usr/lib64/heartbeat` depending on your system architecture.

Finally, copy your `drbd.conf` to the peer node and issue `drbdadm adjust resource` to reconfigure your resource and reflect your changes.

Testing dopd functionality

To test whether your dopd setup is working correctly, interrupt the replication link of a configured and connected resource while Heartbeat services are running normally. You may do so simply by physically unplugging the network link, but that is fairly invasive. Instead, you may insert a temporary `iptables` rule to drop incoming DRBD traffic to

the TCP port used by your resource.

After this, you will be able to observe the resource connection state change from *Connected* to *Connecting*. Allow a few seconds to pass, and you should see the disk statebecome *Outdated/DUnknown*. That is what dopd is responsible for.

Any attempt to switch the outdated resource to the primary role will fail after this.

When re-instituting network connectivity (either by plugging the physical link or by removing the temporary `iptables` rule you inserted previously), the connection state will change to *Connected*, and then promptly to *SyncTarget* (assuming changes occurred on the primary node during the network interruption). Then you will be able to observe a brief synchronization period, and finally, the previously outdated resource will be marked as *UpToDate* again.

14.4.2. Resource-level fencing using the Cluster Information Base (CIB)

In order to enable resource-level fencing for Pacemaker, you will have to set two options in `drbd.conf`:

```
resource <resource> {
  net {
    fencing resource-only;
    ...
  }
  handlers {
    fence-peer "/usr/lib/drbd/crm-fence-peer.9.sh";
    after-resync-target "/usr/lib/drbd/crm-unfence-peer.9.sh";
    ...
  }
  ...
}
```

Thus, if the DRBD replication link becomes disconnected, the `crm-fence-peer.9.sh` script contacts the cluster manager, determines the Pacemaker Master/Slave resource associated with this DRBD resource, and ensures that the Master/Slave resource no longer gets promoted on any node other than the currently active one. Conversely, when the connection is re-established and DRBD completes its synchronization process, then that constraint is removed and the cluster manager is free to promote the resource on any node again.

14.5. Using stacked DRBD resources in Pacemaker clusters

Stacking is deprecated in DRBD version 9.x, as more nodes can be implemented on a single level. See Defining network connections for details.

Stacked resources allow DRBD to be used for multi-level redundancy in multiple-node clusters, or to establish off-site disaster recovery capability. This section describes how to configure DRBD and Pacemaker in such configurations.

14.5.1. Adding off-site disaster recovery to Pacemaker clusters

In this configuration scenario, we would deal with a two-node high availability cluster in one site, plus a separate node which would presumably be housed off-site. The third node acts as a disaster recovery node and is a standalone server. Consider the following illustration to describe the concept.

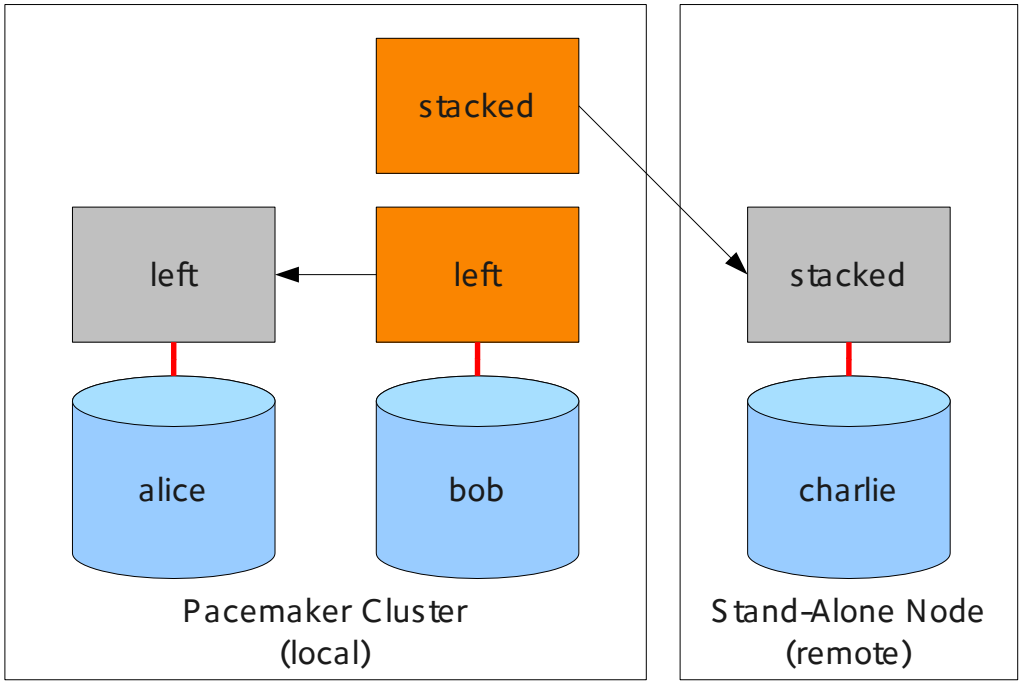


Figure 14. DRBD resource stacking in Pacemaker clusters

In this example, 'alice' and 'bob' form a two-node Pacemaker cluster, whereas 'charlie' is an off-site node not managed by Pacemaker.

To create such a configuration, you would first configure and initialize DRBD resources as described in Creating a stacked three-node setup. Then, configure Pacemaker with the following CRM configuration:

```
primitive p_drbd_r0 ocf:linbit:drbd \
    params drbd_resource="r0"

primitive p_drbd_r0-U ocf:linbit:drbd \
    params drbd_resource="r0-U"

primitive p_ip_stacked ocf:heartbeat:IPaddr2 \
    params ip="192.168.42.1" nic="eth0"

ms ms_drbd_r0 p_drbd_r0 \
    meta master-max="1" master-node-max="1" \
        clone-max="2" clone-node-max="1" \
        notify="true" globally-unique="false"

ms ms_drbd_r0-U p_drbd_r0-U \
    meta master-max="1" clone-max="1" \
        clone-node-max="1" master-node-max="1" \
        notify="true" globally-unique="false"

colocation c_drbd_r0-U_on_drbd_r0 \
        inf: ms_drbd_r0-U ms_drbd_r0:Master

colocation c_drbd_r0-U_on_ip \
        inf: ms_drbd_r0-U p_ip_stacked

colocation c_ip_on_r0_master \
        inf: p_ip_stacked ms_drbd_r0:Master

order o_ip_before_r0-U \
        inf: p_ip_stacked ms_drbd_r0-U:start

order o_drbd_r0_before_r0-U \
        inf: ms_drbd_r0:promote ms_drbd_r0-U:start
```

Assuming you created this configuration in a temporary file named `/tmp/crm.txt`, you may import it into the live cluster configuration with the following command:

```
crm configure < /tmp/crm.txt
```

This configuration will ensure that the following actions occur in the correct order on the 'alice'/'bob' cluster:

1. Pacemaker starts the DRBD resource `r0` on both cluster nodes, and promotes one node to the Master (DRBD Primary) role.
2. Pacemaker then starts the IP address 192.168.42.1, which the stacked resource is to use for replication to the third node. It does so on the node it has previously promoted to the Master role for `r0` DRBD resource.
3. On the node which now has the Primary role for `r0` and also the replication IP address for `r0-U`, Pacemaker now starts the `r0-U` DRBD resource, which connects and replicates to the off-site node.
4. Pacemaker then promotes the `r0-U` resource to the Primary role too, so it can be used by an application.

Thus, this Pacemaker configuration ensures that there is not only full data redundancy between cluster nodes, but also to the third, off-site node.

This type of setup is usually deployed together with DRBD Proxy.

14.5.2. Using stacked resources to achieve 4-way redundancy in Pacemaker clusters

In this configuration, a total of three DRBD resources (two unstacked, one stacked) are used to achieve 4-way storage redundancy. This means that of a 4-node cluster, up to three nodes can fail while still providing service availability.

Consider the following illustration to explain the concept.

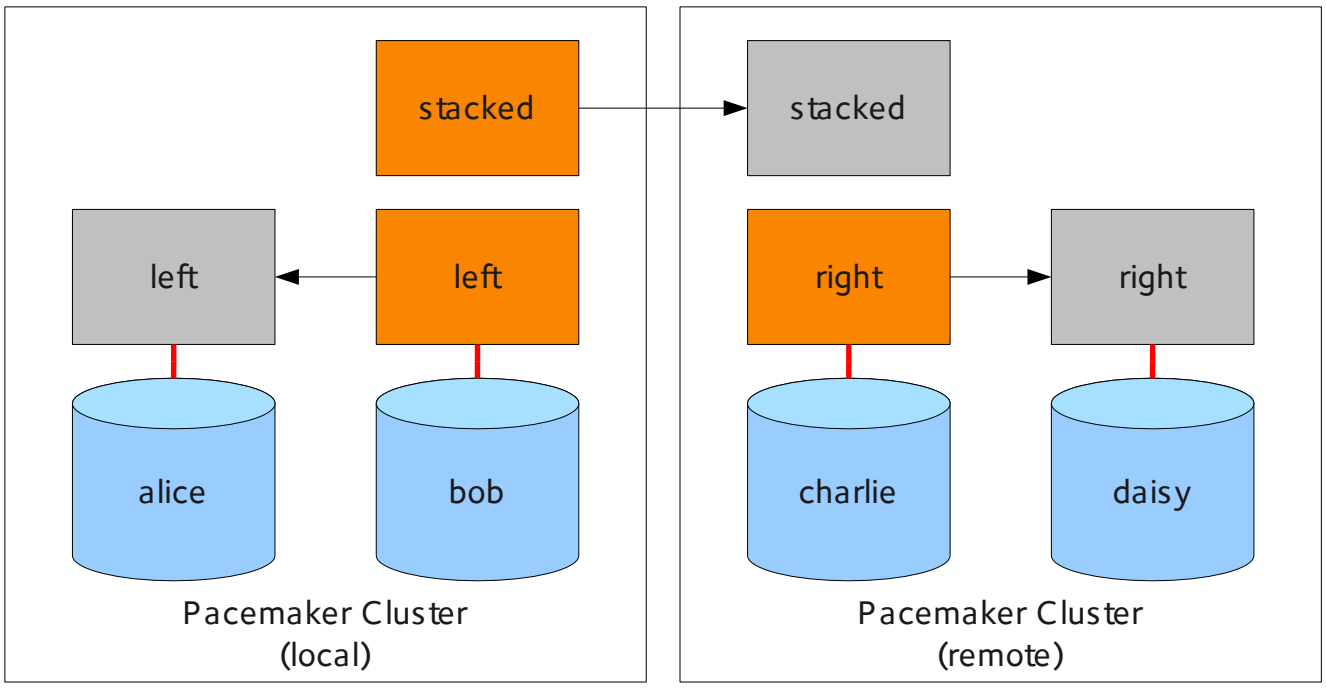


Figure 15. DRBD resource stacking in Pacemaker clusters

In this example, 'alice', 'bob', 'charlie', and 'daisy' form two two-node Pacemaker clusters. 'alice' and 'bob' form the cluster named `left` and replicate data using a DRBD resource between them, while 'charlie' and 'daisy' do the same with a separate DRBD resource, in a cluster named `right`. A third, stacked DRBD resource connects the two clusters.

Due to limitations in the Pacemaker cluster manager as of Pacemaker version 1.0.5, it is not possible to create this setup in a single four-node cluster without disabling CIB validation, which is an advanced process not recommended for general-purpose use. It is anticipated that this is being addressed in future Pacemaker releases.

To create such a configuration, you would first configure and initialize DRBD resources as described in Creating a stacked three-node setup (except that the remote half of the DRBD configuration is also stacked, not just the local cluster). Then, configure Pacemaker with the following CRM configuration, starting with the cluster `left`:

```
primitive p_drbd_left ocf:linbit:drbd \
    params drbd_resource="left"

primitive p_drbd_stacked ocf:linbit:drbd \
    params drbd_resource="stacked"

primitive p_ip_stacked_left ocf:heartbeat:IPaddr2 \
    params ip="10.9.9.100" nic="eth0"

ms ms_drbd_left p_drbd_left \
    meta master-max="1" master-node-max="1" \
        clone-max="2" clone-node-max="1" \
        notify="true"

ms ms_drbd_stacked p_drbd_stacked \
    meta master-max="1" clone-max="1" \
        clone-node-max="1" master-node-max="1" \
        notify="true" target-role="Master"

colocation c_ip_on_left_master \
        inf: p_ip_stacked_left ms_drbd_left:Master

colocation c_drbd_stacked_on_ip_left \
        inf: ms_drbd_stacked p_ip_stacked_left

order o_ip_before_stacked_left \
        inf: p_ip_stacked_left ms_drbd_stacked:start

order o_drbd_left_before_stacked_left \
        inf: ms_drbd_left:promote ms_drbd_stacked:start
```

Assuming you created this configuration in a temporary file named `/tmp/crm.txt`, you may import it into the live cluster configuration with the following command:

```
crm configure < /tmp/crm.txt
```

After adding this configuration to the CIB, Pacemaker will execute the following actions:

1. Bring up the DRBD resource `left` replicating between 'alice' and 'bob' promoting the resource to the Master role on one of these nodes.
2. Bring up the IP address 10.9.9.100 (on either 'alice' or 'bob', depending on which of these holds the Master role for the resource `left`).
3. Bring up the DRBD resource `stacked` on the same node that holds the just-configured IP address.
4. Promote the stacked DRBD resource to the Primary role.

Now, proceed on the cluster `right` by creating the following configuration:

```
primitive p_drbd_right ocf:linbit:drbd \
    params drbd_resource="right"

primitive p_drbd_stacked ocf:linbit:drbd \
    params drbd_resource="stacked"

primitive p_ip_stacked_right ocf:heartbeat:IPaddr2 \
    params ip="10.9.10.101" nic="eth0"

ms ms_drbd_right p_drbd_right \
    meta master-max="1" master-node-max="1" \
        clone-max="2" clone-node-max="1" \
        notify="true"

ms ms_drbd_stacked p_drbd_stacked \
    meta master-max="1" clone-max="1" \
        clone-node-max="1" master-node-max="1" \
        notify="true" target-role="Slave"

colocation c_drbd_stacked_on_ip_right \
        inf: ms_drbd_stacked p_ip_stacked_right

colocation c_ip_on_right_master \
        inf: p_ip_stacked_right ms_drbd_right:Master

order o_ip_before_stacked_right \
        inf: p_ip_stacked_right ms_drbd_stacked:start

order o_drbd_right_before_stacked_right \
        inf: ms_drbd_right:promote ms_drbd_stacked:start
```

After adding this configuration to the CIB, Pacemaker will execute the following actions:

1. Bring up the DRBD resource `right` replicating between 'charlie' and 'daisy', promoting the resource to the Master role on one of these nodes.
2. Bring up the IP address 10.9.10.101 (on either 'charlie' or 'daisy', depending on which of these holds the Master role for the resource `right`).
3. Bring up the DRBD resource `stacked` on the same node that holds the just-configured IP address.
4. Leave the stacked DRBD resource in the Secondary role (due to `target-role="Slave"`).

14.6. Configuring DRBD to replicate between two SAN-backed Pacemaker clusters

This is a somewhat advanced setup usually employed in split-site configurations. It involves two separate Pacemaker clusters, where each cluster has access to a separate Storage Area Network (SAN). DRBD is then used to replicate data stored on that SAN, across an IP link between sites.

Consider the following illustration to describe the concept.

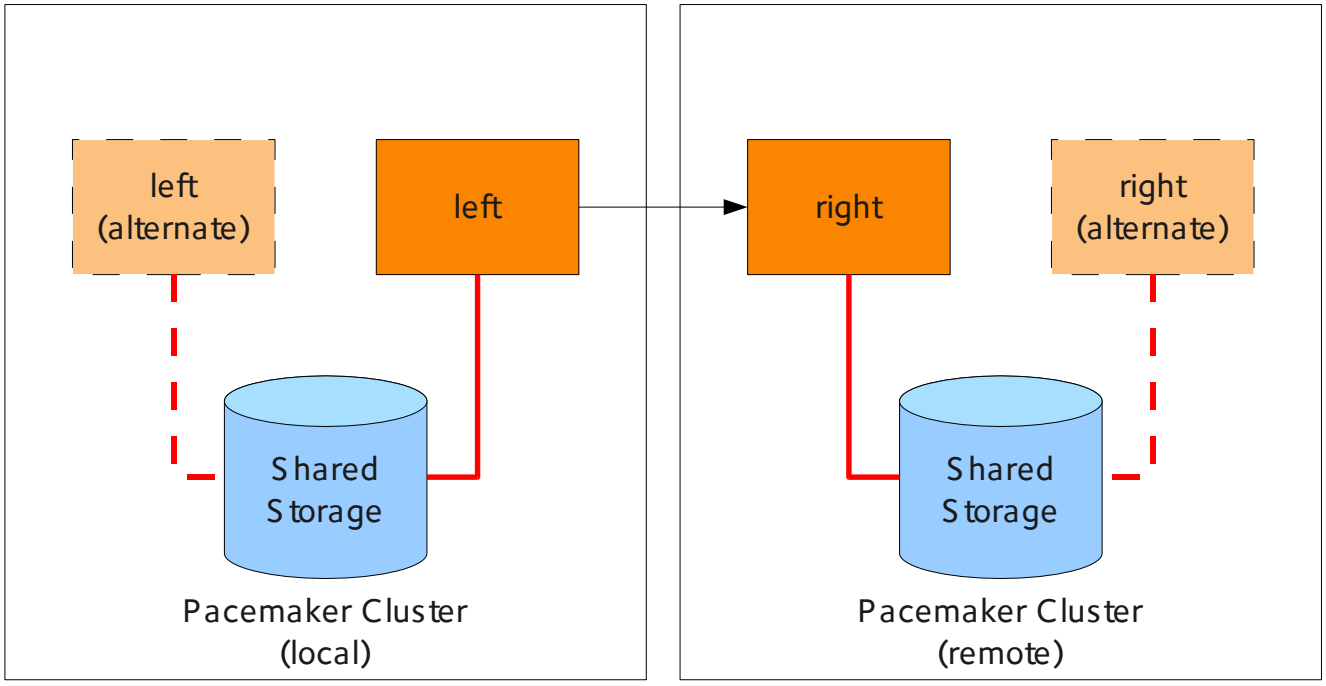


Figure 16. Using DRBD to replicate between SAN-based clusters

Which of the individual nodes in each site currently acts as the DRBD peer is not explicitly defined — the DRBD peers are said to *float*; that is, DRBD binds to virtual IP addresses not tied to a specific physical machine.

This type of setup is usually deployed together with DRBD Proxy and/or truck based replication.

Since this type of setup deals with shared storage, configuring and testing STONITH is absolutely vital for it to work properly.

14.6.1. DRBD resource configuration

To enable your DRBD resource to float, configure it in `drbd.conf` in the following fashion:

```
resource <resource> {
  ...
  device /dev/drbd0;
  disk /dev/sda1;
  meta-disk internal;
  floating 10.9.9.100:7788;
  floating 10.9.10.101:7788;
}
```

The `floating` keyword replaces the `on <host>` sections normally found in the resource configuration. In this mode, DRBD identifies peers by IP address and TCP port, rather than by host name. It is important to note that the addresses specified must be virtual cluster IP addresses, rather than physical node IP addresses, for floating to function properly. As shown in the example, in split-site configurations the two floating addresses can be expected to belong to two separate IP networks — it is thus vital for routers and firewalls to properly allow DRBD replication traffic between the nodes.

14.6.2. Pacemaker resource configuration

A DRBD floating peers setup, in terms of Pacemaker configuration, involves the following items (in each of the two Pacemaker clusters involved):

- A virtual cluster IP address.
- A master/slave DRBD resource (using the DRBD OCF resource agent).
- Pacemaker constraints ensuring that resources are started on the correct nodes, and in the correct order.

To configure a resource named `mysql` in a floating peers configuration in a 2-node cluster, using the replication address `10.9.9.100`, configure Pacemaker with the following `crm` commands:

```
crm configure
crm(live)configure# primitive p_ip_float_left ocf:heartbeat:IPaddr2 \
                    params ip=10.9.9.100
crm(live)configure# primitive p_drbd_mysql ocf:linbit:drbd \
                    params drbd_resource=mysql
crm(live)configure# ms ms_drbd_mysql drbd_mysql \
                    meta master-max="1" master-node-max="1" \
                         clone-max="1" clone-node-max="1" \
                         notify="true" target-role="Master"
crm(live)configure# order drbd_after_left \
                      inf: p_ip_float_left ms_drbd_mysql
crm(live)configure# colocation drbd_on_left \
                      inf: ms_drbd_mysql p_ip_float_left
crm(live)configure# commit
bye
```

After adding this configuration to the CIB, Pacemaker will execute the following actions:

1. Bring up the IP address 10.9.9.100 (on either 'alice' or 'bob').
2. Bring up the DRBD resource according to the IP address configured.
3. Promote the DRBD resource to the Primary role.

Then, in order to create the matching configuration in the other cluster, configure *that* Pacemaker instance with the following commands:

```
crm configure
crm(live)configure# primitive p_ip_float_right ocf:heartbeat:IPaddr2 \
                    params ip=10.9.10.101
crm(live)configure# primitive drbd_mysql ocf:linbit:drbd \
                    params drbd_resource=mysql
crm(live)configure# ms ms_drbd_mysql drbd_mysql \
                    meta master-max="1" master-node-max="1" \
                         clone-max="1" clone-node-max="1" \
                         notify="true" target-role="Slave"
crm(live)configure# order drbd_after_right \
                      inf: p_ip_float_right ms_drbd_mysql
crm(live)configure# colocation drbd_on_right
                      inf: ms_drbd_mysql p_ip_float_right
crm(live)configure# commit
bye
```

After adding this configuration to the CIB, Pacemaker will execute the following actions:

1. Bring up the IP address 10.9.10.101 (on either 'charlie' or 'daisy').
2. Bring up the DRBD resource according to the IP address configured.
3. Leave the DRBD resource in the Secondary role (due to `target-role="Slave"`).

14.6.3. Site fail-over

In split-site configurations, it may be necessary to transfer services from one site to another. This may be a consequence of a scheduled transition, or of a disastrous event. In case the transition is a normal, anticipated event, the recommended course of action is this:

- Connect to the cluster on the site about to relinquish resources, and change the affected DRBD resource's `target-role` attribute from `Master` to `Slave`. This will shut down any resources depending on the Primary role of the DRBD resource, demote it, and continue to run, ready to receive updates from a new Primary.
- Connect to the cluster on the site about to take over resources, and change the affected DRBD resource's `target-role` attribute from `Slave` to `Master`. This will promote the DRBD resources, start any other Pacemaker resources depending on the Primary role of the DRBD resource, and replicate updates to the remote site.
- To fail back, simply reverse the procedure.

In the event that of a catastrophic outage on the active site, it can be expected that the site is off line and no longer replicated to the backup site. In such an event:

- Connect to the cluster on the still-functioning site resources, and change the affected DRBD resource's `target-role` attribute from `Slave` to `Master`. This will promote the DRBD resources, and start any other Pacemaker resources depending on the Primary role of the DRBD resource.
- When the original site is restored or rebuilt, you may connect the DRBD resources again, and subsequently fail back using the reverse procedure.

Chapter 15. Integrating DRBD with Red Hat Cluster

This chapter describes using DRBD as replicated storage for Red Hat Cluster high availability clusters.

This guide uses the unofficial term *Red Hat Cluster* to refer to a product that has had multiple official product names over its history, including *Red Hat Cluster Suite* and *Red Hat Enterprise Linux High Availability Add-On*.

15.1. Red Hat Cluster background information

15.1.1. Fencing

Red Hat Cluster, originally designed primarily for shared storage clusters, relies on node fencing to prevent concurrent, uncoordinated access to shared resources. The Red Hat Cluster fencing infrastructure relies on the fencing daemon `fenced`, and fencing agents implemented as shell scripts.

Even though DRBD-based clusters utilize no shared storage resources and thus fencing is not strictly required from DRBD's standpoint, Red Hat Cluster Suite still requires fencing even in DRBD-based configurations.

15.1.2. The Resource Group Manager

The resource group manager (`rgmanager`, alternatively `clurgmgr`) is akin to Pacemaker. It serves as the cluster management suite's primary interface with the applications it is configured to manage.

Red Hat Cluster resources

A single highly available application, filesystem, IP address and the like is referred to as a *resource* in Red Hat Cluster terminology.

Where resources depend on each other — such as, for example, an NFS export depending on a filesystem being mounted — they form a *resource tree*, a form of nesting resources inside another. Resources in inner levels of nesting may inherit parameters from resources in outer nesting levels. The concept of resource trees is absent in Pacemaker.

Red Hat Cluster services

Where resources form a co-dependent collection, that collection is called a *service*. This is different from Pacemaker, where such a collection is referred to as a *resource group*.

rgmanager resource agents

The resource agents invoked by `rgmanager` are similar to those used by Pacemaker, in the sense that they utilize the same shell-based API as defined in the Open Cluster Framework (OCF), although Pacemaker utilizes some extensions not defined in the framework. Thus in theory, the resource agents are largely interchangeable between Red Hat Cluster Suite and Pacemaker — in practice however, the two cluster management suites use different resource agents even for similar or identical tasks.

Red Hat Cluster resource agents install into the `/usr/share/cluster/` directory. Unlike Pacemaker OCF resource agents which are by convention self-contained, some Red Hat Cluster resource agents are split into a `.sh` file containing the actual shell code, and a `.metadata` file containing XML resource agent metadata.

DRBD includes a Red Hat Cluster resource agent. It installs into the customary directory as `drbd.sh` and `drbd.metadata`.

15.2. Red Hat Cluster configuration

This section outlines the configuration steps necessary to get Red Hat Cluster running. Preparing your cluster configuration is fairly straightforward; all a DRBD-based Red Hat Cluster requires are two participating nodes (referred to as *Cluster Members* in Red Hat's documentation) and a fencing device.

For more information about configuring Red Hat clusters, see Red Hat's documentation on the Red Hat Cluster and GFS.

15.2.1. The `cluster.conf` file

RHEL clusters keep their configuration in a single configuration file, `/etc/cluster/cluster.conf`. You may manage your cluster configuration in the following ways:

Editing the configuration file directly

This is the most straightforward method. It has no prerequisites other than having a text editor available.

Using the `system-config-cluster` *GUI*

This is a GUI application written in Python using Glade. It requires the availability of an X display (either directly on a server console, or tunneled via SSH).

Using the Conga web-based management infrastructure

The Conga infrastructure consists of a node agent (`ricci`) communicating with the local cluster manager, cluster resource manager, and cluster LVM daemon, and an administration web application (`luci`) which may be used to configure the cluster infrastructure using a simple web browser.

15.3. Using DRBD in Red Hat Cluster fail-over clusters

This section deals exclusively with setting up DRBD for Red Hat Cluster fail-over clusters not involving GFS. For GFS (and GFS2) configurations, please see Using GFS with DRBD.

This section, like Integrating DRBD with Pacemaker clusters, assumes you are about to configure a highly available MySQL database with the following configuration parameters:

- The DRBD resources to be used as your database storage area is named `mysql`, and it manages the device `/dev/drbd0`.
- The DRBD device holds an ext3 filesystem which is to be mounted to `/var/lib/mysql` (the default MySQL data directory).
- The MySQL database is to utilize that filesystem, and listen on a dedicated cluster IP address, 192.168.42.1.

15.3.1. Setting up your cluster configuration

To configure your highly available MySQL database, create or modify your `/etc/cluster/cluster.conf` file to contain the following configuration items.

To do that, open `/etc/cluster/cluster.conf` with your preferred text editing application. Then, include the following items in your resource configuration:

```
<rm>
  <resources />
  <service autostart="1" name="mysql">
    <drbd name="drbd-mysql" resource="mysql">
      <fs device="/dev/drbd/by-res/mysql/0"
          mountpoint="/var/lib/mysql"
          fstype="ext3"
          name="mysql"
          options="noatime"/>
    </drbd>
    <ip address="192.168.42.1" monitor_link="1"/>
    <mysql config_file="/etc/my.cnf"
           listen_address="192.168.42.1"
           name="mysqld"/>
  </service>
</rm>
```

This example assumes a single-volume resource.

Nesting resource references inside one another in `<service/>` is the Red Hat Cluster way of expressing resource dependencies.

Be sure to increment the `config_version` attribute, found on the root `<cluster>` element, after you have completed your configuration. Then, issue the following commands to commit your changes to the running cluster configuration:

```
# ccs_tool update /etc/cluster/cluster.conf
# cman_tool version -r <version>
```

In the second command, be sure to replace *<version>* with the new cluster configuration version number.

Both the `system-config-cluster` GUI configuration utility and the Conga web based cluster management infrastructure will complain about your cluster configuration after including the `drbd` resource agent in your `cluster.conf` file. This is due to the design of the Python cluster management wrappers provided by these two applications which does not expect third party extensions to the cluster infrastructure.

Thus, when you utilize the `drbd` resource agent in cluster configurations, it is not recommended to utilize `system-config-cluster` nor Conga for cluster configuration purposes. Using either of these tools to only monitor the cluster's status, however, is expected to work fine.

Chapter 16. Using LVM with DRBD

This chapter deals with managing DRBD in conjunction with LVM2. In particular, it covers

- using LVM Logical Volumes as backing devices for DRBD;
- using DRBD devices as Physical Volumes for LVM;
- combining these to concepts to implement a layered LVM approach using DRBD.

If you happen to be unfamiliar with these terms to begin with, LVM primer may serve as your LVM starting point — although you are always encouraged, of course, to familiarize yourself with LVM in some more detail than this section provides.

16.1. LVM primer

LVM2 is an implementation of logical volume management in the context of the Linux device mapper framework. It has practically nothing in common, other than the name and acronym, with the original LVM implementation. The old implementation (now retroactively named "LVM1") is considered obsolete; it is not covered in this section.

When working with LVM, it is important to understand its most basic concepts:

Physical Volume (PV)

A PV is an underlying block device exclusively managed by LVM. PVs can either be entire hard disks or individual partitions. It is common practice to create a partition table on the hard disk where one partition is dedicated to the use by the Linux LVM.

The partition type "Linux LVM" (signature `0x8E`) can be used to identify partitions for exclusive use by LVM. This, however, is not required — LVM recognizes PVs by way of a signature written to the device upon PV initialization.

Volume Group (VG)

A VG is the basic administrative unit of the LVM. A VG may include one or more several PVs. Every VG has a unique name. A VG may be extended during runtime by adding additional PVs, or by enlarging an existing PV.

Logical Volume (LV)

LVs may be created during runtime within VGs and are available to the other parts of the kernel as regular block devices. As such, they may be used to hold a file system, or for any other purpose block devices may be used for. LVs may be resized while they are online, and they may also be moved from one PV to another (as long as the PVs are part of the same VG).

Snapshot Logical Volume (SLV)

Snapshots are temporary point-in-time copies of LVs. Creating snapshots is an operation that completes almost instantly, even if the original LV (the *origin volume*) has a size of several hundred GiByte. Usually, a snapshot requires significantly less space than the original LV.

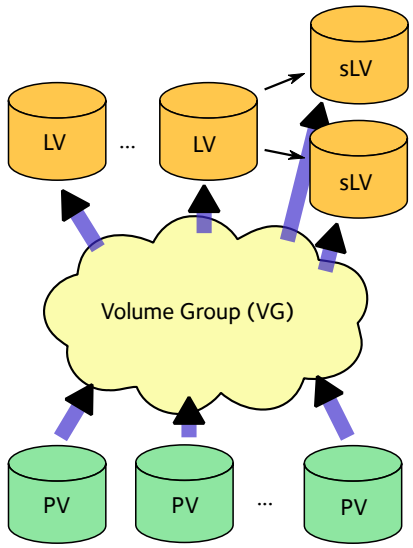


Figure 17. LVM overview

16.2. Using a Logical Volume as a DRBD backing device

Since an existing Logical Volume is simply a block device in Linux terms, you may of course use it as a DRBD backing device. To use LV's in this manner, you simply create them, and then initialize them for DRBD as you normally would.

This example assumes that a Volume Group named `foo` already exists on both nodes of on your LVM-enabled system, and that you wish to create a DRBD resource named `r0` using a Logical Volume in that Volume Group.

First, you create the Logical Volume:

```
# lvcreate --name bar --size 10G foo
Logical volume "bar" created
```

Of course, you must complete this command on both nodes of your DRBD cluster. After this, you should have a block device named `/dev/foo/bar` on either node.

Then, you can simply enter the newly-created volumes in your resource configuration:

```
resource r0 {
  ...
  on alice {
    device /dev/drbd0;
    disk   /dev/foo/bar;
    ...
  }
  on bob {
    device /dev/drbd0;
    disk   /dev/foo/bar;
    ...
  }
}
```

Now you can continue to bring your resource up, just as you would if you were using non-LVM block devices.

16.3. Using automated LVM snapshots during DRBD synchronization

While DRBD is synchronizing, the *SyncTarget*'s state is *Inconsistent* until the synchronization completes. If in this situation the *SyncSource* happens to fail (beyond repair), this puts you in an unfortunate position: the node with good data is dead, and the surviving node has bad (inconsistent) data.

When serving DRBD off an LVM Logical Volume, you can mitigate this problem by creating an automated snapshot when synchronization starts, and automatically removing that same snapshot once synchronization has completed successfully.

In order to enable automated snapshotting during resynchronization, add the following lines to your resource configuration:

Listing 12. Automating snapshots before DRBD synchronization

```
resource r0 {
  handlers {
    before-resync-target "/usr/lib/drbd/snapshot-resync-target-lvm.sh";
    after-resync-target "/usr/lib/drbd/unsnapshot-resync-target-lvm.sh";
  }
}
```

The two scripts parse the $DRBD_RESOURCE environment variable which DRBD automatically passes to any `handler` it invokes. The `snapshot-resync-target-lvm.sh` script then creates an LVM snapshot for any volume the resource contains, then synchronization kicks off. In case the script fails, the synchronization *does not commence*.

Once synchronization completes, the `unsnapshot-resync-target-lvm.sh` script removes the snapshot, which is then no longer needed. In case unsnapshotting fails, the snapshot continues to linger around.

You should review dangling snapshots as soon as possible. A full snapshot causes both the snapshot itself *and its origin volume* to fail.

If at any time your *SyncSource* does fail beyond repair and you decide to revert to your latest snapshot on the peer, you may do so by issuing the `lvconvert -M` command.

16.4. Configuring a DRBD resource as a Physical Volume

In order to prepare a DRBD resource for use as a Physical Volume, it is necessary to create a PV signature on the DRBD device. In order to do so, issue one of the following commands on the node where the resource is currently in the primary role:

```
# pvcreate /dev/drbdX
```

or

```
# pvcreate /dev/drbd/by-res/<resource>/0
```

This example assumes a single-volume resource.

Now, it is necessary to include this device in the list of devices LVM scans for PV signatures. In order to do this, you must edit the LVM configuration file, normally named `/etc/lvm/lvm.conf`. Find the line in the `devices` section that contains the `filter` keyword and edit it accordingly. If *all* your PVs are to be stored on DRBD devices, the following is an appropriate `filter` option:

```
filter = [ "a|drbd.*|", "r|.*|" ]
```

This filter expression accepts PV signatures found on any DRBD devices, while rejecting (ignoring) all others.

By default, LVM scans all block devices found in `/dev` for PV signatures. This is equivalent to `filter = [ "a|.*|" ]`.

If you want to use stacked resources as LVM PVs, then you will need a more explicit filter configuration. You need to make sure that LVM detects PV signatures on stacked resources, while ignoring them on the corresponding lower-level resources and backing devices. This example assumes that your lower-level DRBD resources use device minors 0 through 9, whereas your stacked resources are using device minors from 10 upwards:

```
filter = [ "a|drbd1[0-9]|", "r|.*|" ]
```

This filter expression accepts PV signatures found only on the DRBD devices `/dev/drbd10` through `/dev/drbd19`, while rejecting (ignoring) all others.

After modifying the `lvm.conf` file, you must run the `vgscan` command so LVM discards its configuration cache and re-scans devices for PV signatures.

You may of course use a different `filter` configuration to match your particular system configuration. What is important to remember, however, is that you need to

- Accept (include) the DRBD devices you wish to use as PVs;
- Reject (exclude) the corresponding lower-level devices, so as to avoid LVM finding duplicate PV signatures.

In addition, you should disable the LVM cache by setting:

```
write_cache_state = 0
```

After disabling the LVM cache, make sure you remove any stale cache entries by deleting `/etc/lvm/cache/.cache`.

You must repeat the above steps on the peer nodes, too.

If your system has its root filesystem on LVM, Volume Groups will be activated from your initial ramdisk (initrd) during boot. In doing so, the LVM tools will evaluate an `lvm.conf` file included in the initrd image. Thus, after you make any changes to your `lvm.conf`, you should be certain to update your initrd with the utility appropriate for your distribution (`mkinitrd`, `update-initramfs` etc.).

When you have configured your new PV, you may proceed to add it to a Volume Group, or create a new Volume Group from it. The DRBD resource must, of course, be in the primary role while doing so.

```
# vgcreate <name> /dev/drbdX
```

While it is possible to mix DRBD and non-DRBD Physical Volumes within the same Volume Group, doing so is not recommended and unlikely to be of any practical value.

When you have created your VG, you may start carving Logical Volumes out of it, using the `lvcreate` command (as with a non-DRBD-backed Volume Group).

16.5. Adding a new DRBD volume to an existing Volume Group

Occasionally, you may want to add new DRBD-backed Physical Volumes to a Volume Group. Whenever you do so, a new volume should be added to an existing resource configuration. This preserves the replication stream and ensures write fidelity across all PVs in the VG.

if your LVM volume group is managed by Pacemaker as explained in Highly available LVM with Pacemaker, it is *imperative* to place the cluster in maintenance mode prior to making changes to the DRBD configuration.

Extend your resource configuration to include an additional volume, as in the following example:

```
resource r0 {
  volume 0 {
    device    /dev/drbd1;
    disk      /dev/sda7;
    meta-disk internal;
  }
  volume 1 {
    device    /dev/drbd2;
    disk      /dev/sda8;
    meta-disk internal;
  }
  on alice {
    address   10.1.1.31:7789;
  }
  on bob {
    address   10.1.1.32:7789;
  }
}
```

Make sure your DRBD configuration is identical across nodes, then issue:

```
# drbdadm adjust r0
```

This will implicitly call `drbdsetup new-minor r0 1` to enable the new volume `1` in the resource `r0`. Once the new volume has been added to the replication stream, you may initialize and add it to the volume group:

```
# pvcreate /dev/drbd/by-res/<resource>/1
# vgextend <name> /dev/drbd/by-res/<resource>/1
```

This will add the new PV `/dev/drbd/by-res/<resource>/1` to the `<name>` VG, preserving write fidelity across the entire VG.

16.6. Nested LVM configuration with DRBD

It is possible, if slightly advanced, to both use Logical Volumes as backing devices for DRBD *and* at the same time use a DRBD device itself as a Physical Volume. To provide an example, consider the following configuration:

- We have two partitions, named `/dev/sda1`, and `/dev/sdb1`, which we intend to use as Physical Volumes.
- Both of these PVs are to become part of a Volume Group named `local`.
- We want to create a 10-GiB Logical Volume in this VG, to be named `r0`.

- This LV will become the local backing device for our DRBD resource, also named `r0`, which corresponds to the device `/dev/drbd0`.
- This device will be the sole PV for another Volume Group, named `replicated`.
- This VG is to contain two more logical volumes named `foo`(4 GiB) and `bar`(6 GiB).

In order to enable this configuration, follow these steps:

- Set an appropriate `filter` option in your `/etc/lvm/lvm.conf`:

  ```
  filter = ["a|sd.*|", "a|drbd.*|", "r|.*|"]
  ```

  This filter expression accepts PV signatures found on any SCSI and DRBD devices, while rejecting (ignoring) all others.

  After modifying the `lvm.conf` file, you must run the `vgscan` command so LVM discards its configuration cache and re-scans devices for PV signatures.

- Disable the LVM cache by setting:

  ```
  write_cache_state = 0
  ```

  After disabling the LVM cache, make sure you remove any stale cache entries by deleting `/etc/lvm/cache/.cache`.

- Now, you may initialize your two SCSI partitions as PVs:

  ```
  # pvcreate /dev/sda1
  Physical volume "/dev/sda1" successfully created
  # pvcreate /dev/sdb1
  Physical volume "/dev/sdb1" successfully created
  ```

- The next step is creating your low-level VG named `local`, consisting of the two PVs you just initialized:

  ```
  # vgcreate local /dev/sda1 /dev/sda2
  Volume group "local" successfully created
  ```

- Now you may create your Logical Volume to be used as DRBD's backing device:

  ```
  # lvcreate --name r0 --size 10G local
  Logical volume "r0" created
  ```

- Repeat all steps, up to this point, on the peer node.
- Then, edit your `/etc/drbd.conf` to create a new resource named `r0`:

```
resource r0 {
  device /dev/drbd0;
  disk /dev/local/r0;
  meta-disk internal;
  on <host> { address <address>:<port>; }
  on <host> { address <address>:<port>; }
}
```

After you have created your new resource configuration, be sure to copy your `drbd.conf` contents to the peer node.

- After this, initialize your resource as described in Enabling your resource for the first time(on both nodes).
- Then, promote your resource (on one node):

```
# drbdadm primary r0
```

- Now, on the node where you just promoted your resource, initialize your DRBD device as a new Physical Volume:

```
# pvcreate /dev/drbd0
Physical volume "/dev/drbd0" successfully created
```

- Create your VG named `replicated`, using the PV you just initialized, on the same node:

```
# vgcreate replicated /dev/drbd0
Volume group "replicated" successfully created
```

- Finally, create your new Logical Volumes within this newly-created VG via

```
# lvcreate --name foo --size 4G replicated
Logical volume "foo" created
# lvcreate --name bar --size 6G replicated
Logical volume "bar" created
```

The Logical Volumes `foo` and `bar` will now be available as `/dev/replicated/foo` and `/dev/replicated/bar` on the local node.

16.6.1. Switching the VG to the other node

To make them available on the other node, first issue the following sequence of commands on the primary node:

```
# vgchange -a n replicated
0 logical volume(s) in volume group "replicated" now active
# drbdadm secondary r0
```

Then, issue these commands on the other (still secondary) node:

```
# drbdadm primary r0
# vgchange -a y replicated
2 logical volume(s) in volume group "replicated" now active
```

After this, the block devices `/dev/replicated/foo` and `/dev/replicated/bar` will be available on the other (now primary) node.

16.7. Highly available LVM with Pacemaker

The process of transferring volume groups between peers and making the corresponding logical volumes available can be automated. The Pacemaker LVM resource agent is designed for exactly that purpose.

In order to put an existing, DRBD-backed volume group under Pacemaker management, run the following commands in the `crm` shell:

Listing 13. Pacemaker configuration for DRBD-backed LVM Volume Group

```
primitive p_drbd_r0 ocf:linbit:drbd \
  params drbd_resource="r0" \
  op monitor interval="29s" role="Master" \
  op monitor interval="31s" role="Slave"
ms ms_drbd_r0 p_drbd_r0 \
  meta master-max="1" master-node-max="1" \
       clone-max="2" clone-node-max="1" \
       notify="true"
primitive p_lvm_r0 ocf:heartbeat:LVM \
  params volgrpname="r0"
colocation c_lvm_on_drbd inf: p_lvm_r0 ms_drbd_r0:Master
order o_drbd_before_lvm inf: ms_drbd_r0:promote p_lvm_r0:start
commit
```

After you have committed this configuration, Pacemaker will automatically make the `r0` volume group available on whichever node currently has the Primary (Master) role for the DRBD resource.

Chapter 17. Using GFS with DRBD

This chapter outlines the steps necessary to set up a DRBD resource as a block device holding a shared Global File System (GFS). It covers both GFS and GFS2.

In order to use GFS on top of DRBD, you must configure DRBD in dual-primary mode.

All cluster file systems *require* fencing - not only via the DRBD resource, but STONITH! A faulty member *must* be killed.

You'll want these settings:

```
net {
    fencing resource-and-stonith;
}
handlers {
    # Make sure the other node is confirmed
    # dead after this!
    outdate-peer "/sbin/kill-other-node.sh";
}
```

There must be *no* volatile caches! Please see https://fedorahosted.org/cluster/wiki/DRBD_Cookbook for some more information.

17.1. GFS primer

The Red Hat Global File System (GFS) is Red Hat's implementation of a concurrent-access shared storage file system. As any such filesystem, GFS allows multiple nodes to access the same storage device, in read/write fashion, simultaneously without risking data corruption. It does so by using a Distributed Lock Manager (DLM) which manages concurrent access from cluster members.

GFS was designed, from the outset, for use with conventional shared storage devices. Regardless, it is perfectly possible to use DRBD, in dual-primary mode, as a replicated storage device for GFS. Applications may benefit from reduced read/write latency due to the fact that DRBD normally reads from and writes to local storage, as opposed to the SAN devices GFS is normally configured to run from. Also, of course, DRBD adds an additional physical copy to every GFS filesystem, thus adding redundancy to the concept.

GFS makes use of a cluster-aware variant of LVM, termed Cluster Logical Volume Manager or CLVM. As such, some parallelism exists between using DRBD as the data storage for GFS, and using DRBD as a Physical Volume for conventional LVM.

GFS file systems are usually tightly integrated with Red Hat's own cluster management framework, the Red Hat Cluster. This chapter explains the use of DRBD in conjunction with GFS in the Red Hat Cluster context.

GFS, CLVM, and Red Hat Cluster are available in Red Hat Enterprise Linux (RHEL) and distributions derived from it, such as CentOS. Packages built from the same sources are also available in Debian GNU/Linux. This chapter assumes running GFS on a Red Hat Enterprise Linux system.

17.2. Creating a DRBD resource suitable for GFS

Since GFS is a shared cluster file system expecting concurrent read/write storage access from all cluster nodes, any DRBD resource to be used for storing a GFS filesystem must be configured in dual-primary mode. Also, it is recommended to use some of DRBD's features for automatic recovery from split brain. To do all this, include the following lines in the resource configuration:

```
resource <resource> {
  net {
    allow-two-primaries yes;
    after-sb-0pri discard-zero-changes;
    after-sb-1pri discard-secondary;
    after-sb-2pri disconnect;
    ...
  }
  ...
}
```

By configuring auto-recovery policies, you are configuring effectively configuring automatic data-loss! Be sure you understand the implications.

Once you have added these options to your freshly-configured resource, you may initialize your resource as you normally would. Since the `allow-two-primaries` option is set to `yes` for this resource, you will be able to promote the resourceto the primary role on two nodes.

17.3. Configuring LVM to recognize the DRBD resource

GFS uses CLVM, the cluster-aware version of LVM, to manage block devices to be used by GFS. In order to use CLVM with DRBD, ensure that your LVM configuration

- uses clustered locking. To do this, set the following option in `/etc/lvm/lvm.conf`:

  ```
  locking_type = 3
  ```

- scans your DRBD devices to recognize DRBD-based Physical Volumes (PVs). This applies as to conventional (non-clustered) LVM; see Configuring a DRBD resource as a Physical Volume for details.

17.4. Configuring your cluster to support GFS

After you have created your new DRBD resource and completed your initial cluster configuration, you must enable and start the following system services on both nodes of your GFS cluster:

- `cman` (this also starts `ccsd` and `fenced`),
- `clvmd`.

17.5. Creating a GFS filesystem

In order to create a GFS filesystem on your dual-primary DRBD resource, you must first initialize it as a Logical Volume for LVM.

Contrary to conventional, non-cluster-aware LVM configurations, the following steps must be completed on only one node due to the cluster-aware nature of CLVM:

```
# pvcreate /dev/drbd/by-res/<resource>/0
Physical volume "/dev/drbd<num>" successfully created
# vgcreate <vg-name> /dev/drbd/by-res/<resource>/0
Volume group "<vg-name>" successfully created
# lvcreate --size <size> --name <lv-name> <vg-name>
Logical volume "<lv-name>" created
```

This example assumes a single-volume resource.

CLVM will immediately notify the peer node of these changes; issuing `lvs` (or `lvdisplay`) on the peer node will list the newly created logical volume.

Now, you may proceed by creating the actual filesystem:

```
# mkfs -t gfs -p lock_dlm -j 2 /dev/<vg-name>/<lv-name>
```

Or, for a GFS2 filesystem:

```
# mkfs -t gfs2 -p lock_dlm -j 2 -t <cluster>:<name>
    /dev/<vg-name>/<lv-name>
```

The `-j` option in this command refers to the number of journals to keep for GFS. This must be identical to the number of nodes with concurrent *Primary* role in the GFS cluster; since DRBD does not support more than two *Primary* nodes until DRBD 9.1, the value to set here is always 2.

The `-t` option, applicable only for GFS2 filesystems, defines the lock table name. This follows the format *<cluster>:<name>*, where *<cluster>* must match your cluster name as defined in `/etc/cluster/cluster.conf`. Thus, only members of that cluster will be permitted to use the filesystem. By contrast, *<name>* is an arbitrary file system name unique in the cluster.

17.6. Using your GFS filesystem

After you have created your filesystem, you may add it to `/etc/fstab`:

```
/dev/<vg-name>/<lv-name> <mountpoint> gfs defaults 0 0
```

For a GFS2 filesystem, simply change the filesystem type:

```
/dev/<vg-name>/<lv-name> <mountpoint> gfs2 defaults 0 0
```

Do not forget to make this change on both (or, with DRBD 9.1, all) cluster nodes.

After this, you may mount your new filesystem by starting the `gfs` service (on both nodes):

```
# service gfs start
```

From then onwards, as long as you have DRBD configured to start automatically on system startup, before the RHCS services and the `gfs` service, you will be able to use this GFS file system as you would use one that is configured on traditional shared storage.

Chapter 18. Using OCFS2 with DRBD

This chapter outlines the steps necessary to set up a DRBD resource as a block device holding a shared Oracle Cluster File System, version 2 (OCFS2).

All cluster file systems *require* fencing - not only via the DRBD resource, but STONITH! A faulty member *must* be killed.

You'll want these settings:

```
net {
    fencing resource-and-stonith;
}
handlers {
    # Make sure the other node is confirmed
    # dead after this!
    outdate-peer "/sbin/kill-other-node.sh";
}
```

There must be *no* volatile caches! You might take a few hints of the page at https://fedorahosted.org/cluster/wiki/DRBD_Cookbook, although that's about GFS2, not OCFS2.

18.1. OCFS2 primer

The Oracle Cluster File System, version 2 (OCFS2) is a concurrent access shared storage file system developed by Oracle Corporation. Unlike its predecessor OCFS, which was specifically designed and only suitable for Oracle database payloads, OCFS2 is a general-purpose filesystem that implements most POSIX semantics. The most common use case for OCFS2 is arguably Oracle Real Application Cluster (RAC), but OCFS2 may also be used for load-balanced NFS clusters, for example.

Although originally designed for use with conventional shared storage devices, OCFS2 is equally well suited to be deployed on dual-Primary DRBD. Applications reading from the filesystem may benefit from reduced read latency due to the fact that DRBD reads from and writes to local storage, as opposed to the SAN devices OCFS2 otherwise normally runs on. In addition, DRBD adds redundancy to OCFS2 by adding an additional copy to every filesystem image, as opposed to just a single filesystem image that is merely shared.

Like other shared cluster file systems such as GFS, OCFS2 allows multiple nodes to access the same storage device, in read/write mode, simultaneously without risking data corruption. It does so by using a Distributed Lock Manager (DLM) which manages concurrent access from cluster nodes. The DLM itself uses a virtual file system (`ocfs2_dlmfs`) which is separate from the actual OCFS2 file systems present on the system.

OCFS2 may either use an intrinsic cluster communication layer to manage cluster membership and filesystem mount and unmount operation, or alternatively defer those tasks to the Pacemakercluster infrastructure.

OCFS2 is available in SUSE Linux Enterprise Server (where it is the primarily supported shared cluster file system), CentOS, Debian GNU/Linux, and Ubuntu Server Edition. Oracle also provides packages for Red Hat Enterprise Linux (RHEL). This chapter assumes running OCFS2 on a SUSE Linux Enterprise Server system.

18.2. Creating a DRBD resource suitable for OCFS2

Since OCFS2 is a shared cluster file system expecting concurrent read/write storage access from all cluster nodes, any DRBD resource to be used for storing a OCFS2 filesystem must be configured in dual-primary mode. Also, it is recommended to use some of DRBD's features for automatic recovery from split brain. To do all this, include the following lines in the resource configuration:

```
resource <resource> {
  net {
    # allow-two-primaries yes;
    after-sb-0pri discard-zero-changes;
    after-sb-1pri discard-secondary;
    after-sb-2pri disconnect;
    ...
  }
  ...
}
```

By setting auto-recovery policies, you are effectively configuring automatic data-loss! Be sure you understand the implications.

It is not recommended to set the `allow-two-primaries` option to `yes` upon initial configuration. You should do so after the initial resource synchronization has completed.

Once you have added these options to your freshly-configured resource, you may initialize your resource as you normally would. After you set the `allow-two-primaries` option to `yes` for this resource, you will be able to promote the resourceto the primary role on both nodes.

With DRBD 9.1 it will be possible to have more than two nodes in the *Primary* role; with DRBD 9.0 you can only use two primaries, but more nodes can be in *Secondary* role to provide more redundancy.

18.3. Creating an OCFS2 filesystem

Now, use OCFS2's `mkfs` implementation to create the file system:

```
# mkfs -t ocfs2 -N 2 -L ocfs2_drbd0 /dev/drbd0
mkfs.ocfs2 1.4.0
Filesystem label=ocfs2_drbd0
Block size=1024 (bits=10)
Cluster size=4096 (bits=12)
Volume size=205586432 (50192 clusters) (200768 blocks)
7 cluster groups (tail covers 4112 clusters, rest cover 7680 clusters)
Journal size=4194304
Initial number of node slots: 2
Creating bitmaps: done
Initializing superblock: done
Writing system files: done
Writing superblock: done
Writing backup superblock: 0 block(s)
Formatting Journals: done
Writing lost+found: done
mkfs.ocfs2 successful
```

This will create an OCFS2 file system with two node slots on `/dev/drbd0`, and set the filesystem label to `ocfs2_drbd0`. You may specify other options on `mkfs` invocation; please see the `mkfs.ocfs2` system manual page for details.

18.4. Pacemaker OCFS2 management

18.4.1. Adding a Dual-Primary DRBD resource to Pacemaker

An existing Dual-Primary DRBD resourcemay be added to Pacemaker resource management with the following `crm` configuration:

```
primitive p_drbd_ocfs2 ocf:linbit:drbd \
  params drbd_resource="ocfs2"
ms ms_drbd_ocfs2 p_drbd_ocfs2 \
  meta master-max=2 clone-max=2 notify=true
```

Note the `master-max=2` meta variable; it enables dual-Master mode for a Pacemaker master/slave set. This requires that `allow-two-primaries` is also set to `yes` in the DRBD configuration. Otherwise, Pacemaker will flag a configuration error during resource validation.

18.4.2. Adding OCFS2 management capability to Pacemaker

In order to manage OCFS2 and the kernel Distributed Lock Manager (DLM), Pacemaker uses a total of three different resource agents:

- `ocf:pacemaker:controld` — Pacemaker's interface to the DLM;
- `ocf:ocfs2:o2cb` — Pacemaker's interface to OCFS2 cluster management;
- `ocf:heartbeat:Filesystem` — the generic filesystem management resource agent which supports cluster file systems when configured as a Pacemaker clone.

You may enable all nodes in a Pacemaker cluster for OCFS2 management by creating a *cloned group* of resources, with the following `crm` configuration:

```
primitive p_controld ocf:pacemaker:controld
primitive p_o2cb ocf:ocfs2:o2cb
group g_ocfs2mgmt p_controld p_o2cb
clone cl_ocfs2mgmt g_ocfs2mgmt meta interleave=true
```

Once this configuration is committed, Pacemaker will start instances of the `controld` and `o2cb` resource types on all nodes in the cluster.

18.4.3. Adding an OCFS2 filesystem to Pacemaker

Pacemaker manages OCFS2 filesystems using the conventional `ocf:heartbeat:Filesystem` resource agent, albeit in clone mode. To put an OCFS2 filesystem under Pacemaker management, use the following `crm` configuration:

```
primitive p_fs_ocfs2 ocf:heartbeat:Filesystem \
  params device="/dev/drbd/by-res/ocfs2/0" directory="/srv/ocfs2" \
         fstype="ocfs2" options="rw,noatime"
clone cl_fs_ocfs2 p_fs_ocfs2
```

This example assumes a single-volume resource.

18.4.4. Adding required Pacemaker constraints to manage OCFS2 filesystems

In order to tie all OCFS2-related resources and clones together, add the following contraints to your Pacemaker configuration:

```
order o_ocfs2 ms_drbd_ocfs2:promote cl_ocfs2mgmt:start cl_fs_ocfs2:start
colocation c_ocfs2 cl_fs_ocfs2 cl_ocfs2mgmt ms_drbd_ocfs2:Master
```

18.5. Legacy OCFS2 management (without Pacemaker)

The information presented in this section applies to legacy systems where OCFS2 DLM support is not available in Pacemaker. It is preserved here for reference purposes only. New installations should always use the Pacemaker approach.

18.5.1. Configuring your cluster to support OCFS2

Creating the configuration file

OCFS2 uses a central configuration file, `/etc/ocfs2/cluster.conf`.

When creating your OCFS2 cluster, be sure to add both your hosts to the cluster configuration. The default port (7777) is usually an acceptable choice for cluster interconnect communications. If you choose any other port number, be sure to choose one that does not clash with an existing port used by DRBD (or any other configured TCP/IP).

If you feel less than comfortable editing the `cluster.conf` file directly, you may also use the `ocfs2console` graphical configuration utility which is usually more convenient. Regardless of the approach you selected, your `/etc/ocfs2/cluster.conf` file contents should look roughly like this:

```
node:
    ip_port = 7777
    ip_address = 10.1.1.31
    number = 0
    name = alice
    cluster = ocfs2

node:
    ip_port = 7777
    ip_address = 10.1.1.32
    number = 1
    name = bob
    cluster = ocfs2

cluster:
    node_count = 2
    name = ocfs2
```

When you have configured you cluster configuration, use `scp` to distribute the configuration to both nodes in the cluster.

Configuring the O2CB driver

SUSE Linux Enterprise systems

On SLES, you may utilize the `configure` option of the `o2cb` init script:

```
# /etc/init.d/o2cb configure
Configuring the O2CB driver.

This will configure the on-boot properties of the O2CB driver.
The following questions will determine whether the driver is loaded on
boot.  The current values will be shown in brackets ('[]').  Hitting
<ENTER> without typing an answer will keep that current value.  Ctrl-C
will abort.

Load O2CB driver on boot (y/n) [y]:
Cluster to start on boot (Enter "none" to clear) [ocfs2]:
Specify heartbeat dead threshold (>=7) [31]:
Specify network idle timeout in ms (>=5000) [30000]:
Specify network keepalive delay in ms (>=1000) [2000]:
Specify network reconnect delay in ms (>=2000) [2000]:
Use user-space driven heartbeat? (y/n) [n]:
Writing O2CB configuration: OK
Loading module "configfs": OK
Mounting configfs filesystem at /sys/kernel/config: OK
Loading module "ocfs2_nodemanager": OK
Loading module "ocfs2_dlm": OK
Loading module "ocfs2_dlmfs": OK
Mounting ocfs2_dlmfs filesystem at /dlm: OK
Starting O2CB cluster ocfs2: OK
```

.Debian GNU/Linux systems

On Debian, the `configure` option to `/etc/init.d/o2cb` is not available. Instead, reconfigure the `ocfs2-tools` package to enable the driver:

```
# dpkg-reconfigure -p medium -f readline ocfs2-tools
Configuring ocfs2-tools
Would you like to start an OCFS2 cluster (O2CB) at boot time? yes
Name of the cluster to start at boot time: ocfs2
The O2CB heartbeat threshold sets up the maximum time in seconds that a node
awaits for an I/O operation. After it, the node "fences" itself, and you will
probably see a crash.

It is calculated as the result of: (threshold - 1) x 2.

Its default value is 31 (60 seconds).

Raise it if you have slow disks and/or crashes with kernel messages like:

o2hb_write_timeout: 164 ERROR: heartbeat write timeout to device XXXX after NNNN
milliseconds
O2CB Heartbeat threshold: `31`
        Loading filesystem "configfs": OK
Mounting configfs filesystem at /sys/kernel/config: OK
Loading stack plugin "o2cb": OK
Loading filesystem "ocfs2_dlmfs": OK
Mounting ocfs2_dlmfs filesystem at /dlm: OK
Setting cluster stack "o2cb": OK
Starting O2CB cluster ocfs2: OK
```

18.5.2. Using your OCFS2 filesystem

When you have completed cluster configuration and created your file system, you may mount it as any other file system:

```
# mount -t ocfs2 /dev/drbd0 /shared
```

Your kernel log (accessible by issuing the command `dmesg`) should then contain a line similar to this one:

```
ocfs2: Mounting device (147,0) on (node 0, slot 0) with ordered data mode.
```

From that point forward, you should be able to simultaneously mount your OCFS2 filesystem on both your nodes, in read/write mode.

Chapter 19. Using Xen with DRBD

This chapter outlines the use of DRBD as a Virtual Block Device (VBD) for virtualization environments using the Xen hypervisor.

19.1. Xen primer

Xen is a virtualization framework originally developed at the University of Cambridge (UK), and later being maintained by XenSource, Inc. (now a part of Citrix). It is included in reasonably recent releases of most Linux distributions, such as Debian GNU/Linux (since version 4.0), SUSE Linux Enterprise Server (since release 10), Red Hat Enterprise Linux (since release 5), and many others.

Xen uses paravirtualization — a virtualization method involving a high degree of cooperation between the virtualization host and guest virtual machines — with selected guest operating systems for improved performance in comparison to conventional virtualization solutions (which are typically based on hardware emulation). Xen also supports full hardware emulation on CPUs that support the appropriate virtualization extensions; in Xen parlance, this is known as HVM ("hardware-assisted virtual machine").

At the time of writing, CPU extensions supported by Xen for HVM are Intel's Virtualization Technology (VT, formerly codenamed "Vanderpool"), and AMD's Secure Virtual Machine (SVM, formerly known as "Pacifica").

Xen supports *live migration*, which refers to the capability of transferring a running guest operating system from one physical host to another, without interruption.

When a DRBD resource is used as a replicated Virtual Block Device (VBD) for Xen, it serves to make the entire contents of a domU's virtual disk available on two servers, which can then be configured for automatic fail-over. That way, DRBD does not only provide redundancy for Linux servers (as in non-virtualized DRBD deployment scenarios), but also for any other operating system that can be virtualized under Xen — which, in essence, includes any operating system available on 32- or 64-bit Intel compatible architectures.

19.2. Setting DRBD module parameters for use with Xen

For Xen Domain-0 kernels, it is recommended to load the DRBD module with the parameter `disable_sendpage` set to `1`. To do so, create (or open) the file `/etc/modprobe.d/drbd.conf` and enter the following line:

```
options drbd disable_sendpage=1
```

19.3. Creating a DRBD resource suitable to act as a Xen VBD

Configuring a DRBD resource that is to be used as a Virtual Block Device for Xen is fairly straightforward — in essence, the typical configuration matches that of a DRBD resource being used for any other purpose. However, if you want to enable live migration for your guest instance, you need to enable dual-primary modefor this resource:

```
resource <resource> {
  net {
    allow-two-primaries yes;
    ...
  }
  ...
}
```

Enabling dual-primary mode is necessary because Xen, before initiating live migration, checks for write access on all VBDs a resource is configured to use on both the source and the destination host for the migration.

19.4. Using DRBD VBDs

In order to use a DRBD resource as the virtual block device, you must add a line like the following to your Xen domU configuration:

```
disk = [ 'drbd:<resource>,xvda,w' ]
```

This example configuration makes the DRBD resource named *resource* available to the domU as `/dev/xvda` in read/write mode (`w`).

Of course, you may use multiple DRBD resources with a single domU. In that case, simply add more entries like the one provided in the example to the `disk` option, separated by commas.

There are three sets of circumstances under which you cannot use this approach:

- You are configuring a fully virtualized (HVM) domU.
- You are installing your domU using a graphical installation utility, *and* that graphical installer does not support the `drbd:` syntax.
- You are configuring a domU without the `kernel`, `initrd`, and `extra` options, relying instead on `bootloader` and `bootloader_args` to use a Xen pseudo-bootloader, *and* that pseudo-bootloader does not support the `drbd:` syntax.
    - pygrub+ (prior to Xen 3.3) and `domUloader.py` (shipped with Xen on SUSE Linux Enterprise Server 10) are two examples of pseudo-bootloaders that do not support the `drbd:` virtual block device configuration syntax.
    - `pygrub` from Xen 3.3 forward, and the `domUloader.py` version that ships with SLES 11 *do* support this syntax.

Under these circumstances, you must use the traditional `phy:` device syntax and the DRBD device name that is associated with your resource, not the resource name. That, however, requires that you manage DRBD state transitions outside Xen, which is a less flexible approach than that provided by the `drbd` resource type.

19.5. Starting, stopping, and migrating DRBD-backed domU's

Starting the domU

Once you have configured your DRBD-backed domU, you may start it as you would any other domU:

```
# xm create <domU>
Using config file "/etc/xen/<domU>".
Started domain <domU>
```

In the process, the DRBD resource you configured as the VBD will be promoted to the primary role, and made accessible to Xen as expected.

Stopping the domU

This is equally straightforward:

```
# xm shutdown -w <domU>
Domain <domU> terminated.
```

Again, as you would expect, the DRBD resource is returned to the secondary role after the domU is successfully shut down.

Migrating the domU

This, too, is done using the usual Xen tools:

```
# xm migrate --live <domU> <destination-host>
```

In this case, several administrative steps are automatically taken in rapid succession: * The resource is promoted to the primary role on *destination-host*. * Live migration of *domU* is initiated on the local host. * When migration to the destination host has completed, the resource is demoted to the secondary role locally.

The fact that both resources must briefly run in the primary role on both hosts is the reason for having to configure the resource in dual-primary mode in the first place.

19.6. Internals of DRBD/Xen integration

Xen supports two Virtual Block Device types natively:

`phy`

This device type is used to hand "physical" block devices, available in the host environment, off to a guest domU in an essentially transparent fashion.

`file`

This device type is used to make file-based block device images available to the guest domU. It works by creating a loop block device from the original image file, and then handing that block device off to the domU in much the same fashion as the `phy` device type does.

If a Virtual Block Device configured in the `disk` option of a domU configuration uses any prefix other than `phy:`, `file:`, or no prefix at all (in which case Xen defaults to using the `phy` device type), Xen expects to find a helper script named `block-`*prefix* in the Xen scripts directory, commonly `/etc/xen/scripts`.

The DRBD distribution provides such a script for the `drbd` device type, named `/etc/xen/scripts/block-drbd`. This script handles the necessary DRBD resource state transitions as described earlier in this chapter.

19.7. Integrating Xen with Pacemaker

In order to fully capitalize on the benefits provided by having a DRBD-backed Xen VBD's, it is recommended to have Heartbeat manage the associated domU's as Heartbeat resources.

You may configure a Xen domU as a Pacemaker resource, and automate resource failover. To do so, use the Xen OCF resource agent. If you are using the `drbd` Xen device type described in this chapter, you will *not* need to configure any separate `drbd` resource for use by the Xen cluster resource. Instead, the `block-drbd` helper script will do all the necessary resource transitions for you.

Optimizing DRBD performance

Chapter 20. Measuring block device performance

20.1. Measuring throughput

When measuring the impact of using DRBD on a system's I/O throughput, the *absolute* throughput the system is capable of is of little relevance. What is much more interesting is the *relative* impact DRBD has on I/O performance. Thus it is always necessary to measure I/O throughput both with and without DRBD.

The tests described in this section are intrusive; they overwrite data and bring DRBD devices out of sync. It is thus vital that you perform them only on scratch volumes which can be discarded after testing has completed.

I/O throughput estimation works by writing significantly large chunks of data to a block device, and measuring the amount of time the system took to complete the write operation. This can be easily done using a fairly ubiquitous utility, `dd`, whose reasonably recent versions include a built-in throughput estimation.

A simple `dd`-based throughput benchmark, assuming you have a scratch resource named `test`, which is currently connected and in the secondary role on both nodes, is one like the following:

```
# TEST_RESOURCE=test
# TEST_DEVICE=$(drbdadm sh-dev $TEST_RESOURCE | head -1)
# TEST_LL_DEVICE=$(drbdadm sh-ll-dev $TEST_RESOURCE | head -1)
# drbdadm primary $TEST_RESOURCE
# for i in $(seq 5); do
    dd if=/dev/zero of=$TEST_DEVICE bs=1M count=512 oflag=direct
  done
# drbdadm down $TEST_RESOURCE
# for i in $(seq 5); do
    dd if=/dev/zero of=$TEST_LL_DEVICE bs=1M count=512 oflag=direct
  done
```

This test simply writes 512MiB of data to your DRBD device, and then to its backing device for comparison. Both tests are repeated 5 times each to allow for some statistical averaging. The relevant result is the throughput measurements generated by `dd`.

For freshly enabled DRBD devices, it is normal to see slightly reduced performance on the first `dd` run. This is due to the Activity Log being "cold", and is no cause for concern.

See our Optimizing DRBD throughput chapter for some performance numbers.

20.2. Measuring latency

Latency measurements have objectives completely different from throughput benchmarks: in I/O latency tests, one writes a very small chunk of data (ideally the smallest chunk of data that the system can deal with), and observes the time it takes to complete that write. The process is usually repeated several times to account for normal statistical fluctuations.

Just as throughput measurements, I/O latency measurements may be performed using the ubiquitous `dd` utility, albeit with different settings and an entirely different focus of observation.

Provided below is a simple `dd`-based latency micro-benchmark, assuming you have a scratch resource named `test` which is currently connected and in the secondary role on both nodes:

```
# TEST_RESOURCE=test
# TEST_DEVICE=$(drbdadm sh-dev $TEST_RESOURCE | head -1)
# TEST_LL_DEVICE=$(drbdadm sh-ll-dev $TEST_RESOURCE | head -1)
# drbdadm primary $TEST_RESOURCE
# dd if=/dev/zero of=$TEST_DEVICE bs=4k count=1000 oflag=direct
# drbdadm down $TEST_RESOURCE
# dd if=/dev/zero of=$TEST_LL_DEVICE bs=4k count=1000 oflag=direct
```

This test writes 1,000 chunks with 4kiB each to your DRBD device, and then to its backing device for comparison. 4096 bytes is the smallest block size that a Linux system (on all architectures except s390), modern hard disks, and SSDs, are expected to handle.

It is important to understand that throughput measurements generated by dd are completely irrelevant for this test; what is important is the *time* elapsed during the completion of said 1,000 writes. Dividing this time by 1,000 gives the average latency of a single block write.

This is the *worst-case*, in that it is single-threaded and does one write strictly after the one before, ie. runs with an I/O-depth of 1. Please take a look at Latency vs. IOPs.

Furthermore, see our Optimizing DRBD latency chapter for some typical performance values.

Chapter 21. Optimizing DRBD throughput

This chapter deals with optimizing DRBD throughput. It examines some hardware considerations with regard to throughput optimization, and details tuning recommendations for that purpose.

21.1. Hardware considerations

DRBD throughput is affected by both the bandwidth of the underlying I/O subsystem (disks, controllers, and corresponding caches), and the bandwidth of the replication network.

I/O subsystem throughput

I/O subsystem throughput is determined, largely, by the number and type of storage units (disks, SSDs, other Flash storage [like FusionIO], ...) that can be written to in parallel. A single, reasonably recent, SCSI or SAS disk will typically allow streaming writes of roughly 40MiB/s to the single disk; an SSD will do 300MiB/s; one of the recent Flash storages (NVMe) will be at 1GiB/s. When deployed in a striping configuration, the I/O subsystem will parallelize writes across disks, effectively multiplying a single disk's throughput by the number of stripes in the configuration. Thus the same, 40MB/s disks will allow effective throughput of 120MB/s in a RAID-0 or RAID-1+0 configuration with three stripes, or 200MB/s with five stripes; with SSDs and/or NVMe you can easily get to 1GiB/sec.

A RAID-controller with RAM and a BBU can speed up short spikes (by buffering them), and so too-short benchmark tests might show speeds like 1GiB/s too; for sustained writes its buffers will just run full, and then not be of much help, though.

Disk *mirroring* (RAID-1) in hardware typically has little, if any, effect on throughput. Disk *striping with parity* (RAID-5) does have an effect on throughput, usually an adverse one when compared to striping; RAID-5 and RAID-6 in software even more so.

Network throughput

Network throughput is usually determined by the amount of traffic present on the network, and on the throughput of any routing/switching infrastructure present. These concerns are, however, largely irrelevant in DRBD replication links which are normally dedicated, back-to-back network connections. Thus, network throughput may be improved either by switching to a higher-throughput hardware (such as 10 Gigabit Ethernet, or 56GiB Infiniband), or by using link aggregation over several network links, as one may do using the Linux `bonding` network driver.

21.2. Throughput overhead expectations

When estimating the throughput overhead associated with DRBD, it is important to consider the following natural limitations:

- DRBD throughput is limited by that of the raw I/O subsystem.
- DRBD throughput is limited by the available network bandwidth.

The *lower* of these two establishes the theoretical throughput *maximum* available to DRBD. DRBD then reduces that throughput number by its additional overhead, which can be expected to be less than 3 percent.

- Consider the example of two cluster nodes containing I/O subsystems capable of 600 MB/s throughput, with a Gigabit Ethernet link available between them. Gigabit Ethernet can be expected to produce 110 MB/s throughput for TCP connections, thus the network connection would be the bottleneck in this configuration and one would expect about 110 MB/s maximum DRBD throughput.
- By contrast, if the I/O subsystem is capable of only 80 MB/s for sustained writes, then it constitutes the bottleneck, and you should expect only about 77 MB/s maximum DRBD throughput.

21.3. Tuning recommendations

DRBD offers a number of configuration options which may have an effect on your system's throughput. This section list some recommendations for tuning for throughput. However, since throughput is largely hardware dependent, the

effects of tweaking the options described here may vary greatly from system to system. It is important to understand that these recommendations should not be interpreted as "silver bullets" which would magically remove any and all throughput bottlenecks.

21.3.1. Setting `max-buffers` and `max-epoch-size`

These options affect write performance on the secondary nodes. `max-buffers` is the maximum number of buffers DRBD allocates for writing data to disk while `max-epoch-size` is the maximum number of write requests permitted between two write barriers. `max-buffers` must be equal or bigger to `max-epoch-size` to increase performance. The default for both is 2048; setting it to around 8000 should be fine for most reasonably high-performance hardware RAID controllers.

```
resource <resource> {
  net {
    max-buffers    8000;
    max-epoch-size 8000;
    ...
  }
  ...
}
```

21.3.2. Tuning the TCP send buffer size

The TCP send buffer is a memory buffer for outgoing TCP traffic. By default, it is set to a size of 128 KiB. For use in high-throughput networks (such as dedicated Gigabit Ethernet or load-balanced bonded connections), it may make sense to increase this to a size of 2MiB, or perhaps even more. Send buffer sizes of more than 16MiB are generally not recommended (and are also unlikely to produce any throughput improvement).

```
resource <resource> {
  net {
    sndbuf-size 2M;
    ...
  }
  ...
}
```

DRBD also supports TCP send buffer auto-tuning. After enabling this feature, DRBD will dynamically select an appropriate TCP send buffer size. TCP send buffer auto tuning is enabled by simply setting the buffer size to zero:

```
resource <resource> {
  net {
    sndbuf-size 0;
    ...
  }
  ...
}
```

Please note that your `sysctl`'s settings `net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem` will still influence the behaviour; you should check these settings, and perhaps set them similar to `131072 1048576 16777216` (minimum 128kiB, default 1MiB, max 16MiB).

`net.ipv4.tcp_mem` is a different beast, with a different unit - do not touch, wrong values can easily push your machine into out-of-memory situations!

21.3.3. Tuning the Activity Log size

If the application using DRBD is write intensive in the sense that it frequently issues small writes scattered across the device, it is usually advisable to use a fairly large activity log. Otherwise, frequent metadata updates may be detrimental to write performance.

```
resource <resource> {
  disk {
    al-extents 6007;
    ...
  }
  ...
}
```

21.3.4. Disabling barriers and disk flushes

The recommendations outlined in this section should be applied *only* to systems with non-volatile (battery backed) controller caches.

Systems equipped with battery backed write cache come with built-in means of protecting data in the face of power failure. In that case, it is permissible to disable some of DRBD's own safeguards created for the same purpose. This may be beneficial in terms of throughput:

```
resource <resource> {
  disk {
    disk-barrier no;
    disk-flushes no;
    ...
  }
  ...
}
```

21.4. Achieving better Read Performance via increased Redundancy

As detailed in the man page of `drbd.conf` under `read-balancing`, you can increase your read performance by adding more copies of your data.

As a ballpark figure: with a single node processing read requests, `fio` on a *FusionIO* card gave us 100k IOPs; after enabling `read-balancing`, the performance jumped to 180k IOPs, ie. +80%!

So, in case you're running a read-mostly workload (big databases with lots of random reads), it might be worth a try to turn `read-balancing` on - and, perhaps, add another copy for still more read IO throughput.

Chapter 22. Optimizing DRBD latency

This chapter deals with optimizing DRBD latency. It examines some hardware considerations with regard to latency minimization, and details tuning recommendations for that purpose.

22.1. Hardware considerations

DRBD latency is affected by both the latency of the underlying I/O subsystem (disks, controllers, and corresponding caches), and the latency of the replication network.

I/O subsystem latency

For *rotating media* the I/O subsystem latency is primarily a function of disk rotation speed. Thus, using fast-spinning disks is a valid approach for reducing I/O subsystem latency.

For *solid state media* (like SSDs) the Flash storage controller is the determining factor; the next most important thing is the amount of unused capacity. Using DRBD's Trim/Discard support will help you provide the controller with the needed information which blocks it can recycle. That way, when a write requests comes in, it can use a block that got cleaned ahead-of-time and doesn't have to wait **now** until there's space available [14: On low-end hardware you can help that a bit by reserving some space - just keep 10% to 20% of the total space unpartitioned.].

Likewise, the use of a battery-backed write cache (BBWC) reduces write completion times, also reducing write latency. Most reasonable storage subsystems come with some form of battery-backed cache, and allow the administrator to configure which portion of this cache is used for read and write operations. The recommended approach is to disable the disk read cache completely and use all available cache memory for the disk write cache.

Network latency

Network latency is, in essence, the packet round-trip time (RTT) between hosts. It is influenced by a number of factors, most of which are irrelevant on the dedicated, back-to-back network connections recommended for use as DRBD replication links. Thus, it is sufficient to accept that a certain amount of latency always exists in network links, which typically is on the order of 100 to 200 microseconds (µs) packet RTT for Gigabit Ethernet.

Network latency may typically be pushed below this limit only by using lower-latency network protocols, such as running DRBD over Dolphin Express using Dolphin SuperSockets, or a 10GBe direct connection; these are typically in the 50µs range. Even better is InfiniBand, which provides even lower latencies.

22.2. Latency overhead expectations

As for throughput, when estimating the latency overhead associated with DRBD, there are some important natural limitations to consider:

- DRBD latency is bound by that of the raw I/O subsystem.
- DRBD latency is bound by the available network latency.

The *sum* of the two establishes the theoretical latency *minimum* incurred to DRBD [15: for protocol C, because the other node(s) have to write to stable storage, too]. DRBD then adds to that latency a slight additional latency overhead, which can be expected to be less than 1 percent.

- Consider the example of a local disk subsystem with a write latency of 3ms and a network link with one of 0.2ms. Then the expected DRBD latency would be 3.2 ms or a roughly 7-percent latency increase over just writing to a local disk.

Latency may be influenced by a number of other factors, including CPU cache misses, context switches, and others.

22.3. Latency vs. IOPs

IOPs is the abbreviation of "*I/O operations per second*".

Marketing typically doesn't like numbers that get smaller; press releases aren't written with "*Latency reduced by 10µs, from 50µs to 40µs now!*" in mind, they like "*Performance increased by 25%, from 20000 to now 25000 IOPs*" much more. Therefore IOPs were invented - to get a number that says "higher is better".

So, in other words, IOPs are the reciprocal of latency. What you'll have to keep in mind is that the method given in Measuring latency gives you the latency resp. the number of IOPs for a purely sequential, single-threaded IO load, while most other documentation will give numbers for some highly parallel load [16: Like in "*16 threads, IO-depth of 32*" - this means that 512 I/O-requests are being done in parallel!], because this gives much "prettier" numbers. With that kind of trick DRBD does offer you 100000 IOPs, too!

So, please don't shy away from measuring serialized, single-threaded latency. If you want lots of IOPs, run the `fio` utility with `threads=8` and an `iodepth=16`, or some similar settings... But please keep in mind that these number will not have any meaning to your setup, unless you're driving a database with many tens or hundreds of client connections active at the same time.

22.4. Tuning recommendations

22.4.1. Setting DRBD's CPU mask

DRBD allows for setting an explicit CPU mask for its kernel threads. This is particularly beneficial for applications which would otherwise compete with DRBD for CPU cycles.

The CPU mask is a number in whose binary representation the least significant bit represents the first CPU, the second-least significant bit the second, and so forth. A set bit in the bitmask implies that the corresponding CPU may be used by DRBD, whereas a cleared bit means it must not. Thus, for example, a CPU mask of 1 (`00000001`) means DRBD may use the first CPU only. A mask of 12 (`00001100`) implies DRBD may use the third and fourth CPU.

An example CPU mask configuration for a resource may look like this:

```
resource <resource> {
  options {
    cpu-mask 2;
    ...
  }
  ...
}
```

Of course, in order to minimize CPU competition between DRBD and the application using it, you need to configure your application to use only those CPUs which DRBD does not use.

Some applications may provide for this via an entry in a configuration file, just like DRBD itself. Others include an invocation of the `taskset` command in an application init script.

It makes sense to keep the DRBD threads running on the same L2/L3 caches.

But: the numbering of CPUs doesn't have to correlate with the physical partitioning. You can try the `lstopo` (or `hwloc-ls`) program for X11 or `hwloc-info -v -p` for console output to get an overview of the topology.

22.4.2. Modifying the network MTU

It may be beneficial to change the replication network's maximum transmission unit (MTU) size to a value higher than the default of 1500 bytes. Colloquially, this is referred to as "enabling Jumbo frames".

The MTU may be changed using the following commands:

```
# ifconfig <interface> mtu <size>
```

or

```
# ip link set <interface> mtu <size>
```

<interface> refers to the network interface used for DRBD replication. A typical value for *<size>* would be 9000 (bytes).

22.4.3. Enabling the deadline I/O scheduler

When used in conjunction with high-performance, write back enabled hardware RAID controllers, DRBD latency may benefit greatly from using the simple deadline I/O scheduler, rather than the CFQ scheduler. The latter is typically enabled by default.

Modifications to the I/O scheduler configuration may be performed via the `sysfs` virtual file system, mounted at `/sys`. The scheduler configuration is in `/sys/block/`*`device`*, where *<device>* is the backing device DRBD uses.

Enabling the deadline scheduler works via the following command:

```
# echo deadline > /sys/block/<device>/queue/scheduler
```

You may then also set the following values, which may provide additional latency benefits:

- Disable front merges:

  ```
  # echo 0 > /sys/block/<device>/queue/iosched/front_merges
  ```

- Reduce read I/O deadline to 150 milliseconds (the default is 500ms):

  ```
  # echo 150 > /sys/block/<device>/queue/iosched/read_expire
  ```

- Reduce write I/O deadline to 1500 milliseconds (the default is 3000ms):

  ```
  # echo 1500 > /sys/block/<device>/queue/iosched/write_expire
  ```

If these values effect a significant latency improvement, you may want to make them permanent so they are automatically set at system startup. Debian and Ubuntu systems provide this functionality via the `sysfsutils` package and the `/etc/sysfs.conf` configuration file.

You may also make a global I/O scheduler selection by passing the `elevator` option via your kernel command line. To do so, edit your boot loader configuration (normally found in `/etc/default/grub` if you are using the GRUB bootloader) and add `elevator=deadline` to your list of kernel boot options.

Learning more

Chapter 23. DRBD Internals

This chapter gives *some* background information about some of DRBD's internal algorithms and structures. It is intended for interested users wishing to gain a certain degree of background knowledge about DRBD. It does not dive into DRBD's inner workings deep enough to be a reference for DRBD developers. For that purpose, please refer to the papers listed in Publications, and of course to the comments in the DRBD source code.

23.1. DRBD meta data

DRBD stores various pieces of information about the data it keeps in a dedicated area. This metadata includes:

- the size of the DRBD device,
- the Generation Identifier (GI, described in detail in Generation Identifiers),
- the Activity Log (AL, described in detail in The Activity Log).
- the quick-sync bitmap (described in detail in The quick-sync bitmap),

This metadata may be stored *internally* or *externally*. Which method is used is configurable on a per-resource basis.

23.1.1. Internal meta data

Configuring a resource to use internal meta data means that DRBD stores its meta data on the same physical lower-level device as the actual production data. It does so by setting aside an area at the *end* of the device for the specific purpose of storing metadata.

Advantage

Since the meta data are inextricably linked with the actual data, no special action is required from the administrator in case of a hard disk failure. The meta data are lost together with the actual data and are also restored together.

Disadvantage

In case of the lower-level device being a single physical hard disk (as opposed to a RAID set), internal meta data may negatively affect write throughput. The performance of write requests by the application may trigger an update of the meta data in DRBD. If the meta data are stored on the same magnetic disk of a hard disk, the write operation may result in two additional movements of the write/read head of the hard disk.

If you are planning to use internal meta data in conjunction with an existing lower-level device that already has data which you wish to preserve, you *must* account for the space required by DRBD's meta data.

Otherwise, upon DRBD resource creation, the newly created metadata would overwrite data at the end of the lower-level device, potentially destroying existing files in the process.

To avoid that, you must do one of the following things:

- Enlarge your lower-level device. This is possible with any logical volume management facility (such as LVM) as long as you have free space available in the corresponding volume group. It may also be supported by hardware storage solutions.
- Shrink your existing file system on your lower-level device. This may or may not be supported by your file system.
- If neither of the two are possible, use external meta data instead.

To estimate the amount by which you must enlarge your lower-level device or shrink your file system, see Estimating meta data size.

23.1.2. External meta data

External meta data is simply stored on a separate, dedicated block device distinct from that which holds your production data.

Advantage

For some write operations, using external meta data produces a somewhat improved latency behavior.

Disadvantage

Meta data are not inextricably linked with the actual production data. This means that manual intervention is required in the case of a hardware failure destroying just the production data (but not DRBD meta data), to effect a full data sync from the surviving node onto the subsequently replaced disk.

Use of external meta data is also the only viable option if *all* of the following apply:

- You are using DRBD to duplicate an existing device that already contains data you wish to preserve, *and*
- that existing device does not support enlargement, *and*
- the existing file system on the device does not support shrinking.

To estimate the required size of the block device dedicated to hold your device meta data, see Estimating meta data size.

External meta data requires a minimum of a 1MB device size.

23.1.3. Estimating meta data size

You may calculate the exact space requirements for DRBD's meta data using the following formula:

$$M_s = \lceil \frac{C_s}{2^{18}} \rceil * 8 * N + 72$$

Figure 18. Calculating DRBD meta data size (exactly)

C_s is the data device size in sectors, and N is the number of peers.

You may retrieve the device size (in bytes) by issuing `blockdev --getsize64 <device>`; to convert to MB, divide by 1048576 (= 2^{20} or 1024^2).

In practice, you may use a reasonably good approximation, given below. Note that in this formula, the unit is megabytes, not sectors:

$$M_{MB} < \frac{C_{MB}}{32768} * N + 1$$

Figure 19. Estimating DRBD meta data size (approximately)

23.2. Generation Identifiers

DRBD uses *generation identifiers* (GIs) to identify "generations" of replicated data.

This is DRBD's internal mechanism used for

- determining whether the two nodes are in fact members of the same cluster (as opposed to two nodes that were connected accidentally),
- determining the direction of background re-synchronization (if necessary),

- determining whether full re-synchronization is necessary or whether partial re-synchronization is sufficient,
- identifying split brain.

23.2.1. Data generations

DRBD marks the start of a new *data generation* at each of the following occurrences:

- The initial device full sync,
- a disconnected resource switching to the primary role,
- a resource in the primary role disconnecting.

Thus, we can summarize that whenever a resource is in the *Connected* connection state, and both nodes' disk state is *UpToDate*, the current data generation on both nodes is the same. The inverse is also true. Note that the current implementation uses the lowest bit to encode the role of the node (Primary/Secondary). Therefore, the lowest bit might be different on distinct nodes even if they are considered to have the same data generation.

Every new data generation is identified by an 8-byte, universally unique identifier (UUID).

23.2.2. The generation identifier tuple

DRBD keeps some pieces of information about current and historical data generations in the local resource meta data:

Current UUID

This is the generation identifier for the current data generation, as seen from the local node's perspective. When a resource is *Connected* and fully synchronized, the current UUID is identical between nodes.

Bitmap UUIDs

This is the UUID of the generation against which this on-disk bitmap is tracking changes (per remote host). Like the on-disk sync bitmap itself, this identifier is only relevant while the remote host is disconnected.

Historical UUIDs

These are the identifiers of data generations preceding the current one, sized to have one slot per (possible) remote host.

Collectively, these items are referred to as the *generation identifier tuple*, or "*GI tuple*" for short.

23.2.3. How generation identifiers change

Start of a new data generation

When a node in *Primary* role loses connection to its peer (either by network failure or manual intervention), DRBD modifies its local generation identifiers in the following manner:

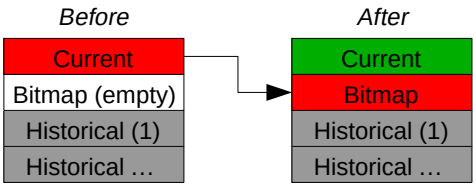


Figure 20. GI tuple changes at start of a new data generation

1. The primary creates a new UUID for the new data generation. This becomes the *new* current UUID for the primary node.
2. The *previous* current UUID now refers to the generation the bitmap is tracking changes against, so it becomes the new bitmap UUID for the primary node.
3. On the secondary node(s), the GI tuple remains unchanged.

Completion of re-synchronization

When re-synchronization concludes, the synchronization target adopts the entire GI tuple from the synchronization source.

The synchronization source keeps the same set, and doesn't generate new UUIDs.

23.2.4. How DRBD uses generation identifiers

When a connection between nodes is established, the two nodes exchange their currently available generation identifiers, and proceed accordingly. A number of possible outcomes exist:

Current UUIDs empty on both nodes

The local node detects that both its current UUID and the peer's current UUID are empty. This is the normal occurrence for a freshly configured resource that has not had the initial full sync initiated. No synchronization takes place; it has to be started manually.

Current UUIDs empty on one node

The local node detects that the peer's current UUID is empty, and its own is not. This is the normal case for a freshly configured resource on which the initial full sync has just been initiated, the local node having been selected as the initial synchronization source. DRBD now sets all bits in the on-disk sync bitmap (meaning it considers the entire device out-of-sync), and starts synchronizing as a synchronization source. In the opposite case (local current UUID empty, peer's non-empty), DRBD performs the same steps, except that the local node becomes the synchronization target.

Equal current UUIDs

The local node detects that its current UUID and the peer's current UUID are non-empty and equal. This is the normal occurrence for a resource that went into disconnected mode at a time when it was in the secondary role, and was not promoted on either node while disconnected. No synchronization takes place, as none is necessary.

Bitmap UUID matches peer's current UUID

The local node detects that its bitmap UUID matches the peer's current UUID, and that the peer's bitmap UUID is empty. This is the normal and expected occurrence after a secondary node failure, with the local node being in the primary role. It means that the peer never became primary in the meantime and worked on the basis of the same data generation all along. DRBD now initiates a normal, background re-synchronization, with the local node becoming the synchronization source. If, conversely, the local node detects that *its* bitmap UUID is empty, and that the *peer's* bitmap matches the local node's current UUID, then that is the normal and expected occurrence after a failure of the local node. Again, DRBD now initiates a normal, background re-synchronization, with the local node becoming the synchronization target.

Current UUID matches peer's historical UUID

The local node detects that its current UUID matches one of the peer's historical UUID's. This implies that while the two data sets share a common ancestor, and the peer node has the up-to-date data, the information kept in the peer node's bitmap is outdated and not usable. Thus, a normal synchronization would be insufficient. DRBD now marks the entire device as out-of-sync and initiates a full background re-synchronization, with the local node becoming the synchronization target. In the opposite case (one of the local node's historical UUID matches the peer's current UUID), DRBD performs the same steps, except that the local node becomes the synchronization source.

Bitmap UUIDs match, current UUIDs do not

The local node detects that its current UUID differs from the peer's current UUID, and that the bitmap UUID's match. This is split brain, but one where the data generations have the same parent. This means that DRBD invokes split brain auto-recovery strategies, if configured. Otherwise, DRBD disconnects and waits for manual split brain resolution.

Neither current nor bitmap UUIDs match

The local node detects that its current UUID differs from the peer's current UUID, and that the bitmap UUID's *do not* match. This is split brain with unrelated ancestor generations, thus auto-recovery strategies, even if configured, are moot. DRBD disconnects and waits for manual split brain resolution.

No UUIDs match

Finally, in case DRBD fails to detect even a single matching element in the two nodes' GI tuples, it logs a warning about

unrelated data and disconnects. This is DRBD's safeguard against accidental connection of two cluster nodes that have never heard of each other before.

23.3. The Activity Log

23.3.1. Purpose

During a write operation DRBD forwards the write operation to the local backing block device, but also sends the data block over the network. These two actions occur, for all practical purposes, simultaneously. Random timing behavior may cause a situation where the write operation has been completed, but the transmission via the network has not yet taken place, or vice versa.

If, at this moment, the active node fails and fail-over is being initiated, then this data block is out of sync between nodes — it has been written on the failed node prior to the crash, but replication has not yet completed. Thus, when the node eventually recovers, this block must be removed from the data set during subsequent synchronization. Otherwise, the crashed node would be "one write ahead" of the surviving node, which would violate the "all or nothing" principle of replicated storage. This is an issue that is not limited to DRBD, in fact, this issue exists in practically all replicated storage configurations. Many other storage solutions (just as DRBD itself, prior to version 0.7) thus require that after a failure of the active node the data must be fully synchronized after its recovery.

DRBD's approach, since version 0.7, is a different one. The *activity log* (AL), stored in the meta data area, keeps track of those blocks that have "recently" been written to. Colloquially, these areas are referred to as *hot extents*.

If a temporarily failed node that was in active mode at the time of failure is synchronized, only those hot extents highlighted in the AL need to be synchronized (plus any blocks marked in the bitmap on the now-active peer), rather than the full device. This drastically reduces synchronization time after an active node crash.

23.3.2. Active extents

The activity log has a configurable parameter, the number of active extents. Every active extent adds 4MiB to the amount of data being retransmitted after a Primary crash. This parameter must be understood as a compromise between the following opposites:

Many active extents

Keeping a large activity log improves write throughput. Every time a new extent is activated, an old extent is reset to inactive. This transition requires a write operation to the meta data area. If the number of active extents is high, old active extents are swapped out fairly rarely, reducing meta data write operations and thereby improving performance.

Few active extents

Keeping a small activity log reduces synchronization time after active node failure and subsequent recovery.

23.3.3. Selecting a suitable Activity Log size

Consideration of the number of extents should be based on the desired synchronization time at a given synchronization rate. The number of active extents can be calculated as follows:

$$E = \frac{R * t_{sync}}{4MiB}$$

Figure 21. Active extents calculation based on sync rate and target sync time

R is the synchronization rate, given in MiB/s. t_{sync} is the target synchronization time, in seconds. E is the resulting number of active extents.

To provide an example, suppose the cluster has an I/O subsystem with a throughput rate of 200 MiByte/s that was configured to a synchronization rate (R) of 60 MiByte/s, and we want to keep the target synchronization time (t_{sync}) at 4 minutes or 240 seconds:

Figure 22. Active extents calculation based on sync rate and target sync time (example)

On a final note, DRBD 9 needs to keep an AL even on the Secondary nodes, as their data might be used to synchronize other Secondary nodes.

23.4. The quick-sync bitmap

The quick-sync bitmap is the internal data structure which DRBD uses, on a per-resource per-peer basis, to keep track of blocks being in sync (identical on both nodes) or out-of sync. It is only relevant when a resource is in disconnected mode.

In the quick-sync bitmap, one bit represents a 4-KiB chunk of on-disk data. If the bit is cleared, it means that the corresponding block is still in sync with the peer node. That implies that the block has not been written to since the time of disconnection. Conversely, if the bit is set, it means that the block has been modified and needs to be re-synchronized whenever the connection becomes available again.

As DRBD detects write I/O on a disconnected device, and hence starts setting bits in the quick-sync bitmap, it does so in RAM — thus avoiding expensive synchronous metadata I/O operations. Only when the corresponding blocks turn cold (that is, expire from the Activity Log), DRBD makes the appropriate modifications in an on-disk representation of the quick-sync bitmap. Likewise, if the resource happens to be manually shut down on the remaining node while disconnected, DRBD flushes the *complete* quick-sync bitmap out to persistent storage.

When the peer node recovers or the connection is re-established, DRBD combines the bitmap information from both nodes to determine the *total data set* that it must re-synchronize. Simultaneously, DRBD examines the generation identifiers to determine the *direction* of synchronization.

The node acting as the synchronization source then transmits the agreed-upon blocks to the peer node, clearing sync bits in the bitmap as the synchronization target acknowledges the modifications. If the re-synchronization is now interrupted (by another network outage, for example) and subsequently resumed it will continue where it left off — with any additional blocks modified in the meantime being added to the re-synchronization data set, of course.

Re-synchronization may be also be paused and resumed manually with the `drbdadm pause-sync` and `drbdadm resume-sync` commands. You should, however, not do so light-heartedly — interrupting re-synchronization leaves your secondary node's disk *Inconsistent* longer than necessary.

23.5. The Peer-Fencing interface

DRBD has an interface defined for fencing [17: For a discussion about Fencing and STONITH, please see the corresponding Pacemaker page http://clusterlabs.org/doc/crm_fencing.html.] the peer node in case of the replication link being interrupted. The `drbd-peer-outdater` helper, bundled with Heartbeat, is the reference implementation for this interface. However, you may easily implement your own peer fencing helper program.

The fencing helper is invoked only in case

1. a `fence-peer` handler has been defined in the resource's (or `common`) `handlers` section, *and*
2. the `fencing` option for the resource is set to either `resource-only` or `resource-and-stonith`, *and*
3. the replication link is interrupted long enough for DRBD [18: That means eg. a TCP timeout, the `ping-timeout`, or the kernel triggers a connection abort eg. because the network link goes down.] to detect a network failure.

The program or script specified as the `fence-peer` handler, when it is invoked, has the DRBD_RESOURCE and DRBD_PEER environment variables available. They contain the name of the affected DRBD resource and the peer's hostname, respectively.

Any peer fencing helper program (or script) must return one of the following exit codes:

Table 1. `fence-peer` *handler exit codes*

Exit code	Implication
3	Peer's disk state was already *Inconsistent*.
4	Peer's disk state was successfully set to *Outdated* (or was *Outdated* to begin with).
5	Connection to the peer node failed, peer could not be reached.
6	Peer refused to be outdated because the affected resource was in the primary role.
7	Peer node was successfully fenced off the cluster. This should never occur unless `fencing` is set to `resource-and-stonith` for the affected resource.

Chapter 24. More Information about DRBD Manage

Here you will find some more information about DRBD Manage internals, techniques, and strategies.

24.1. Free Space reporting

DRBD Manage can report free space in two ways:

- Per node, via `drbdmanage nodes`; this will tell the "physical" space available, which might not mean much if using Thin LVs for storing data.
- Via `drbdmanage list-free-space`; this will return the size for the single largest volume that can be created with the defined replication count.

  So, with 10 storage nodes each having 1TiB free space, the value returned will be 1TiB, and allocating such a volume will not change the free space value.

  For storage nodes with 20GiB, 15GiB, 10GiB, and 5GiB, the free space for 3-way-redundancy will be 10GiB, and 15GiB for 2-way.

The free space issue is further muddled quite a bit by thin LVM pools (one or multiple, depending on storage backend in DRBD Manage - please see Configuring storage plugins for more details), and DRBD Manage snapshots.

24.2. Policy plugin, waiting for deployment

When deploying a resource (resp. volume), a snapshot, or when resizing a volume, you might need to trade between performance and availability.

For a small example, if you have 3 storage servers, but one is down because of a hardware or software upgrade, and you choose to create a new resource that should keep three copies - what should happen?

- Should volume creation block until the third node is available again (hours, days, weeks)?
- Should volume creation just continue? Should it continue even if only one storage server is available, even if 3-way redundancy was requested?
- Should snapshot creation wait, or continue if at least one server was able to create the snapshot?

The more server you have, the more likely it is that at least one of them is non-operating at any given time; so you need to decide on your policies, and configure them.

To avoid having to make such decisions in each driver accessing DRBD Manage [19: Like OpenAttic, OpenNebula, Openstack, ProxMox, etc.] individually (duplicating quite a lot of code), we provide one DRBD Manage plugin [20: "Plugin" might not be the best term here; the code is in the standard DRBD Manage distribution, but it gets called via the `run_external_plugin` API.] that aims to provide these functionality.

The exact configuration depends on the specific service using DRBD Manage, of course; but the basic approach is to allow the admin to specify the waiting behaviour via a JSON blob, which might be given in the configuration via simple text string (see eg. Policy configuration for OpenStack).

The available policies for `WaitForResource` and `WaitForSnapshot` are:

- `count` will wait for an absolute number of deployments; note that these don't have to be in sync [21: On thick LVM a 1TiB deployment would take some time for the initial sync, and you most probably don't plan to wait for that, right?] but just **deployed**.
- `ratio` is similar, but uses a value from `0.0` to `1.0`, which gets multiplied with the number of *planned* deployments. So, for a resource that has (diskful) assignments on 5 servers, and 3 of them are available, a ratio of `0.6` would work.

  NOTE

The given *ratio* will be compared using "greater-or-equal"; so passing `0.5` in does **not** mean "wait for majority", as one-of-two deployments available would already be sufficient! Use `0.51` to get the intended meaning.

If you pass in more than one policy, *any* of them is sufficient; so, for `count=3` and `ratio=0.75`, 3-out-of-5 servers would return `result=true`.

24.2.1. More detailed information for driver-writers

For people implementing software that accesses DBRDmanage via the DBus API, here's an example usage (in mostly Python syntax):

```
result, data = run_external_plugin(
                    "drbdmanage.plugins.plugins.wait_for.WaitForResource",
                    { "resource": "foo",
                      "starttime": <unix-timestamp, ie. time(NULL), of first call>
                      "count": "3",
                      "timeout": "15"})
```

would return the normal array of success/error data [22: Which will always mean some calling/internal error; "Policy not satisfied" or "timeout" will be reported *on this level* as "completed succesfully"!], and an additional dict with detailed status data:

```
{
    "policy": "count",
    "result": "true",
    "timeout": "false"
}
```

If at least 3 servers had the given resource deployed (at the time of the call), `result` got `"true"` because the `count` policy was satisfied.

You can wait with/for these plugins/events:

- `drbdmanage.plugins.plugins.wait_for.WaitForResource` needs a `resource` key as input data;
- `drbdmanage.plugins.plugins.wait_for.WaitForSnapshot` needs a `resource` and a `snapshot` key as input data;
- `drbdmanage.plugins.plugins.wait_for.WaitForVolumeSize` requires `resource`, `volnr`, and `req_size` (net size, in KiB) input arguments.

Please note that because of DBus timeouts this API function does **not** block; it will return immediately. That's the reason for the `starttime` and `timeout` values in the input dictionary; if the plugin sees that the current time is after `starttime` plus `timeout`, it will set the `timeout` result to `"true"`, so that the driver knows to abort and return an error.

Chapter 25. Getting more information

25.1. Commercial DRBD support

Commercial DRBD support, consultancy, and training services are available from the project's sponsor company, LINBIT.

25.2. Public mailing list

The public mailing list for general usage questions regarding DRBD is drbd-user@lists.linbit.com. This is a subscribers-only mailing list, you may subscribe at http://lists.linbit.com/drbd-user/. A complete list archive is available at http://lists.linbit.com/pipermail/drbd-user/.

25.3. Public IRC Channels

Some of the DRBD developers can occasionally be found on the `irc.freenode.net` public IRC server, particularly in the following channels:

- `#drbd`, and
- `#clusterlabs`.

Getting in touch on IRC is a good way of discussing suggestions for improvements in DRBD, and having developer level discussions.

When asking about problems, please use some public paste service to provide the DRBD configuration, logfiles, `drbdsetup status --verbose --statistics` output, and `/proc/drbd` contents. Without that data it's hard to help.

25.4. Official Twitter account

LINBIT maintains an official twitter account.

If you tweet about DRBD, please include the `#drbd` hashtag.

25.5. Publications

DRBD's authors have written and published a number of papers on DRBD in general, or a specific aspect of DRBD. Here is a short selection:

- Philipp Reisner. 'DRBD 9 - What's New'. 2012. Available at http://www.drbd.org/fileadmin/drbd/publications/whats-new-drbd-9.pdf // FIXME only bad version at https://www.netways.de/fileadmin/images/Events_Trainings/Events/OSDC/2013/Slides_2013/Philipp_Reisner_Neues_in_DRBD9.pdf // FIXME - put fixed version there
- Lars Ellenberg. 'DRBD v8.0.x and beyond'. 2007. Available at http://drbd.linbit.com/fileadmin/drbd/publications/drbd8.linux-conf.eu.2007.pdf
- Philipp Reisner. 'DRBD v8 - Replicated Storage with Shared Disk Semantics'. 2007. Available at http://drbd.linbit.com/fileadmin/drbd/publications/drbd8_orig.pdf FIXME sagt 2005?
- Philipp Reisner. 'Rapid resynchronization for replicated storage'. 2006. Available at http://drbd.linbit.com/fileadmin/drbd/publications/drbd-activity-logging_v6.pdf.

You can find many more on http://drbd.linbit.com/home/publications/.

25.6. Other useful resources

- Wikipedia keeps an entry on DRBD.
- Both the Linux-HA wiki and
- ClusterLabs website have some useful information about utilizing DRBD in High Availability clusters.

Appendices

Appendix A: Recent changes

This appendix is for users who upgrade from earlier DRBD versions to DRBD 9.0. It highlights some important changes to DRBD's configuration and behavior.

A.1. Connections

With DRBD 9 data can be replicated across more than two nodes.

This also means that stacking DRBD volumes is now deprecated (though still possible); and that using DRBD as a network-blockdevice (a *DRBD client*) now makes sense.

Associated with this change are

- Metadata size changes (one bitmap per peer)
- `/proc/drbd` now only gives minimal information, see `drbdadm status`
- Resynchronization from/to multiple peers
- The activity log is used even when in Secondary role

A.2. Auto-Promote Feature

DRBD 9 can be configured to do the *Primary*/*Secondary* role switch automatically, on demand.

This feature replaces both the `become-primary-on` configuration value, as well as the old Heartbeat v1 `drbddisk` script.

Please see Automatic Promotion of Resources for more details.

A.3. Increased Performance

DRBD 9 has seen noticeable performance improvements, depending on your specific hardware it's up to two magnitudes faster (measuring number of I/O operations/second for random writes).

A.4. Multiple Volumes in one Resource

Volumes are a new concept in DRBD 8.4. Prior to 8.4, every resource had only one block device associated with it, thus there was a one-to-one relationship between DRBD devices and resources. Since 8.4, multiple volumes (each corresponding to one block device) may share a single replication connection, which in turn corresponds to a single resource.

This results in a few other necessary changes in behaviour:

A.4.1. Changes to udev symlinks

The DRBD udev integration scripts manage symlinks pointing to individual block device nodes. These exist in the `/dev/drbd/by-res/` and `/dev/drbd/by-disk/` directories.

Since DRBD 8.4, a single resource may correspond to multiple volumes, `/dev/drbd/by-res/<resource>` is a *directory*, containing symlinks pointing to individual volumes:

Listing 14. udev managed DRBD symlinks in DRBD 8.4

```
lrwxrwxrwx 1 root root 11 2015-08-09 19:32 /dev/drbd/by-res/home/0 -> ../../drbd0
lrwxrwxrwx 1 root root 11 2015-08-09 19:32 /dev/drbd/by-res/data/0 -> ../../drbd1
lrwxrwxrwx 1 root root 11 2015-08-09 19:33 /dev/drbd/by-res/nfs-root/0 -> ../../drbd2
lrwxrwxrwx 1 root root 11 2015-08-09 19:33 /dev/drbd/by-res/nfs-root/1 -> ../../drbd3
```

Configurations where filesystems are referred to by such a symlink must be updated when moving to DRBD 8.4, usually by simply appending `/0` to the symlink path. If the filesystem was referenced via `UUID=` or `/dev/drbdX` in `/etc/fstab`, no change is necessary.

A.5. Changes to the configuration syntax

This section highlights changes to the configuration syntax. It affects the DRBD configuration files in `/etc/drbd.d`, and `/etc/drbd.conf`.

The `drbdadm` parser still accepts pre-8.4 configuration syntax and automatically translates, internally, into the current syntax. Unless you are planning to use new features from DRBD 9, there is no requirement to modify your configuration to the current syntax. It is, however, recommended that you eventually adopt the new syntax, as the old format will no longer be supported in DRBD 9.

A.5.1. Boolean configuration options

`drbd.conf` supports a variety of boolean configuration options. In pre DRBD 8.4 syntax, these boolean options would be set as follows:

Listing 15. Pre-DRBD 8.4 configuration example with boolean options

```
resource test {
    disk {
        no-md-flushes;
    }
}
```

This led to configuration issues if you wanted to set a boolean variable in the `common` configuration section, and then override it for individual resources:

Listing 16. Pre-DRBD 8.4 configuration example with boolean options in `common` section

```
common {
    disk {
        no-md-flushes;
    }
}
resource test {
    disk {
        # No facility to enable disk flushes previously disabled in
        # "common"
    }
}
```

In DRBD 8.4, all boolean options take a value of `yes` or `no`, making them easily configurable both from `common` and from individual `resource` sections:

Listing 17. DRBD 8.4 configuration example with boolean options in `common` *section*

```
common {
  md-flushes no;
}
resource test {
  disk {
    md-flushes yes;
  }
}
```

A.5.2. `syncer` section no longer exists

Prior to DRBD 8.4, the configuration syntax allowed for a `syncer` section which has become obsolete in 8.4. All previously existing `syncer` options have now moved into the `net` or `disk` sections of resources.

Listing 18. Pre-DRBD 8.4 configuration example with `syncer` *section*

```
resource test {
  syncer {
    al-extents 3389;
    verify-alg md5;
  }
  ...
}
```

The above example is expressed, in DRBD 8.4 syntax, as follows:

Listing 19. DRBD 8.4 configuration example with `syncer` *section replaced*

```
resource test {
  disk {
    al-extents 3389;
  }
  net {
    verify-alg md5;
  }
  ...
}
```

A.5.3. `protocol` option is no longer special

In prior DRBD releases, the `protocol` option was awkwardly (and counter-intuitively) required to be specified on its own, rather than as part of the `net` section. DRBD 8.4 removes this anomaly:

Listing 20. Pre-DRBD 8.4 configuration example with standalone `protocol` *option*

```
resource test {
  protocol C;
  ...
  net {
    ...
  }
  ...
}
```

The equivalent DRBD 8.4 configuration syntax is:

Listing 21. DRBD 8.4 configuration example with `protocol` *option within* `net` *section*

```
resource test {
  net {
    protocol C;
    ...
  }
  ...
}
```

A.5.4. New per-resource `options` section

DRBD 8.4 introduces a new `options` section that may be specified either in a `resource` or in the `common` section. The `cpu-mask` option has moved into this section from the `syncer` section in which it was awkwardly configured before. The `on-no-data-accessible` option has also moved to this section, rather than being in `disk` where it had been in pre-8.4 releases.

Listing 22. Pre-DRBD 8.4 configuration example with `cpu-mask` *and* `on-no-data-accessible`

```
resource test {
  syncer {
    cpu-mask ff;
  }
  disk {
    on-no-data-accessible suspend-io;
  }
  ...
}
```

The equivalent DRBD 8.4 configuration syntax is:

Listing 23. DRBD 8.4 configuration example with `options` *section*

```
resource test {
  options {
    cpu-mask ff;
    on-no-data-accessible suspend-io;
  }
  ...
}
```

A.6. On-line changes to network communications

A.6.1. Changing the replication protocol

Prior to DRBD 8.4, changes to the replication protocol were impossible while the resource was on-line and active. You would have to change the `protocol` option in your resource configuration file, then issue `drbdadm disconnect` and finally `drbdadm connect` on both nodes.

In DRBD 8.4, the replication protocol can be changed on the fly. You may, for example, temporarily switch a connection to asynchronous replication from its normal, synchronous replication mode.

Listing 24. Changing replication protocol while connection is established

```
drbdadm net-options --protocol=A <resource>
```

A.6.2. Changing from single-Primary to dual-Primary replication

Prior to DRBD 8.4, it was impossible to switch between single-Primary to dual-Primary or back while the resource was on-line and active. You would have to change the `allow-two-primaries` option in your resource configuration file, then issue `drbdadm disconnect` and finally `drbdadm connect` on both nodes.

In DRBD 8.4, it is possible to switch modes on-line.

It is *required* for an application using DRBD dual-Primary mode to use a clustered file system or some other distributed locking mechanism. This applies regardless of whether dual-Primary mode is enabled on a temporary or permanent basis.

Refer to Temporary dual-primary mode for switching to dual-Primary mode while the resource is on-line.

A.7. Changes to the `drbdadm` command

A.7.1. Changes to pass-through options

Prior to DRBD 8.4, if you wanted `drbdadm` to pass special options through to `drbdsetup`, you had to use the arcane `-- --<option>` syntax, as in the following example:

Listing 25. Pre-DRBD 8.4 `drbdadm` pass-through options

```
drbdadm -- --discard-my-data connect <resource>
```

Instead, `drbdadm` now accepts those pass-through options as normal options:

Listing 26. DRBD 8.4 `drbdadm` pass-through options

```
drbdadm connect --discard-my-data <resource>
```

The old syntax is still supported, but its use is strongly discouraged. However, if you choose to use the new, more straightforward syntax, you must specify the option (`--discard-my-data`) *after* the subcommand (`connect`) and *before* the resource identifier.

A.7.2. `--force` option replaces `--overwrite-data-of-peer`

The `--overwrite-data-of-peer` option is no longer present in DRBD 8.4. It has been replaced by the simpler `--force`. Thus, to kick off an initial resource synchronization, you no longer use the following command:

Listing 27. Pre-DRBD 8.4 initial sync `drbdadm` commands

```
drbdadm -- --overwrite-data-of-peer primary <resource>
```

Use the command below instead:

Listing 28. DRBD 8.4 initial sync `drbdadm` commands

```
drbdadm primary --force <resource>
```

A.8. Changed default values

In DRBD 8.4, several `drbd.conf` default values have been updated to match improvements in the Linux kernel and available server hardware.

A.8.1. Number of concurrently active Activity Log extents (`al-extents`)

`al-extents`' previous default of 127 has changed to 1237, allowing for better performance by reducing the amount of metadata disk write operations. The associated extended resynchronization time after a primary node crash, which this change introduces, is marginal given the ubiquity of Gigabit Ethernet and higher-bandwidth replication links.

A.8.2. Run-length encoding (`use-rle`)

Run-length encoding (RLE) for bitmap transfers is enabled by default in DRBD 8.4; the default for the `use-rle` option is `yes`. RLE greatly reduces the amount of data transferred during the quick-sync bitmap exchange (which occurs any time two disconnected nodes reconnect).

A.8.3. I/O error handling strategy (`on-io-error`)

DRBD 8.4 defaults to masking I/O errors, which replaces the earlier behavior of passing them on to upper layers in the I/O stack. This means that a DRBD volume operating on a faulty drive automatically switches to the *Diskless* disk state and continues to serve data from its peer node.

A.8.4. Variable-rate synchronization

Variable-rate synchronization is on by default in DRBD 8.4. The default settings are equivalent to the following configuration options:

Listing 29. DRBD 8.4 default options for variable-rate synchronization

```
resource test {
  disk {
    c-plan-ahead 20;
    c-fill-target 50k;
    c-min-rate 250k;
  }
  ...
```

A.8.5. Number of configurable DRBD devices (`minor-count`)

The maximum number of configurable DRBD devices (previously 255) is 1,048,576 (2^{20}) in DRBD 8.4. This is more of a theoretical limit that is unlikely to be reached in production systems.

Chapter 26. DRBD Manage

DRBD Manage will reach its EoL end of 2018 and will be replaced by LINSTOR.

DRBD Manage is an abstraction layer which takes over management of logical volumes (LVM) and management of configuration files for DRBD. Features of DRBD Manage include creating, resizing, and removing of replicated volumes. Additionally, DRBD Manage handles taking snapshots and creating volumes in consistency groups.

This chapter outlines typical administrative tasks encountered during day-to-day operations. It does not cover troubleshooting tasks, these are covered in detail in Troubleshooting and error recovery. If you plan to use 'LVM' as storage plugin, please see read section Configuring LVM **now**, and then return to this point.

26.1. Migrating resources from DRBDManage to LINSTOR

The LINSTOR client contains a sub-command that can generate a migration script that adds existing DRBDManage nodes and resources to a LINSTOR cluster. Migration can be done without downtime. If you do not plan to migrate existing resources, continue with the next section.

The first thing to check is if the DRBDManage cluster is in a healthy state. If the output of `drbdmanage assignments` looks good, you can export the existing cluster database via `drbdmanage export-ctrlvol > ctrlvol.json`. You can then use that as input for the LINSTOR client. The client does **not** immediately migrate your resources, it just generates a shell script. Therefore, you can run the migration assistant multiple times and review/modify the generated shell script before actually executing it. Migration script generation is started via `linstor dm-migrate ctrlvol.json dmmmigrate.sh`. The script will ask a few questions and then generate the shell script. After carefully reading the script, you can then shutdown DRBDManage, and rename the following files. If you do not rename them, the lower-level drbd-utils pick up both kinds of resource files, the ones from DRBDManage and the ones from LINSTOR.

Obviously, you need the linstor-controller service started on one node and the linstor-satellite service on all nodes.

```
# drbdmanage shutdown -qc # on all nodes
# mv /etc/drbd.d/drbdctrl.res{,.dis} # on all nodes
# mv /etc/drbd.d/drbdmanage-resources.res{,.dis} # on all nodes
# bash dmmigrate.sh
```

26.2. Initializing your cluster

We assume that the following steps are accomplished on **all** cluster nodes:

1. The DRBD9 kernel module is installed and loaded
2. `drbd-utils` are installed
3. LVM tools are installed
4. `drbdmanage` and its dependencies are installed

Note that drbdmanage uses dbus-activation to start its server component when necessary, do not start the server manually.

The first step is to review the configuration file of `drbdmanage` (`/etc/drbdmanaged.cfg`) and to create an *LVM volume* group with the name specified in the configuration. In the following we use the default name, which is `drbdpool`, and assume that the volume group consists of `/dev/sda6` and `/dev/sda7`. Creating the volume group is a step that has to be executed on **every** cluster node:

```
# vgcreate drbdpool /dev/sda6 /dev/sda7
```

The second step is to initialize the so called control volume, which is then used to redundantly store your cluster configuration. If the node has multiple interfaces, you have to specify the IP address of the network interface that DRBD should use to communicate with other nodes in the cluster, otherwise the IP is optional. This step must only be done on exactly one cluster node.

```
# drbdmanage init 10.43.70.2
```

We recommend using 'drbdpool' as the name of your LVM volume group as it is the default value and makes your administration life easier. If, for whatever reason, you decide to use a different name, make sure that the option **drbdctrl-vg** is set accordingly in `/etc/drbdmanaged.cfg`. Configuration will be discussed in Cluster configuration.

26.3. Adding nodes to your cluster

Adding nodes to your cluster is easy and requires a single command with two parameters:

1. A node name which **must** match the output of `uname -n`
2. The IP address of the node.

Note

If DNS is configured properly, the tab-completion of `drbdmanage` is able to complete the IP of the given node name.

```
# drbdmanage add-node bravo 10.43.70.3
```

Here we assume that the command was executed on node 'alpha'. If the 'root' user is allowed to execute commands as 'root' on 'bravo' via `ssh`, then the node 'bravo' will automatically join your cluster.

If `ssh` access with public-key authentication is not possible, `drbdmanage` will print a join command that has to be executed on node 'bravo'. You can always query `drbdmanage` to output the join command for a specific node:

```
# drbdmanage howto-join bravo
# drbdmanage join -p 6999 10.43.70.3 1 alpha 10.43.70.2 0 cOQutgNMrinBXb09A3io
```

26.3.1. Types of DRBD Manage nodes

There are quite a few different types of DRBD Manage nodes; please see the diagram below.

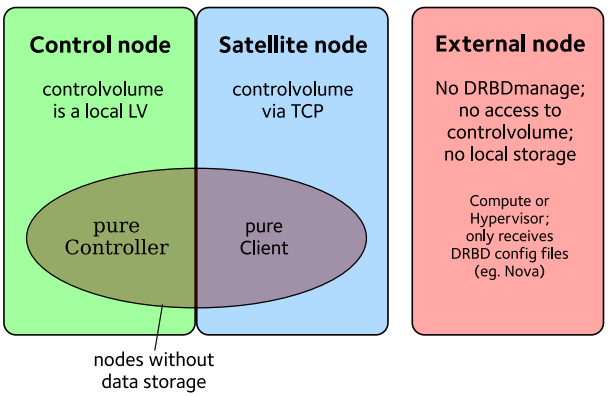


Figure 23. DRBD Manage node types

The rational behind the different types of nodes is as follows: Currently, a DRBD9/DRBD Manage cluster is limited to ~30 nodes. This is the current DRBD9 limit of nodes for a replicated resource. As DRBD Manage uses a DRBD volume

itself (i.e., the control volume) to distribute the cluster information, DRBD Manage was also limited by the maximum number of DRBD9 nodes per resource.

The satellites concept relaxes that limit by splitting the cluster into:

- Control nodes: Nodes having direct access to the control volume.
- Satellite nodes: Nodes that are not directly connected to to the control volume, but are able to receive the content of the control volume via a control node.

In a cluster there is one special node, which we call the "leader". The leader is selected from the set of control nodes and it is the only node that writes data to the control volume (i.e., it has the control volume in DRBD Primary role). All the other control nodes in the cluster automatically switch their role to a "satellite node" and receive their cluster information via TCP/IP, like ordinary satellite nodes. If the current leader node fails, the cluster automatically selects a new leader node among the control nodes.

Control nodes have:

- Direct access to the control volume
- One of them is in the leader role, the rest act like satellite nodes.
- Local storage if it is a normal control node
- No local storage if it is a pure controller

Satellite nodes have:

- No direct access to the control volume, they receive a copy of the cluster configuration via TCP/IP.
- Local storage if it is a normal satellite node
- No local storage if it is a pure client

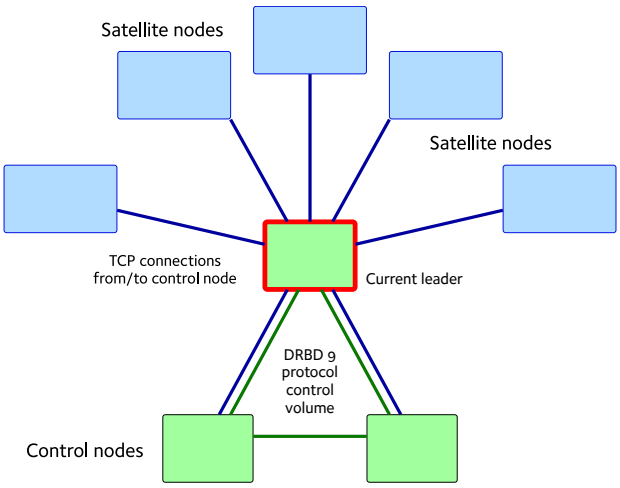


Figure 24. Cluster consisting of satellite nodes

External nodes:

- Have no access to the control volume at all (no dedicated TCP/IP connection to a control node) and no local storage
- Gets its configuration via a different channel (e.g., DRBD configuration via `scp`)
- These are not the droids you are looking for, if you are not sure if you want to use that type of nodes.

26.3.2. Adding a control node

```
# drbdmanage add-node bravo 10.43.70.3
```

26.3.3. Adding a pure controller node

```
# drbdmanage add-node --no-storage bravo 10.43.70.3
```

26.3.4. Adding a satellite node

Here we assume that the node charlie was not added to the cluster so far. The following command adds charlie as a satellite node.

```
# drbdmanage add-node --satellite charlie 10.43.70.4
```

26.3.5. Adding a pure client node

```
# drbdmanage add-node --satellite --no-storage charlie 10.43.70.4
```

26.3.6. Adding an external node

```
# drbdmanage add-node --external delta 10.43.70.5
```

26.4. Cluster configuration

Drbdmanage knows many configuration settings like the log-level or the storage plugin that should be used (i.e., LVM, ThinLV, ThinPool, ZPool, or ThinZpool). Executing `drbdmanage modify-config` starts an editor that is used to specify theses settings. The configuration is split in several sections. If an option is specified in the `[GLOBAL]` section, this setting is used in the entire cluster. Additionally, it is possible to specify settings per node and per site. Node sections follow a syntax of `[Node:nodename]`. If an option is set globally and per node, the node setting overrules the global setting.

It is also possible to group nodes into **sites**. In order to make node 'alpha' part of site 'mysite', you have to specify the 'site' option in alpha's node section:

```
# drbdmanage modify-config
[Node:alpha]
site = mysite
```

It is then also possible to specify drbdmanage settings per site using `[Site:]` sections. Lets assume that you want to set the 'loglevel' option in general to 'INFO', for site 'mysite' to 'WARN' and for node alpha, which is also part of site 'mysite' to DEBUG. This would result in the following configuration:

```
# drbdmanage modify-config
[GLOBAL]
loglevel = INFO

[Site:mysite]
loglevel = WARN

[Node:alpha]
site = mysite
loglevel = DEBUG
```

By executing `drbdmanage modify-config` without any options, you can edit global, per site and per node settings. It is also possible to execute 'modify-config' for a specific node. In this per-node view, it is possible to set further per-node specific settings like the storage plugin discussed in Configuring storage plugins.

26.5. Configuring storage plugins

Storage plugins are **per node** settings that are set with the help of the 'modify-config' sub command.

Lets assume you want to use the 'ThinLV' plugin for node 'bravo', where you want to set the 'pool-name' option to 'mythinpool':

```
# drbdmanage modify-config --node bravo
[GLOBAL]
loglevel = INFO

[Node:bravo]
storage-plugin = drbdmanage.storage.lvm_thinlv.LvmThinLv

[Plugin:ThinLV]
pool-name = mythinpool
```

26.5.1. Configuring LVM

More recent versions of the 'LVM tools' support detecting of file system signatures. Unfortunately the feature set of `lvcreate` varies a lot between distributions: Some of them support `--wipesignatures`, some support `--yes`, and that in all possible combinations. None of them supports a generic force flag. If `lvcreate` detects an existing file system signature, it prompts for input and therefore halts processing. If you use modern 'LVM tools', set this option in `/etc/lvm/lvm.conf`: `wipe_signatures_when_zeroing_new_lvs = 0`. Drbdmanage itself executes `wipefs` on created block devices.

If you use a version of 'LVM' where resources from snapshots are not activated, which we saw for the 'LvmThinPool' plugin, also set `auto_set_activation_skip = 0` in `/etc/lvm/lvm.conf`.

26.5.2. Configuring ZFS

For ZFS the same configuration steps apply, like setting the 'storage-plugin' for the node that should make use of ZFS volumes. Please note that we don't make use of ZFS as a file system, but of ZFS as a logical volume manager. The admin is then free to create any file system she/he desires on top of the DRBD device backed by a ZFS volume. It is also important to note that if you make use of the ZFS plugin, all DRBD resources are created on ZFS, but in case this node is a control node, it still needs LVM for it's control volume.

In the most common case only the following steps are necessary.

```
# zpool create drbdpool /dev/sdX /dev/sdY
# drbdmanage modify-config --node bravo
[Node:bravo]
storage-plugin = drbdmanage.storage.zvol2.Zvol2
```


Currently it is not supported to switch storage plugins on the fly. The workflow is: Add a new node, modify the configuration for that node, make use of the node. Changing other settings (like the log-level) on the fly is perfectly fine.

26.5.3. Discussion of the storage plugins

DRBD Manage has four supported storage plugins as of this writing:

- Thick LVM (`drbdmanage.storage.lvm.Lvm`);
- Thin LVM with a single thin pool (`drbdmanage.storage.lvm_thinlv.LvmThinLv`)
- Thin LVM with thin pools for each volume (`drbdmanage.storage.lvm_thinpool.LvmThinPool`)
- Thick ZFS (`drbdmanage.storage.zvol2.Zvol2`)
- Thin ZFS (`drbdmanage.storage.zvol2_thinlv.ZvolThinLv2`)

For ZFS also legacy plugins (without the "2") exist. New users, and users that did not uses ZFS snapshots should use/switch to the newer version. An on-the-fly storage plugin switch is supported in this particular case.

Here's a short discussion of the relative advantages and disadvantages of these plugins.

Table 2. DRBD Manage storage plugins, comparison

Topic	`lvm.Lvm`	`lvm_thinlv.LvmThinLv`	`lvm_thinpool.LvmThinPool`
Pools	the VG is the pool	a single Thin pool	one Thin pool for each volume
Free Space reporting	Exact	Free space goes down as per written data and snapshots, needs monitoring	Each pool carves some space out of the VG, but still needs to be monitored if snapshots are used
Allocation	Fully pre-allocated	thinly allocated, needs nearly zero space initially	
Snapshots	— not supported —	Fast, efficient (copy-on-write)	
Stability	Well established, known code, very stable	Some kernel versions have bugs re Thin LVs, destroying data	
Recovery	Easiest - text editor, and/or lvm configuration archives in `/etc/lvm/`, in the worst case dd with offset/length	All data in one pool, might incur running `thin_check` across **everything** (needs CPU, memory, time)	Independent Pools, so not all volumes damaged at the same time, faster `thin_check` (less CPU, memory, time)

26.6. Creating and deploying resources/volumes

In the following scenario we assume that the goal is to create a resource 'backups' with a size of '500 GB' that is replicated among 3 cluster nodes. First we show how to achieve the goal in individual steps, and then show a short-cut how to achieve it in a single step:

First, we create a new resource:

```
# drbdmanage add-resource backups
```

Second, we create a new volume within that resource:

```
# drbdmanage add-volume backups 500GB
```

In case we would not have used 'add-resource' in the first step, `drbdmanage` would have known that the resource did not exist and it would have created it.

The third step is to deploy the resource to 3 cluster nodes:

```
# drbdmanage deploy-resource backups 3
```

In this case `drbdmanage` chooses 3 nodes that fit all requirements best, which is by default the set of nodes with the most free space in the `drbdpool` volume group. We will see how to manually assign resources to specific nodes in a moment.

As deploying a new resource/volume to a set of nodes is a very common task, `drbdmanage` provides the following short-cut:

```
# drbdmanage add-volume backups 500GB --deploy 3
```

Manual deployment can be achieved by **assigning** a resource to specific nodes. For example if you decide to assign the 'backups' resource to 'bravo' and 'charlie', you should execute the following steps:

```
# drbdmanage add-volume backups 500GB
# drbdmanage assign-resource backups bravo
# drbdmanage assign-resource backups charlie
```

26.7. Managing snapshots

In the following we assume that the *ThinLV* plugin is used on all nodes that have deployed resources from which snapshots should be taken. For further information on how to configure the storage plugin, please refer to Cluster configuration.

26.7.1. Creating a snapshot

Here we continue the example presented in the previous sections, namely nodes 'alpha', 'bravo', 'charlie', and 'delta' with a resource 'backups' deployed on the first three nodes. The name of the snapshot will be 'snap_backups', and we want the snapshot to be taken on nodes 'bravo' and 'charlie'.

```
# drbdmanage create-snapshot snap_backups backups bravo charlie
```

26.7.2. Restoring a snapshot

In the following we want to restore the content of the snapshot 'snap_backups' to a new resource named 'res_backup_from_snap'.

```
# drbdmanage restore-snapshot res_backup_from_snap backups snap_backups
```

This will create a new resource with the name 'res_backup_from_snap'. This resource then gets automatically deployed to these nodes where currently the resource 'backups' is deployed.

26.7.3. Removing a snapshot

An existing snapshot can be removed as follows:

```
# drbdmanage remove-snapshot backups snap_backups
```

26.8. Checking the state of your cluster

`Drbdmanage` provides various commands to check the state of your cluster. These commands start with a 'list-' prefix and provide various filtering and sorting options. The '--groupby' option can be used to group and sort the output in multiple dimensions. Additional output can be turned on by using the '--show' option. In the following we show some typical examples:

```
# drbdmanage list-nodes
# drbdmanage list-volumes --groupby Size
# drbdmanage list-volumes --groupby Size --groupby Minor
# drbdmanage list-volumes --groupby Size --show Port
```

26.9. Setting options for resources

Currently, it is possible to set the following `drbdsetup` options:

1. net-options
2. peer-device-options
3. disk-options
4. resource-options

Additionally, it is possible to set DRBD event handler.

As for example *net-options* are allowed in the 'common' section as well as per resource, these commands then provide the according switches.

Setting `max-buffers` for a resource 'backups' looks like this:

```
# drbdmanage net-options --max-buffers 2048 --resource backups
```

Setting this option in the common section looks like this:

```
# drbdmanage net-options --max-buffers 2048 --common
```

Additionally, there is always an '--unset-' option for every option that can be specified. So, unsetting `max-buffers` for a resource 'backups' looks like this:

```
# drbdmanage net-options --unset-max-buffers --resource backups
```

It is possible to visualize currently set options with the 'show-options' subcommand.

Setting *net-options* per site is also supported. Lets assume 'alpha' and 'bravo' should be part of site 'first' and 'charlie' and 'delta' should be part of site 'second'. Further, we want to use DRBD protocol 'C' within the two sites, and protocol 'A' between the sites 'first' and 'second'. This would be set up as follows:

```
# drbdmanage modify-config
[Node:alpha]
site = first

[Node:bravo]
site = first

[Node:charlie]
site = second

[Node:delta]
site = second
```

```
# drbdmanage net-options --protocol C --sites 'first:first'
# drbdmanage net-options --protocol C --sites 'second:second'
# drbdmanage net-options --protocol A --sites 'first:second'
```

The '--sites' parameter follows a 'from:to' syntax, where currently 'from' and 'to' have a symetric semantic. Setting an option from 'first:second' also sets this option from 'second:first'.

DRBD event handler can be set in the 'common' section and per resource:

```
# drbdmanage handlers --common --after-resync-target /path/to/script.sh
```

```
# drbdmanage handlers --common --unset-after-resync-target
```

```
# drbdmanage handlers --resource backups --after-resync-target /path/to/script.sh
```

26.10. Rebalancing data with DRBD Manage

Rebalancing data means moving some assignments around, to make better use of the available resources. We'll discuss the same example as for the manual workflow.

Given is an example policy that data needs to be available on 3 nodes, so you need at least 3 servers for your setup.

Now, as your storage demands grow, you will encounter the need for additional servers. Rather than having to buy 3 more servers at the same time, you can *rebalance* your data across a single additional node.

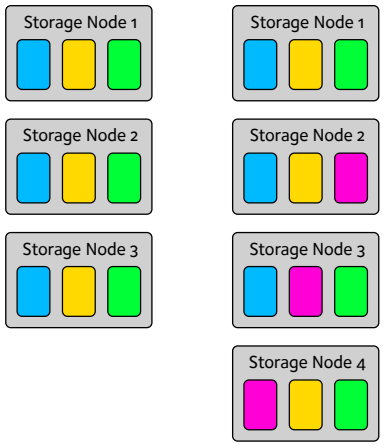


Figure 25. DRBD data rebalancing

First, you need to add the new machine to the cluster; see Adding nodes to your cluster for the commands.

The next step is to add the assignment:

```
# drbdmanage assign <resource> <new-node>
```

Now you need to wait for the (initial) sync to finish; you can eg. use the command `drbdadm status` with (optionally) the resource name.

One of the nodes that *still* has the data will show a status like

```
replication:SyncSource peer-disk:Inconsistent done:5.34
```

while the target node will have a state of *SyncTarget*.

When the target assignment reaches a state of *UpToDate*, you have a full additional copy of your data on this node; now it is safe to remove the assignment from another node:

```
# drbdmanage unassign <resource> <old-node>
```

And voilà - you moved one assignment, in two [23: Or three, if you count waiting for the *UpToDate* state.] easy steps!

26.11. Getting help

The easiest way to get an overview about drbdmanage's subcommands is to read the main man-page (`man drbdmanage`).

A quick way to list available commands on the command line is to type `drbdmanage list`.

Further information on subcommands (e.g., list-nodes) can be retrieved in three ways:

```
# man drbdmanage-list-nodes
# drbdmanage list-nodes -h
# drbdmanage help list-nodes
```

Using the 'help' subcommand is especially helpful when drbdmanage is executed in interactive mode (`drbdmanage interactive`).

One of the most helpful features of drbdmanage is its rich tab-completion, which can be used to complete basically every object drbdmanage knows about (e.g., node names, IP addresses, resource names, ...). In the following we show some possible completions, and their results:

```
# drbdmanage add-node alpha 1<tab> # completes the IP address if hostname can be resolved
# drbdmanage assign-resource b<tab> c<tab> # drbdmanage assign-resource backups charlie
```

If tab-completion does not work out of the box, please try to source the according file:

```
# source /etc/bash_completion.d/drbdmanage # or
# source /usr/share/bash_completion/completions/drbdmanage
```

Chapter 27. DRBD volumes in Openstack

In this chapter you will learn how to use DRBD in Openstack for persistent, replicated, high-performance block storage.

27.1. Openstack Overview

Openstack itself consists of a big range of individual services; the two that are mostly concerned with DRBD are Cinder and Nova. **Cinder** is the block storage service, while **Nova** is the compute node service that's responsible to make the volumes available for the VMs.

DRBD storage volumes can be accessed in two ways: using the iSCSI protocol (for maximum compatibility), and using DRBD client functionality (being submitted to Openstack). For a discussion of these two modes and their differences please see Choosing the Transport Protocol.

27.2. DRBD for Openstack Installation

The *drbdmanage driver* is upstream in Openstack since the Liberty release. It is used (mostly) in the `c-vol` service, so you'll need drbdmanage and DRBD 9 installed on the node(s) running that.

Depending on the specific Openstack variant being used there are a few differences for paths, user names, etc.:

Table 3. Distribution dependent settings

what	rdostack	devstack
Cinder/DRBD Manage driver file location		`/usr/lib/python2.6/site-packages/cinder/volume/drivers/drbdmanagedrv.py`
`/opt/stack/cinder/cinder/volume/drivers/drbdmanagedrv.py`	Cinder configuration file	`/usr/share/cinder/cinder-dist.conf`
`/etc/cinder/cinder.conf`	Admin access data, for sourcing into shell	`/etc/nagios/keystonerc_admin`
`~stack/devstack/accrc/admin/admin`	User used for running `c-vol` service	`cinder`

The generalized installations steps are these:

- In the `cinder.conf` you'll need something like that; the `volume_driver` consists of the class name (last part), and the file path:

```
[DEFAULT]
enabled_backends=drbd-1

[drbd-1]
volume_driver=cinder.volume.drivers.drbdmanagedrv.DrbdManageIscsiDriver
volume_backend_name=DRBD-Managed
drbdmanage_redundancy=1
```

  Please see also Choosing the Transport Protocol for choosing between iSCSI and DRBD transport modes, and other configuration settings.

- Register the backend: (you might need to fetch the authentication environment variables via `source <admin access data>`)

```
# cinder type-create drbd-1
# cinder type-key drbd-1 set volume_backend_name=DRBD-Managed
```

- Allow the user to access the "*org.drbd.drbdmanaged*" service on DBus. For that you need to extend the file `/etc/dbus-1/system.d/org.drbd.drbdmanaged.conf` by an additional stanza like this (replace *USER* by the username as per the table above):

```
<policy user="USER">
  <allow own="org.drbd.drbdmanaged"/>
  <allow send_interface="org.drbd.drbdmanaged"/>
  <allow send_destination="org.drbd.drbdmanaged"/>
</policy>
```

That's it; after a restart of the `c-vol` service you should be able to create your DRBD volumes.

27.2.1. Additional Configuration

The `drbdmanage` backend configuration in `cinder.conf` can contain a few additional settings that modify the exact behaviour.

- `drbdmanage_redundancy = 2` eg. would declare that each volume needs to have 2 storage locations, ie. be replicated once. This means that two times the storage will be used, and that the reported free space looks limited.

  You can request more than two copies of the data; the limit is given by DRBD 9 and the number of storage hosts you have defined.

- `drbdmanage_devs_on_controller = True`: By default each volume will get a DRBD client mapped on the Cinder controller node; apart from being used for iSCSI exports, this might prove helpful for debugging, too.
- In case you need to choose a different iSCSI backend, you can provide an additional configuration to set it, like `iscsi_helper=lioadm`.
- `drbdmanage_resize_policy`, `drbdmanage_resource_policy`, and `drbdmanage_snapshot_policy` configure the behaviour when resizing volumes, resp. creating snapshots or new resources (freshly create or from a snapshot, etc.)

  These are strings that have to be parseable as JSON blobs, for example

  ```
  drbdmanage_snapshot_policy={'count': '1', 'timeout': '60'}
  ```

  See Policy plugin, waiting for deployment for details about the available policies and their configuration items.

  Please be aware that Python's JSON parser is strict – you'll need to use single quotes, for instance, and take other JSON specifications and restrictions into account as well!

  In case you want to call different plugins for this purpose, the `drbdmanage_resize_plugin`, `drbdmanage_resource_plugin`, and `drbdmanage_snapshot_plugin` configuration items exist as well.

- `drbdmanage_net_options` resp. `drbdmanage_resource_options` can be used to set DRBD configuration values on each newly created resource. These already have sane default values; if you want to override, don't forget to add these in again!

Again, these strings get parsed as JSON blobs. The defaults are

```
drbdmanage_net_options = {'connect-int': '4', 'allow-two-primaries': 'yes', 'ko-count':
'30'}
drbdmanage_resource_options = {'auto-promote-timeout': '300'}
```

- `drbdmanage_late_local_assign` and `drbdmanage_late_local_assign_exclude` is a performance optimization for hyperconverged setups; this needs a bit of discussion, so please look at the dedicated chapter Hyperconverged Setups.

These configuration settings can be different from one backend to another.

27.3. Choosing the Transport Protocol

There are two main ways to run DRBD with Cinder:

- accessing storage via iSCSI exports, and
- using the DRBD protocol on the wire.

These are not exclusive; you can define multiple backends, have some of them use iSCSI, and others the DRBD protocol.

27.3.1. iSCSI Transport

The default way to export Cinder volumes is via iSCSI. This brings the advantage of maximum compatibility - iSCSI can be used with every hypervisor, be it VMWare, Xen, HyperV, or KVM.

The drawback is that all data has to be sent to a Cinder node, to be processed by an (userspace) iSCSI daemon; that means that the data needs to pass the kernel/userspace border, and these transitions will cost some performance.

TODO: performance comparision

27.3.2. DRBD Transport

The alternative is to get the data to the VMs by using DRBD as the transport protocol. This means that DRBD 9 [24: The kernel module and userspace, and currently the DRBD Manage daemon too; but please see the note at [s-openstack-drbd-external-NOTE].] needs to be installed on the Nova nodes too, and so restricts them to Linux with KVM at the moment.

One advantage of that solution is that the storage access requests of the VMs can be sent via the DRBD kernel module to the storage nodes, which can then directly access the allocated LVs; this means no Kernel/Userspace transitions on the data path, and consequently better performance. Combined with RDMA capable hardware you should get about the same performance as with VMs accessing a FC backend directly.

Another advantage is that you will be implicitly benefitting from the HA background of DRBD: using multiple storage nodes, possibly available over different network connections, means redundancy and avoiding a single point of failure.

Currently, you'll need to have the hypervisor nodes be part of the DRBD Manage cluster.

When DRBD Manage becomes able to process "*external nodes*", the requirements on the hypervisor nodes will shrink to DRBD 9 kernel module and -userspace only.

27.3.3. Configuring the Transport Protocol

In the storage stanzas in `cinder.conf` you can define the volume driver to use; you can use different drivers for different backend configurations, ie. you can define a 2-way-redundancy iSCSI backend, a 2-way-redundancy DRBD

backend, and a 3-way DRBD backend at the same time. Horizon [25: The Openstack GUI] should offer these storage backends at volume creation time.

The available configuration items for the two drivers are

- for iSCSI:

```
volume_driver=cinder.volume.drivers.drbdmanagedrv.DrbdManageIscsiDriver
```

and

- for DRBD:

```
volume_driver=cinder.volume.drivers.drbdmanagedrv.DrbdManageDrbdDriver
```

The old class name "DrbdManageDriver" is being kept for the time because of compatibility reasons; it's just an alias to the iSCSI driver.

27.4. Some further notes

27.4.1. Free space reporting

The free space that the cinder driver reports is fetched from DRBD Manage, using the defined `drbdmanage_redundancy` setting.

This will return the size for the single largest volume that can be created with this replication count; so, with 10 storage nodes each having 1TiB free space, the value returned for a redundancy count of three will be 1TiB, and allocating such a volume will not change the free space value, as there are three more nodes with that much free space available. For storage nodes with 20GiB, 15GiB, 10GiB, and 5GiB space available, the free space for `drbdmanage_redundancy` being 3 will be 10GiB, and 15GiB for 2.

This issue is further muddled by thin LVM pools (one or multiple, depending on storage backend in DRBD Manage), and snapshots taken from Cinder volumes.

For further information, please see the Openstack Specs about Thin Provisioning - there's the blueprint and the text.

27.4.2. Hyperconverged Setups

The configuration item `drbdmanage_late_local_assign` (available in the DRBD Manage Cinder driver from 1.2.0 on, requiring DRBD Manage 0.98.3 or better) is a performance optimization for hyperconverged setups. With that feature, the driver tries to get a local copy of the data assigned to the hypervisor; that in turn will speed up read IOs, as these won't have to go across the network.

At the time of writing, Nova doesn't pass enough information to Cinder; Cinder isn't told which hypervisor will be used. So the DRBD Manage driver assigns all but one copies at `create_volume` time; the last one is done in the `attach_volume` step, when the hypervisor is known. If this hypervisor is out of space, defined as a storage-less node in DRBD Manage, or otherwise not eligible to receive a copy, any other storage node is used instead, and the target node will receive a *client* assignment only.

Because an image might be copied to the volume before it gets attached to a VM, the "local" assignment can't simply be done on the first access [26: If it assigned on first access, the image copy node (Glance) would receive the copy of the data]. The Cinder driver must be told which nodes are not eligible for local copies; this can be done via `drbdmanage_late_local_assign_exclude`.

For volumes that get cloned from an image stored within Cinder (via a DRBD Manage snapshot), the new resource will

be empty until the `attach_volume` call; at that time the Cinder driver can decide on which nodes the volumes will be deployed, and can actually clone the volume on these.

Free Space Misreported

Late allocation invariably means that the free space numbers are wrong. You might prepare 300 VMs, only to find out that you're running out of disk space when their volumes are in the middle of synchronizing.

But that is a common problem with all thin allocation schemes, so we won't discuss that in more details here.

To summarize:

- You'll need the DRBD Manage Cinder driver 1.2.0 or later, and DRBD Manage 0.98.3 or later.
- The DRBD transport protocol must be used; iSCSI won't offer any locality benefits.
- The `drbdmanage_redundancy` setting must be set to at least two copies.
- To generally enable this feature, set `drbdmanage_late_local_assign` to `True`.
- To specify which hosts should **not** get a local copy, set `drbdmanage_late_local_assign_exclude` to a comma-separated list of hostnames; this should typically include Glance and the Cinder-controller nodes (but not the Cinder-storage nodes!).
- Take care to not run out of disk space.

Here are a few links that show you collected performance data.

- Thirdware Inc. did a Ceph vs. DRBD9 comparison, too; the japanese original can be found in their technical documentation area. A translated (English) version is available on request at sales@linbit.com.
- "*Ceph vs. DRBD9 Performance Comparison*" discusses IOPs, bandwidth, and IO latency; this one needs a free registration on the LINBIT site.

Chapter 28. DRBD Volumes in OpenNebula

This chapter describes DRBD in OpenNebula via the usage of the DRBD Manage storage driver addon.

Detailed installation and configuration instructions and be found in the README.md file of the driver's source.

28.1. OpenNebula Overview

OpenNebula is a flexible and open source cloud management platform which allows its functionality to be extended via the use of addons.

The DRBD Manage addon allows the deployment of virtual machines with highly available images backed by DRBD and attached across the network via the DRBD Transport.

28.2. OpenNebula Installation

Installation of the DRBD Manage storage addon for OpenNebula requires a working OpenNebula cluster as well as a working DRBD Manage cluster.

OpenNebula provides documentation of its design and installation, which should be consulted when considering the creation of a new OpenNebula cluster.

A DRBD cluster with DRBD Manage can be installed and configured by following the instructions in this guide (see Initializing your cluster).

The OpenNebula and DRBD clusters can be somewhat independent of one another with the following exceptions:

- OpenNebula's Front-End and Host nodes must be included in both clusters.
- The Front-End node must have a local copy of the DRBD Manage control volume.

Host nodes do not need a local copy of the DRBD Manage control volume, and virtual machine images are attached to them across the network [27: If a host is also a storage node, it will use a local copy of an image if that is available] (see DRBD Transport). These features simplify the process of adding DRBD to an existing OpenNebula cluster. If this is desired, they may be added to the DRBD Manage cluster using the `--no-storage` and `--satellite` options.

Instructions for installing and configuring the DRBD Manage addon for OpenNebula can be found in the `README.md` file of the addon, which will be rendered by visiting the GitHub page for the driver.

28.3. Deployment Policies

The DRBD Manage addon supports the count and ratio deployment policies (see Policy plugin, waiting for deployment).

By default, the driver considers a deployment successful if one assignment gets available. These policies can be used through the template for the datastore and through the configuration file found under `datastore/drbdmanage.conf`. Policies set in the template will override the ones found in the configuration file.

28.4. Live Migration

Live migration, if enabled, is supported by attaching images to all nodes that do not contain a local copy via DRBD Transport. This makes them available to all nodes in the cluster without consuming additional storage space.

Please note that images that were created before live migration is enabled may need to be assigned to any node they are expected to live migrate to using DRBD Manage directly. This can also be achieved by cold migrating any VM to the Hypervisor that shall later on be available as a live-migration target.

28.5. Free Space Reporting

Free space is calculated differently depending on whether resources are deployed according to `DRBD_REDUNDANCY` or `DRBD_DEPLOYMENT_NODES`.

For datastores using `DRBD_DEPLOYMENT_NODES`, free space is reported based on the most restrictive resources pools from all nodes where resources are being deployed. For example, the capacity of the node with the smallest amount of total storage space is used to determine the total size of the datastore and the node with the least free space is used to determine the remaining space in the datastore.

For a datastore which uses `DRBD_REDUNDANCY`, size and remaining space are determined based on the aggregate storage of the cluster divided by the level of redundancy. It is possible that the free space reported will be larger than the space remaining on any single node. For example, a freshly installed cluster with two storage nodes that both have a capacity of 100GiB and a redundancy level of 1 will report 200GiB of free space, even though no node can store a 200GiB volume. The driver is able to determine this at the time a new image is created and will alert the user.

Chapter 29. DRBD Volumes in Docker

This chapter describes DRBD in Docker via the usage of the DRBD Manage Docker Volume Plugin.

29.1. Docker Overview

Docker is a flexible and open source platform to build, ship, and run distributed applications (a.k.a Linux Containers) for developers and sysadmins.

'drbdmange-docker-volume' is a daemon that is usually socket-activated by 'systemd'. It reads http commands from a local socket and manages DRBD Manage resources that are replicated via DRBD.

29.2. Docker Plugin Installation

There are essentially three different ways to install the DRBD Manage Docker Volume plugin:

- Via LINBIT's PPA
- From Source via `make && sudo make install`
- By building propper packages (`make deb` or `make rpm`).

If you use the PPA or build propper packages, these packages contain a dependency on DRBD Manage in the correct version. If you install from Source, make sure that DRBD Manage itself, and of course Docker, are installed.

The plugin is not enabled by default, to do so, execute the following commands:

```
# systemctl enable docker-drbdmanage-plugin.socket
# systemctl start docker-drbdmanage-plugin.socket
```

Before you can make use of the plugin, make sure you configured a DRBD Manage cluster as described in DRBD Manage.

29.3. Some examples

Here we show a typical example how Docker volumes backed by DRBD are used:

On node alpha:

```
# docker volume create -d drbdmanage --name=dmvol \
                       --opt fs=xfs --opt size=200
# docker run -ti --name=cont \
         -v dmvol:/data --volume-driver=drbdmanage busybox sh
# root@cont: echo "foo" > /data/test.txt
# root@cont: exit
# docker rm cont
```

And then on node bravo:

```
# docker run -ti --name=cont \
         -v dmvol:/data --volume-driver=drbdmanage busybox sh
# root@cont: cat /data/test.txt
  foo
# root@cont: exit
# docker rm cont
# docker volume rm dmvol
```

There is also a BLOG Article on how to use the plugin to deploy a highly-available Wordpress blog.

For further information please read the man page of 'drbdmanage-docker-volume'.

Chapter 30. DRBD Volumes in Proxmox VE

This chapter describes DRBD in Proxmox VE via the DRBD Manage Proxmox Plugin.

30.1. Proxmox VE Overview

Proxmox VE is an easy to use, complete server virtualization environment with KVM, Linux Containers and HA.

'drbdmanage-proxmox' is a Perl plugin for Proxmox that, in combination with DRBD Manage, allows to replicate VM disks on several Proxmox VE nodes. This allows to live-migrate active VMs within a few seconds and with no downtime without needing a central SAN, as the data is already replicated to multiple nodes.

30.2. Proxmox Plugin Installation

LINBIT provides a dedicated public repository for Proxmox VE users. This repository not only contains the Proxmox plugin, but the whole DRBD SDS stack including a customized version of DRBD Manage, DRBD SDS kernel module and user space utilities, and of course the Proxmox Perl plugin itself.

The DRBD9 kernel module gets installed as a `dkms` package (i.e., `drbd-dkms`), therefore you want to install `pve-headers` before you set up/install software from LINBIT's repositories. Following that order ensures that the kernel module is built for your kernel. If you do not follow the latest Proxmox kernel, you have to install kernel headers matching your current kernel (e.g., `pve-headers-$(uname -r)`). If you did not follow that advice and you need to rebuild the dkms package against your current kernel (headers have to be installed), you can issue `apt install --reinstall drbd-dkms`.

LINBIT's repository can be enabled as follows, where "$PVERS" should be set to your Proxmox VE **major version** (e.g., "5", not "5.2"):

```
# wget -O- https://packages.linbit.com/package-signing-pubkey.asc | apt-key add -
# PVERS=5 && echo "deb http://packages.linbit.com/proxmox/ proxmox-$PVERS drbd-9.0" > \
    /etc/apt/sources.list.d/linbit.list
# apt purge drbdmanage
# apt update && apt install drbdmanage-proxmox
```

If you are already (or have been) using Proxmox's version of DRBD Manage, you can simply enable the repository and install `drbdmanage-proxmox`; you don't have to change the existing configuration.

30.3. Proxmox Plugin Configuration

The first step is to set up a static IP address for DRBD traffic. As Proxmox is Debian GNU/Linux based, this is configured via `/etc/network/interfaces`.

The second step is to configure DRBD Manage as described in Initializing your cluster.

The third and last step is to provide a configuration for Proxmox itself. This is done by adding an entry to `/etc/pve/storage.cfg` with the following content, assuming a three node cluster in this example:

```
drbd: drbdstorage
   content images,rootdir
   redundancy 3
```

After that you can create VMs via Proxmox's web GUI by selecting "*drbdstorage*" as storage location.

NOTE: DRBD supports only the **raw** *disk format at the moment.*

At this point you can try to live migrate the VM – as all data is available on both nodes it will take just a few seconds. The overall process might take a bit longer if the VM is under load and if there is a lot of RAM being dirtied all the time. But in any case, the downtime is minimal and you will see no interruption at all.