



The DRBD User's Guide

Table of Contents

Please Read This First	1
Introduction to DRBD	2
1. DRBD Fundamentals	3
1.1. Kernel module	3
1.2. User space administration tools	3
1.3. Resources	4
1.4. Resource roles	5
2. DRBD Features	6
2.1. Single-primary mode	6
2.2. Dual-primary mode	6
2.3. Replication modes	6
2.4. Multiple replication transports	7
2.5. Efficient synchronization	7
2.6. Suspended replication	9
2.7. On-line device verification	9
2.8. Replication traffic integrity checking	10
2.9. Split brain notification and automatic recovery	10
2.10. Support for disk flushes	11
2.11. Disk error handling strategies	12
2.12. Strategies for dealing with outdated data	12
2.13. Three-way replication	13
2.14. Long-distance replication with DRBD Proxy	14
2.15. Truck based replication	14
2.16. Floating peers	15
Building, installing and configuring DRBD	16
3. Installing pre-built DRBD binary packages	17
3.1. Packages supplied by LINBIT	17
3.2. Packages supplied by distribution vendors	17
3.3. Compiling packages from source	18
4. Configuring DRBD	20
4.1. Preparing your lower-level storage	20
4.2. Preparing your network configuration	20
4.3. Configuring your resource	21
4.4. Enabling your resource for the first time	24
4.5. The initial device synchronization	25
4.6. Using truck based replication	26
Working with DRBD	28
5. Common administrative tasks	29

5.1. Checking DRBD status	29
5.2. Enabling and disabling resources	37
5.3. Reconfiguring resources	38
5.4. Promoting and demoting resources	38
5.5. Basic Manual Fail-over	38
5.6. Upgrading DRBD	39
5.7. Downgrading DRBD 8.4 to 8.3	42
5.8. Enabling dual-primary mode	43
5.9. Using on-line device verification	45
5.10. Configuring the rate of synchronization	46
5.11. Configuring checksum-based synchronization	48
5.12. Configuring congestion policies and suspended replication	49
5.13. Configuring I/O error handling strategies	49
5.14. Configuring replication traffic integrity checking	50
5.15. Resizing resources	51
5.16. Disabling backing device flushes	54
5.17. Configuring split brain behavior	55
5.18. Creating a three-node setup	57
5.19. Using DRBD Proxy	59
6. Troubleshooting and error recovery	65
6.1. Dealing with hard drive failure	65
6.2. Dealing with node failure	66
6.3. Manual split brain recovery	68
DRBD-enabled applications	70
7. Integrating DRBD with Pacemaker clusters	71
7.1. Pacemaker primer	71
7.2. Adding a DRBD-backed service to the cluster configuration	71
7.3. Using resource-level fencing in Pacemaker clusters	72
7.4. Using stacked DRBD resources in Pacemaker clusters	75
7.5. Configuring DRBD to replicate between two SAN-backed Pacemaker clusters	81
8. Integrating DRBD with Red Hat Cluster	85
8.1. Red Hat Cluster background information	85
8.2. Red Hat Cluster configuration	86
8.3. Using DRBD in Red Hat Cluster fail-over clusters	86
9. Using LVM with DRBD	89
9.1. LVM primer	89
9.2. Using a Logical Volume as a DRBD backing device	90
9.3. Using automated LVM snapshots during DRBD synchronization	91
9.4. Configuring a DRBD resource as a Physical Volume	92
9.5. Adding a new DRBD volume to an existing Volume Group	93
9.6. Nested LVM configuration with DRBD	94

9.7. Highly available LVM with Pacemaker	97
10. Using GFS2 with DRBD	99
10.1. GFS primer	99
10.2. Creating a DRBD resource suitable for GFS2	99
10.3. Configuring CMAN	100
10.4. Creating a GFS2 filesystem	101
10.5. Using your GFS2 filesystem with Pacemaker	102
11. Using OCFS2 with DRBD	103
11.1. OCFS2 primer	103
11.2. Creating a DRBD resource suitable for OCFS2	104
11.3. Creating an OCFS2 filesystem	104
11.4. Pacemaker OCFS2 management	105
11.5. Legacy OCFS2 management (without Pacemaker)	106
12. Using Xen with DRBD	110
12.1. Xen primer	110
12.2. Setting DRBD module parameters for use with Xen	110
12.3. Creating a DRBD resource suitable to act as a Xen VBD	111
12.4. Using DRBD VBDs	111
12.5. Starting, stopping, and migrating DRBD-backed domU's	112
12.6. Internals of DRBD/Xen integration	113
12.7. Integrating Xen with Pacemaker	113
Optimizing DRBD performance	114
13. Measuring block device performance	115
13.1. Measuring throughput	115
13.2. Measuring latency	115
14. Optimizing DRBD throughput	117
14.1. Hardware considerations	117
14.2. Throughput overhead expectations	117
14.3. Tuning recommendations	118
15. Optimizing DRBD latency	121
15.1. Hardware considerations	121
15.2. Latency overhead expectations	121
15.3. Tuning recommendations	122
Learning more about DRBD	124
16. DRBD Internals	125
16.1. DRBD meta data	125
16.2. Generation Identifiers	127
16.3. The Activity Log	130
16.4. The quick-sync bitmap	132
16.5. The peer fencing interface	132
17. Getting more information	134

17.1. Commercial DRBD support	134
17.2. Public mailing list	134
17.3. Public IRC Channels	134
17.4. Official Twitter account	134
17.5. Publications	134
17.6. Other useful resources	135
Appendices	136
Appendix A: Recent changes	137
A.1. Volumes	137
A.2. Changes to the configuration syntax	138
A.3. On-line changes to network communications	141
A.4. Changes to the <code>drbdadm</code> command	141
A.5. Changed default values	142

Please Read This First

This guide is intended to serve users of the Distributed Replicated Block Device (DRBD) 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 <http://www.drbd.org/users-guide/>.



This guide assumes, throughout, that you are using DRBD version 8.4.0 or later. If you are using a pre-8.4 release of DRBD, please use the version of this guide which has been preserved at <http://www.drbd.org/users-guide-8.3/>.

Please use [the drbd-user mailing list](#) to submit comments.

This guide is organized in seven parts:

- [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, installing and configuring DRBD](#) talks about building DRBD from source, installing pre-built DRBD packages, and contains an overview of getting DRBD running on a cluster system.
- [Working with DRBD](#) is about managing DRBD, configuring and reconfiguring DRBD resources, and 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 about DRBD](#) 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 8.4, 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

The Distributed Replicated Block Device (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 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.

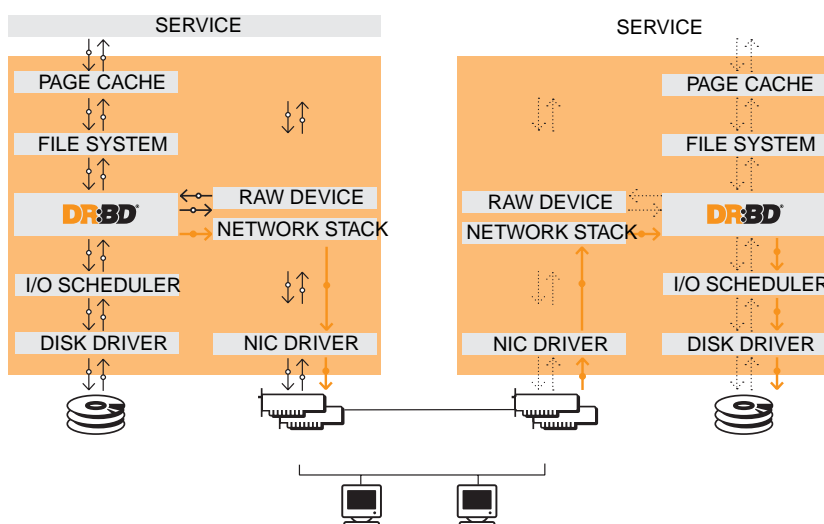


Figure 1. DRBD's position within the Linux I/O stack

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.

drbdadm

The high-level administration tool of the DRBD 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 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 or 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/drbdX`, where `X` is the device minor number. DRBD also allows for user-defined block device names which must, however, start with `drbd_`.



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. As of the time of this writing, each resource involves only two hosts and exactly one connection between these hosts, so for the most part, the terms `resource` and `connection` can be used interchangeably.

At the `drbdadm` level, a connection is addressed by the resource name.

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` or by way of some automated algorithm by a cluster management application. 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. [Common administrative tasks](#) 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 both cluster nodes. 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. 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 with 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 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 both nodes (or 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. Multiple replication transports

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 is available as part of the OFED stack for most current distributions. 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.

2.5. 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.

2.5.1. Variable-rate synchronization

In variable-rate synchronization (the default), 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.

2.5.2. 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.

2.5.3. 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.

2.6. 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.

2.7. 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 (the *verification target*), 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.

2.8. 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. The Linux kernel crypto API provides these; 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 peer requests 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.

2.9. 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 would be modified, 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.

2.10. 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.

2.11. 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 the 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. 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.

2.12. 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 on-going 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.

2.13. Three-way replication



Available in DRBD version 8.3.0 and above

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 with 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:

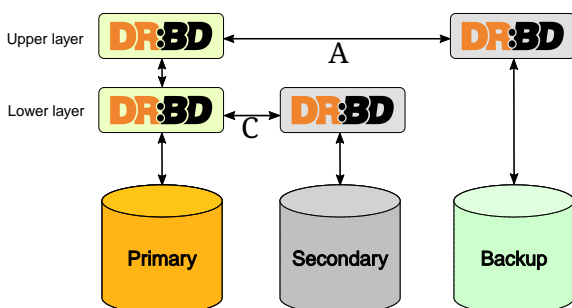


Figure 3. DRBD resource stacking

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.

2.14. Long-distance replication with DRBD Proxy



DRBD Proxy requires DRBD version 8.2.7 or above.

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 well 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 the only part 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.

2.15. 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 days or weeks). Truck based replication allows us to ship a data seed to the remote site, and drastically reduce the initial synchronization time. See [Using truck based replication](#) for details on this use case.

2.16. 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](#).

Building, installing and configuring DRBD

Chapter 3. Installing pre-built DRBD binary packages

3.1. Packages supplied by LINBIT

LINBIT, the DRBD project's sponsor company, provides DRBD binary packages to its commercial support customers. These packages are available at <http://www.linbit.com/support/> and are considered "official" DRBD builds.

These builds are available for the following distributions:

- Red Hat Enterprise Linux (RHEL), versions 5, 6, and 7
- SUSE Linux Enterprise Server (SLES), versions 11SP4, and 12
- Debian GNU/Linux, 8 (jessie), and 9 (stretch)
- Ubuntu Server Edition LTS 16.04 (Xenial Xerus), and LTS 18.04 (Bionic Beaver).

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 `rpm -i` (for new installations) or `rpm -U` (for upgrades), along with the corresponding package names.

For Debian-based systems (Debian GNU/Linux, Ubuntu) systems, `drbd8-utils` and `drbd8-module` packages are installed with `dpkg -i`, or `gdebi` if available.

3.2. Packages supplied by distribution vendors

A number of distributions include 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.

3.2.1. SUSE Linux Enterprise Server

SUSE Linux Enterprise Server (SLES), includes DRBD 0.7 in versions 9 and 10. DRBD 8.3 is included in SLES 11 High Availability Extension (HAE) SP1.

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
```

3.2.2. Debian GNU/Linux

Debian GNU/Linux includes DRBD 8 from the 5.0 release (**lenny**) onwards. In 6.0 (**squeeze**), which is based on a 2.6.32 Linux kernel, Debian ships a backported version of DRBD.

On **squeeze**, since DRBD is already included with the stock kernel, all that is needed to install is the **drbd8-utils** package:

```
apt-get install drbd8-utils
```

On **lenny** (obsolete), you install DRBD by issuing:

```
apt-get install drbd8-utils drbd8-module
```

3.2.3. CentOS

CentOS has had DRBD 8 since release 5.

DRBD can be installed using **yum** (note that you will need the **extras** repository (or EPEL / ELRepo) enabled for this to work):

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

3.2.4. Ubuntu Linux

To install DRBD on Ubuntu, you issue these commands:

```
apt-get update  
apt-get install drbd8-utils
```

On (very) old Ubuntu versions you might need to explicitly install **drbd8-module**, too; in newer versions the default kernel already includes the upstream DRBD version.

3.3. 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.

Chapter 4. Configuring DRBD

4.1. 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 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](#).

4.2. 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 possible to configure a DRBD resource to support more than one TCP connection. 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.

4.3. 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`.

4.3.1. Example configuration

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

previous sections:

Listing 1. Simple DRBD configuration (/etc/drbd.d/global_common.conf)

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

Listing 2. 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.

The configuration above implicitly creates one volume in the resource, numbered zero (`0`). For multiple volumes in one resource, modify the syntax as follows:

Listing 3. 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](#).

4.3.2. 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>.

4.3.3. 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](#).

4.3.4. 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 resource name in the configuration. You may use any arbitrary identifier, however the name must not contain characters other than those found in the US-ASCII character set, and must also not include whitespace.

Every resource configuration must also have 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 both 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;
    }
}
```

4.4. 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>
v08 Magic number not found
Writing meta data...
initialising activity log
NOT initializing bitmap
New drbd meta data block successfully created.
```

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 `/proc/drbd`

DRBD's virtual status file in the `/proc` filesystem, `/proc/drbd`, should now contain information similar to the following:

```
# cat /proc/drbd
version: 8.4.1 (api:1/proto:86-100)
GIT-hash: 91b4c048c1a0e06777b5f65d312b38d47abaea80 build by buildsysteem@linbit, 2011-12-20 12:58:48
0: cs:Connected ro:Secondary/Secondary ds:Inconsistent/Inconsistent C r-----
   ns:0 nr:0 dw:0 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:524236
```



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.

4.5. 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 disk, 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 `/proc/drbd`. 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). You may now 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 [Common administrative tasks](#), which describes common administrative tasks to perform on your resource.

4.6. 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 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>
```

- 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>
```

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.
- 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`.

Working with DRBD

Chapter 5. 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](#).

5.1. Checking DRBD status

5.1.1. Retrieving status with `drbd-overview`

One convenient way to look at DRBD's status is the `drbd-overview` utility.

```
# drbd-overview
0:home                Connected Primary/Secondary
  UpToDate/UpToDate C r--- /home          xfs  200G 158G 43G  79%
1:data                Connected Primary/Secondary
  UpToDate/UpToDate C r--- /mnt/ha1       ext3  9.9G 618M 8.8G  7%
2:nfs-root            Connected Primary/Secondary
  UpToDate/UpToDate C r--- /mnt/netboot  ext3  79G  57G  19G  76%
```

5.1.2. 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
  peer role:Secondary
    replication:Established peer-disk:UpToDate
```

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

You can get more information by passing the `--verbose` and/or `--statistics` arguments to `drbdsetup`:

```
# drbdsetup status home --verbose --statistics
home role:Secondary suspended:no
  write-ordering:flush
  volume:0 minor:0 disk:UpToDate
    size:5033792 read:0 written:0 al-writes:0 bm-writes:0 upper-pending:0
    lower-pending:0 al-suspended:no blocked:no
  peer connection:Connected role:Secondary congested:no
    volume:0 replication:Established peer-disk:UpToDate
      resync-suspended:no
      received:0 sent:0 out-of-sync:0 pending:0 unacked:0
```

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 **role** (see [Resource roles](#)).

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](#) 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:

Replication protocol used by the resource. Either **A**, **B** or **C**. See [Replication modes](#) for details.

5.1.3. One-shot or realtime monitoring via **drbdsetup events2**



This is available only with userspace versions 8.9.3 and kernel module 8.4.6, 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):

Listing 4. 'drbdsetup' example output (lines broken for readability)

```
# 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:

Listing 5. '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.

5.1.4. 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 real-time status information about all DRBD resources currently configured. You may interrogate this file's contents using this command:

```
$ cat /proc/drbd
version: 8.4.0 (api:1/proto:86-100)
GIT-hash: 09b6d528b3b3de50462cd7831c0a3791abc665c3 build by linbit@buildsystem.linbit,
2011-10-12 09:07:35
0: cs:Connected ro:Secondary/Secondary ds:UpToDate/UpToDate C r-----
   ns:0 nr:0 dw:0 dr:656 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
1: cs:Connected ro:Primary/Secondary ds:UpToDate/UpToDate C r---
   ns:0 nr:0 dw:0 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
2: cs:Connected ro:Secondary/Primary ds:UpToDate/UpToDate C r---
   ns:0 nr:0 dw:0 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
```

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

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 from `/proc/drbd` contains various pieces of information about the resource:

`cs` (*connection state*)

Status of the network connection. See [Connection states](#) for details about the various connection states.

`ro` (*roles*)

Roles of the nodes. The role of the local node is displayed first, followed by the role of the partner node shown after the slash. See [Resource roles](#) for details about the possible resource roles.

`ds` (*disk states*)

State of the hard disks. Prior to the slash the state of the local node is displayed, after the slash the state of the hard disk of the partner node is shown. See [Disk states](#) for details about the various disk states.

Replication protocol

Replication protocol used by the resource. Either `A`, `B` or `C`. See [Replication modes](#) for details.

I/O Flags

Six state flags reflecting the I/O status of this resource. See [I/O state flags](#) for a detailed explanation of these flags.

Performance indicators

A number of counters and gauges reflecting the resource's utilization and performance. See [Performance indicators](#) for details.

5.1.5. Connection states

A resource's connection state can be observed either by monitoring `/proc/drbd`, or by issuing the

`drbdadm cstate` command:

```
# drbdadm cstate <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: *WFConnection* and *WFReportParams*.

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*.

WFConnection

This node is waiting until the peer node becomes visible on the network.

WFReportParams

TCP connection has been established, this node waits for the first network packet from the peer.

Connected

A DRBD connection has been established, data mirroring is now active. This is the normal state.

StartingSyncS

Full synchronization, initiated by the administrator, is just starting. The local node will be the source of synchronization. The next possible states are: *SyncSource* or *PausedSyncS*.

StartingSyncT

Full synchronization, initiated by the administrator, is just starting. The local node will be the target of synchronization. Next state: *WFSyncUUID*.

WFBitMapS

Partial synchronization is just starting. The local node will be the source of synchronization. Next possible states: *SyncSource* or *PausedSyncS*.

WFBitMapT

Partial synchronization is just starting. The local node will be the target of synchronization. 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.

5.1.6. Resource roles

A resource's role can be observed either by monitoring `/proc/drbd`, or by issuing the `drbdadm role` command:

```
# drbdadm role <resource>
Primary/Secondary
```

The local resource role is always displayed first, the remote resource role last.

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.

5.1.7. Disk states

A resource's disk state can be observed either by monitoring `/proc/drbd`, or by issuing the `drbdadm dstate` command:

```
# drbdadm dstate <resource>
UpToDate/UpToDate
```

The local disk state is always displayed first, the remote disk state last.

Both the local and the remote 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.

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.

5.1.8. I/O state flags

The I/O state flag field in `/proc/drbd` contains information about the current state of I/O operations associated with the resource. There are six such flags in total, with the following possible values:

1. I/O suspension. Either `r` for *running* or `s` for *suspended* I/O. Normally `r`.
2. Serial resynchronization. When a resource is awaiting resynchronization, but has deferred this because of a `resync-after` dependency, this flag becomes `a`. Normally `-`.
3. Peer-initiated sync suspension. When resource is awaiting resynchronization, but the peer node has suspended it for any reason, this flag becomes `p`. Normally `-`.
4. Locally initiated sync suspension. When resource is awaiting resynchronization, but a user on the local node has suspended it, this flag becomes `u`. Normally `-`.
5. Locally blocked I/O. Normally `-`. May be one of the following flags:
 - `d`: I/O blocked for a reason internal to DRBD, such as a transient disk state.
 - `b`: Backing device I/O is blocking.
 - `n`: Congestion on the network socket.
 - `a`: Simultaneous combination of blocking device I/O and network congestion.
6. Activity Log update suspension. When updates to the Activity Log are suspended, this flag becomes `s`. Normally `-`.

5.1.9. Performance indicators

The second line of `/proc/drbd` information for each resource contains the following counters and gauges:

`ns` (*network send*)

Volume of net data sent to the partner via the network connection; in Kibyte.

`nr` (*network receive*)

Volume of net data received by the partner via the network connection; in Kibyte.

`dw` (*disk write*)

Net data written on local hard disk; in Kibyte.

`dr` (*disk read*)

Net data read from local hard disk; in Kibyte.

al (*activity log*)

Number of updates of the activity log area of the meta data.

bm (*bit map*)

Number of updates of the bitmap area of the meta data.

lo (*local count*)

Number of open requests to the local I/O sub-system issued by DRBD.

pe (*pending*)

Number of requests sent to the partner, but that have not yet been answered by the latter.

ua (*unacknowledged*)

Number of requests received by the partner via the network connection, but that have not yet been answered.

ap (*application pending*)

Number of block I/O requests forwarded to DRBD, but not yet answered by DRBD.

ep (*epochs*)

Number of epoch objects. Usually 1. Might increase under I/O load when using either the **barrier** or the **none** write ordering method.

wo (*write order*)

Currently used write ordering method: **b**(barrier), **f**(flush), **d**(drain) or **n**(none).

oos (*out of sync*)

Amount of storage currently out of sync; in Kibibytes.

5.2. Enabling and disabling resources

5.2.1. 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.

5.2.2. 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.

5.3. 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`.

5.4. Promoting and demoting resources

Manually switching a `resource's role` from secondary to primary (promotion) or vice versa (demotion) is done using 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 `<resource>` is still in the primary role on the peer will result in an error.

A resource configured to allow `dual-primary mode` can be switched to the primary role on both nodes.

5.5. 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>
# 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> <mountpoint>
```

5.6. Upgrading DRBD

Upgrading DRBD is a fairly simple process. This section will cover the process of upgrading from 8.3.x to 8.4.x, however this process should work for all upgrades.

5.6.1. Updating your repository

Due to the number of changes between the 8.3 and 8.4 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-8.4]
name=DRBD 8.4
baseurl=http://packages.linbit.com/<hash>/8.4/rhel6/<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` to reflect the following changes.

```
deb http://packages.linbit.com/<hash>/8.4/debian squeeze main
```



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-get update so Debian recognizes the updated repo.

```
apt-get update
```

5.6.2. Upgrading the packages

Before you begin make sure your resources are in sync. The output of 'cat /proc/drbd' should show UpToDate/UpToDate.

```
bob# cat /proc/drbd

version: 8.3.12 (api:88/proto:86-96)
GIT-hash: e2a8ef4656be026bbae540305fcb998a5991090f build by buildsysteem@linbit, 2011-10-28 10:20:38
0: cs:Connected ro:Secondary/Primary ds:UpToDate/UpToDate C r-----
   ns:0 nr:33300 dw:33300 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
```

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.

Now update your packages with either yum or apt.

```
bob# yum upgrade
```

```
bob# apt-get upgrade
```

Once the upgrade is finished will now have the latest DRBD 8.4 kernel module and drbd-utils on your secondary node, bob. Start DRBD.

- Manually

```
bob# /etc/init.d/drbd start
```

- Pacemaker

```
# crm node online bob
```

The output of 'cat /proc/drbd' on bob should show 8.4.x and look similar to this.

```
version: 8.4.1 (api:1/proto:86-100)
GIT-hash: 91b4c048c1a0e06777b5f65d312b38d47abaea80 build by buildsysteem@linbit, 2011-
12-20 12:58:48
0: cs:Connected ro:Secondary/Primary ds:UpToDate/UpToDate C r-----
   ns:0 nr:12 dw:12 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
```



On the primary node, alice, 'cat /proc/drbd' will still show the prior version, until you upgrade it.

At this point the cluster has two different versions of DRBD. Stop any service using DRBD and then DRBD on the primary node, alice, and promote bob. Again this can be done either manually or via the Pacemaker shell.

- Manually

```
alice # umount /dev/drbd/by-res/r0
alice # /etc/init.d/drbd stop
bob # drbdadm primary r0
bob # mount /dev/drbd/by-res/r0/0 /mnt/drbd
```

Please note that the mount command now references '/0' which defines the volume number of a resource. See [Volumes](#) for more information on the new volumes feature.

- Pacemaker

```
# crm node standby alice
```



This will interrupt running services by stopping them and migrating them to the secondary server, bob.

At this point you can safely upgrade DRBD by using yum or apt.

```
alice# yum upgrade
```

```
alice# apt-get upgrade
```

Once the upgrade is complete you will now have the latest version of DRBD on alice and can start DRBD.

- Manually

```
alice# /etc/init.d/drbd start
```

- Pacemaker

```
alice# crm node online alice
```



Services will still be located on bob and will remain there until you migrate them back.

Both servers should now show the latest version of DRBD in a connected state.

```
version: 8.4.1 (api:1/proto:86-100)
GIT-hash: 91b4c048c1a0e06777b5f65d312b38d47abaea80 build by buildsysteem@linbit, 2011-
12-20 12:58:48
0: cs:Connected ro:Secondary/Primary ds:UpToDate/UpToDate C r-----
   ns:0 nr:12 dw:12 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
```

5.6.3. Migrating your configs

DRBD 8.4 is backward compatible with the 8.3 configs 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.

5.7. Downgrading DRBD 8.4 to 8.3

If you're currently running DRBD 8.4 and would like to revert to 8.3 there are several steps you will have to follow. This section assumes you still have the 8.4 kernel module and 8.4 utilities installed.

Stop any services accessing the DRBD resources, unmount, and demote the devices to Secondary. Then perform the following commands.



These steps will have to be completed on both servers.

```
drbdadm down all
drbdadm apply-al all
rmmod drbd
```

If you're using the LINBIT repositories you can remove the packages using `apt-get remove drbd8-utils drbd8-module-`uname -r`` or `yum remove drbd kmod-drbd`

Now that 8.4 is removed reinstall 8.3. You can do this either by changing your repositories back to the 8.3 repos, or by following the steps located [in the 8.3 User's Guide](#)



If you migrated your configs to the 8.4 format be sure to revert them back to the 8.3 format. See [Changes to the configuration syntax](#) for the options you need to revert.

Once 8.3 is re-installed you can start your DRBD resources either manually using `drbdadm` or `/etc/init.d/drbd start`.

5.8. Enabling dual-primary mode

Dual-primary mode allows a resource to assume the primary role simultaneously on both nodes. 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.

5.8.1. 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;
  }
  disk {
    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.

5.8.2. 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).

5.8.3. Automating promotion on system startup

When a resource is configured to support dual-primary mode, it may also be desirable to automatically switch the resource into the primary role upon system (or DRBD) startup.

```
resource <resource>
  startup {
    become-primary-on both;
  }
  ...
}
```

The `/etc/init.d/drbd` system init script parses this option on startup and promotes resources accordingly.



The `become-primary-on` approach **should be avoided**, we recommend to use a cluster manager if at all possible. See for example [Pacemaker-managed](#) DRBD configurations. In Pacemaker (or other cluster manager) configurations, resource promotion and demotion should always be handled by the cluster manager.

5.9. Using on-line device verification

5.9.1. Enabling on-line verification

[On-line device verification](#) 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 {
    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.

5.9.2. 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 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>
```

5.9.3. 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.

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.conf`), you may also use:

```
42 0 * * 0    root    /sbin/drbdadm verify all
```

5.10. 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.

5.10.1. Variable sync rate configuration

Since DRBD 8.4, the default has switched to variable-rate synchronization. In this mode, DRBD uses an automated control loop algorithm to determine, and permanently 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 200;
        c-max-rate 10M;
        c-fill-target 15M;
    }
}
```



A good starting value for **c-fill-target** is $BDP \times 3$, where BDP is your bandwidth delay product on the replication link.

5.10.2. Permanent fixed sync rate configuration

For testing purposes it might be useful to deactivate the dynamic resync controller, and to configure DRBD to some fixed resynchronization speed. That is only an upper limit, of course - if there is some bottleneck (or just application IO), the desired speed won't be achieved.

The maximum bandwidth a resource uses for background re-synchronization is determined by the **rate** option for a resource. This must be included in the resource configuration'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**.



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 180MB/s, and a Gigabit Ethernet network capable of sustaining 110 MB/s network throughput (the network being the bottleneck), you would calculate:

$$110 \text{ MB/s} * 0.3 = 33 \text{ MB/s}$$

Figure 4. 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:

$$80 \text{ MB/s} * 0.3 = 24 \text{ MB/s}$$

Figure 5. Syncer rate example, 80MB/s effective available bandwidth

In this case, the recommended value for the `rate` option would be `24M`.

5.10.3. Temporary fixed sync rate configuration

It is sometimes desirable to temporarily adjust the sync rate. For example, you might want to speed up background re-synchronization after having performed scheduled maintenance on one of your cluster nodes. Or, you might want to throttle background re-synchronization if it happens to occur at a time when your application is extremely busy with write operations, and you want to make sure that a large portion of the existing bandwidth is available to replication.

For example, in order to make most bandwidth of a Gigabit Ethernet link available to re-synchronization, issue the following command:

```
# drbdadm disk-options --c-plan-ahead=0 --resync-rate=110M <resource>
```

You need to issue this command on the *SyncTarget* node.

To revert this temporary setting and re-enable the synchronization rate set in `/etc/drbd.conf`, issue this command:

```
# drbdadm adjust <resource>
```

5.11. 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.

5.12. 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.

5.13. 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:

1. `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.
2. `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).
3. `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.

5.14. 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.

5.15. Resizing resources

5.15.1. 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 both 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>
```

5.15.2. 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 growing the backing block device:

```
# 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.

5.15.3. 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 -- --size=<new-size> resize <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).

5.15.4. 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>
```

5.16. 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>
```

5.17. Configuring split brain behavior

5.17.1. 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.

5.17.2. Automatic split brain recovery policies



Configuring DRBD to automatically resolve data divergence situations 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, 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 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;
    ...
  }
  ...
}
```

5.18. Creating a three-node setup

A three-node setup involves one DRBD device *stacked* atop another.

5.18.1. 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.

5.18.2. 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 {
  net {
    protocol C;
  }

  on alice {
    device    /dev/drbd0;
    disk      /dev/sda6;
    address   10.0.0.1:7788;
    meta-disk internal;
  }

  on bob {
    device    /dev/drbd0;
    disk      /dev/sda6;
    address   10.0.0.2:7788;
    meta-disk internal;
  }
}

resource r0-U {
  net {
    protocol A;
  }

  stacked-on-top-of r0 {
    device    /dev/drbd10;
    address   192.168.42.1:7788;
  }

  on charlie {
    device    /dev/drbd10;
    disk      /dev/hda6;
    address   192.168.42.2:7788; # 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.

5.18.3. 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.

5.19. Using DRBD Proxy

5.19.1. 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.

5.19.2. 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_3.0.0_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-3.0-3.0.0-1.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.

5.19.3. 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/
```

5.19.4. Configuration

DRBD Proxy is configured in DRBD's main configuration file. It is configured by an additional options 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 {
    net {
        protocol A;
    }
    device      minor 0;
    disk        /dev/sdb1;
    meta-disk    /dev/sdb2;

    proxy {
        memlimit 100M;
        plugin {
            zlib level 9;
        }
    }

    on alice {
        address 127.0.0.1:7789;
        proxy on alice {
            inside 127.0.0.1:7788;
            outside 192.168.23.1:7788;
        }
    }

    on bob {
        address 127.0.0.1:7789;
        proxy on bob {
            inside 127.0.0.1:7788;
            outside 192.168.23.2:7788;
        }
    }
}

```

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.

5.19.5. 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.

```
add connection <name> <listen-lan-ip>:<port> <remote-proxy-ip>:<port>
```

```
<local-proxy-wan-ip>:<port> <local-drbd-ip>:<port>
```

Creates a communication path between two DRBD instances.

```
set memlimit <name> <memlimit-in-bytes>
```

Sets memlimit for connection <name>

```
del connection <name>
```

Deletes communication path named name.

```
show
```

Shows currently configured communication paths.

```
show memusage
```

Shows memory usage of each connection.

```
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.

```
shutdown
```

Shuts down the drbd-proxy program. Attention: this unconditionally terminates any DRBD connections running.

Examples:

```
drbd-proxy-ctl -c "list hconnections"
```

prints configured connections and their status to stdout

Note that the quotes are required.

```
drbd-proxy-ctl -c "list subconnections" | cut -f 2,9,13
```

prints some more detailed info about the individual connections

```
watch -n 1 'drbd-proxy-ctl -c "show memusage"'
```

monitors memory usage.

Note that the quotes are required as listed above.

While the commands above are only accepted from UID 0 (i.e., the **root** user), there's one (information gathering) command that 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.

`print details`

This prints detailed statistics for the currently active connections.

Can be used for monitoring, as this is the only command that may be sent by a user with UID

`quit`

Exits the client program (closes control connection).

5.19.6. About DRBD Proxy plugins

Since DRBD proxy 3.0 the proxy allows to enable a few specific plugins for the WAN connection. The currently available plugins are `zlib` and `lzma`.

The `zlib` plugin uses the GZIP algorithm for compression. The advantage is fairly low CPU usage.

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`. The `lzma` plugin has to be enabled in your license.

Please contact Linbit to find the best settings for your environment - it depends on the CPU (speed, threading count), memory, input and the available output bandwidth.

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`.

5.19.7. 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
```

5.19.8. 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 6. Troubleshooting and error recovery

This chapter describes tasks to be performed in the event of hardware or system failures.

6.1. 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).

6.1.1. 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 dstate` command, you will now be able to verify that the resource is now in *diskless mode*:

```
drbdadm dstate <resource>
Diskless/UpToDate
```

If the disk failure has occurred on your primary node, you may combine this step with a switch-over operation.

6.1.2. 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 dstate` command to verify that the resource is in fact running in diskless mode.

6.1.3. 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, this has to be modified accordingly in the DRBD configuration file.

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 successfully 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 [/proc/drbd](#), as with any background synchronization.

6.1.4. 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 successfully created.

drbdadm attach <resource>
drbdadm invalidate <resource>
```

Here, the `drbdadm invalidate` command triggers synchronization. Again, sync progress may be observed via [/proc/drbd](#).

6.2. 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 *WfConnection* 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 internal information on which blocks are being modified while disconnected.

6.2.1. 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 on line. When that happens, the two nodes will simply re-establish connectivity upon system start-up. After this, DRBD replicates 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.

6.2.2. 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](#).

6.2.3. 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.

- Install the base system and applications.
- Install DRBD and copy `/etc/drbd.conf` and all of `/etc/drbd.d/` from the surviving node.
- 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 node.

6.3. 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 *WFOConnection* (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:



The split brain victim needs to be in the connection state of *StandAlone* or the following commands will return an error. You can ensure it is standalone by issuing:

```
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 connect <resource>
```

You may omit this step if the node is already in the *WFOConnection* state; it will then reconnect automatically.

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 has its modifications overwritten by the remaining primary node.



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 propagate to the victim.

After re-synchronization has completed, the split brain is considered resolved and the two nodes form a fully consistent, redundant replicated storage system again.

DRBD-enabled applications

Chapter 7. 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.

7.1. 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.

7.2. Adding a DRBD-backed service to the cluster configuration

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 6. Pacemaker configuration for DRBD-backed MySQL service

```
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" \
                    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.

7.3. 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> {
    disk {
        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**.

7.3.1. 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";
        ...
    }
    disk {
        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 `WfConnection`. Allow a few seconds to pass, and you should see the `disk state` become `indexterm: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.

7.3.2. 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> {
  disk {
    fencing resource-only;
    ...
  }
  handlers {
    fence-peer "/usr/lib/drbd/crm-fence-peer.sh";
    after-resync-target "/usr/lib/drbd/crm-unfence-peer.sh";
    ...
  }
  ...
}
```

Thus, if the DRBD replication link becomes disconnected, the `crm-fence-peer.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.

7.4. Using stacked DRBD resources in Pacemaker clusters

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.

7.4.1. 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.

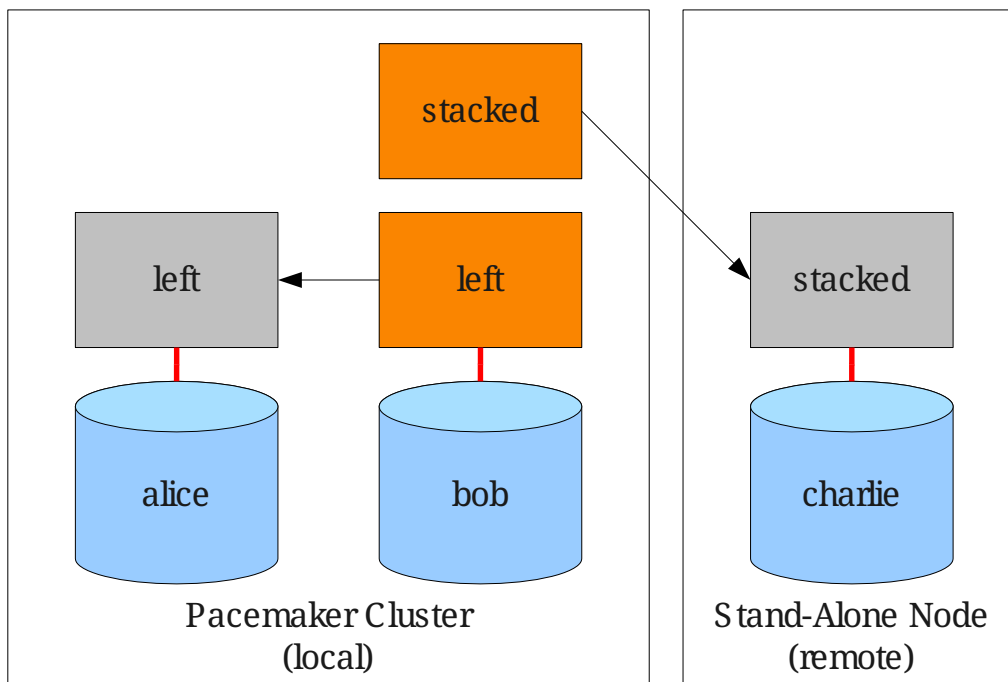


Figure 6. DRBD resource stacking in Pacemaker clusters

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 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' and '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](#).

7.4.2. 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.

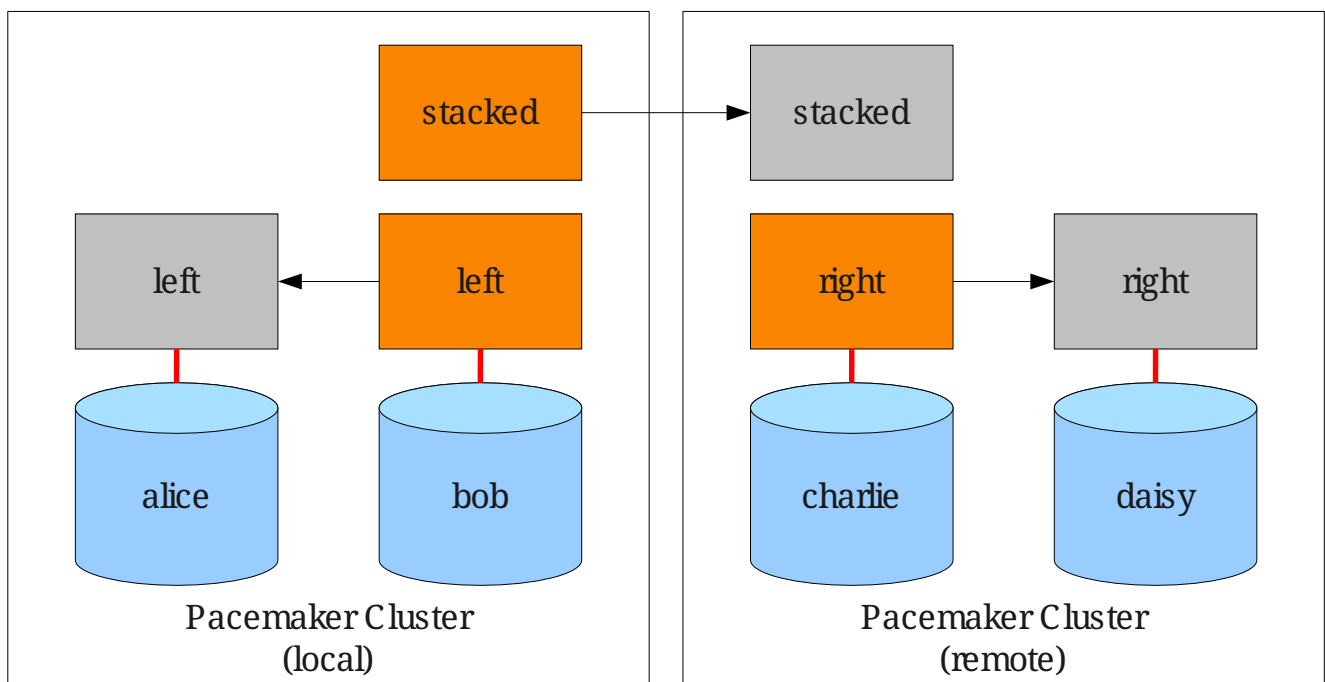


Figure 7. DRBD resource stacking in Pacemaker clusters

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 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"`).

7.5. 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.

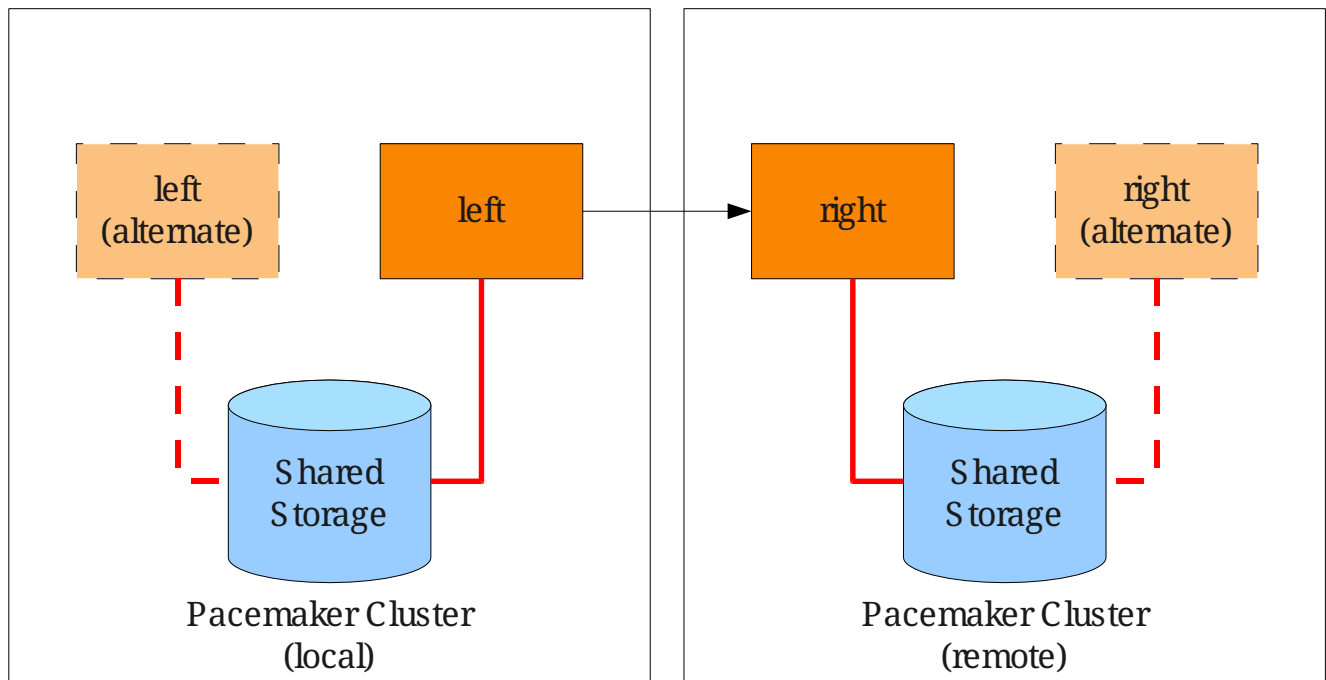


Figure 8. Using DRBD to replicate between SAN-based clusters

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.

7.5.1. 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.

7.5.2. 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"`).

7.5.3. 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 8. 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*.

8.1. Red Hat Cluster background information

8.1.1. 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.

8.1.2. 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`.

8.2. 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](#).

8.2.1. 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.

8.3. 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 GFS2 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.

8.3.1. 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="10.9.9.180" monitor_link="1"/>
    <mysql config_file="/etc/my.cnf"
      listen_address="10.9.9.180"
      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 9. 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.

9.1. 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.

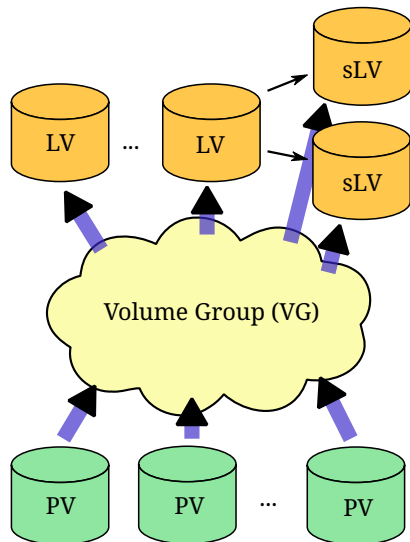


Figure 9. LVM overview

9.2. 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.

9.3. 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 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 7. 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 every volume the resources contains immediately before 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.

9.4. 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 node.



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).

9.5. 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.

9.6. 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:

```
# 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.

9.6.1. 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.

9.7. 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 8. 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 10. Using GFS2 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) version 2 in a nutshell.

For a more detailed howto please consult our tech-guide on [GFS in dual-primary setups](#).



This guide describes a dual-primary setup with DRBD. Dual-primary setups **can easily destroy data** if not configured properly!

Please always read our tech-guide "[Dual primary: think twice](#)", in advance, if you are planning to configure a DRBD dual-primary resource.

If you are not clear or uncertain of anything within this document you may want to consult with the friendly experts at LINBIT beforehand.

10.1. 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 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. Additionally the connection to the Pacemaker cluster manager is explained, which will take care of resource management und STONITH.

GFS, Pacemaker 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.

10.2. Creating a DRBD resource suitable for GFS2

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](#). Promoting the resource on both nodes and starting the GFS filesystem will be handled by Pacemaker. To prepare your DRBD resource, include the following lines in the resource configuration:

```
resource <resource> {
  net {
    allow-two-primaries;
    after-sb-0pri discard-zero-changes;
    after-sb-1pri discard-secondary;
    after-sb-2pri disconnect;
    ...
  }
  ...
}
```



By configuring auto-recovery policies, you are 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 resource](#) to the primary role on both nodes.



Again: Be aware to configure fencing/STONITH and test the setup extensively to cover all possible use cases, especially in dual-primary setups, **before** going into production.

10.2.1. Enable resource fencing for dual-primary resource

In order to enable Resource fencing in DRBD you will need the sections

```
disk {
  fencing resource-and-stonith;
}

handlers {
  fence-peer      "/usr/lib/drbd/crm-fence-peer.sh";
  after-resync-target "/usr/lib/drbd/crm-unfence-peer.sh";
}
```

in your DRBD configuration. These scripts should come with your DRBD installation.



Don't be misled by the shortness of the section [9.1.1. Fencing](#) in the DRBD users guide - with all dual primary setups you have to have fencing in your cluster. See chapters [5.5. Configuring Fence Devices](#) and [5.6. Configuring Fencing for Cluster Members](#) in the Red Hat Cluster documentation for more details.

10.3. Configuring CMAN

GFS needs `cmn`, the Red Hat cluster manager, to work. Since `cmn` is not as flexible and easy to configure we will put `pacemaker` on top of it in the next steps.



If you don't want to use **pacemaker**, please consult the corresponding [manuals](#) for **cman**.

Before we start making a GFS filesystem we will configure **cman**.



If you are configuring a two node cluster, you can not expect it to have a quorum. You will need tell cman to ignore it. This is done by setting

```
# sed -i.orig "s/.*CMAN_QUORUM_TIMEOUT=.*CMAN_QUORUM_TIMEOUT=0/g"
/etc/sysconfig/cman
```

Next create a **cman** cluster configuration in **/etc/cluster/cluster.conf**:

```
<?xml version="1.0"?>
<cluster name="my-cluster" config_version="1">
  <logging debug="off"/>
  <clusternodes>
    <clusternode name="gfs-machine1" nodeid="1">
      <fence>
        <method name="pcmk-redirect">
          <device name="pcmk" port="gfs-machine1"/>
        </method>
      </fence>
    </clusternode>
    <clusternode name="gfs-machine2" nodeid="2">
      <fence>
        <method name="pcmk-redirect">
          <device name="pcmk" port="gfs-machine2"/>
        </method>
      </fence>
    </clusternode>
  </clusternodes>
  <fencedevices>
    <fencedevice name="pcmk" agent="fence_pcmk"/>
  </fencedevices>
</cluster>
```

This tells **cman** that the clustername is **my-cluster**, the cluster node names are **gfs-machine1** and **gfs-machine2**, and that fencing will be done by **pacemaker**.

After you have made the configuration start **cman**.

10.4. Creating a GFS2 filesystem

In order to create a GFS filesystem on your dual-primary DRBD resource, issue this command on (only) **one** (!) node (which must be *Primary*):

```
mkfs -t gfs2 -p lock_dlm -j 2 -t <cluster>:<name> /dev/<drbd-resource>
```

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 in the GFS cluster; since DRBD does not support more than two nodes, the value to set here is always 2.



With DRBD 9 it is possible to share the same disk among more than two nodes; if you want to do that, you'll either have to specify a higher number of journals or create the journals in the live file system.

The `-t` option, 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.

10.5. Using your GFS2 filesystem with Pacemaker

If you want to use Pacemaker as the cluster resource manager, you will have to set up your current configuration and tell Pacemaker to manage your resources.



Make sure to configure Pacemaker also to take care of all the fencing/STONITH actions (see our tech-guide on [GFS in dual-primary setups](#) for further details).

For Pacemaker configuration make a setup as described in [8.2. Adding a DRBD-backed service to the cluster configuration](#).

Since it is a dual-primary setup consider the following changes to the Master-Slave set:

```
crm(live)configure# ms ms_drbd_xyz drbd_xyz \  
                    meta master-max="2" master-node-max="1" \  
                        clone-max="2" clone-node-max="1" \  
                        notify="true"
```

Notice that `master-max` is set to 2, which will cause the DRBD resource to be promoted on both cluster nodes.

Furthermore we want the GFS filesystem also to be started on both nodes, so we simply add a clone of the filesystem primitive:

```
crm(live)configure# clone cl_fs_xyz p_fs_xyz meta interleave="true"
```

Chapter 11. 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:



```
disk {
    fencing resource-and-stonith;
}
handlers {
    # Make sure the other node is confirmed
    # dead after this!
    fence-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.

11.1. 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 [Pacemaker](#) cluster 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.

11.2. 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](#). And, it is necessary for the resource to switch to the primary role immediately after startup. To do all this, include the following lines in the resource configuration:

```
resource <resource> {
    startup {
        become-primary-on both;
        ...
    }
    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 resource](#) to the primary role on both nodes.

11.3. 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.

11.4. Pacemaker OCFS2 management

11.4.1. Adding a Dual-Primary DRBD resource to Pacemaker

An existing [Dual-Primary DRBD resource](#) may 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.

11.4.2. 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.

11.4.3. 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.

11.4.4. Adding required Pacemaker constraints to manage OCFS2 filesystems

In order to tie all OCFS2-related resources and clones together, add the following constraints to your Pacemaker configuration:

```
order o_ocfs2 inf: ms_drbd_ocfs2:promote cl_ocfs2mgmt:start cl_fs_ocfs2:start
colocation c_ocfs2 inf: cl_fs_ocfs2 cl_ocfs2mgmt ms_drbd_ocfs2:Master
```

11.5. 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.

11.5.1. 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 your 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
```

11.5.2. 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 12. Using Xen with DRBD

This chapter outlines the use of DRBD as a Virtual Block Device (VBD) for virtualization environments using the Xen hypervisor.

12.1. 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.

12.2. Setting DRBD module parameters for use with Xen

For Xen Domain-0 kernels, it is recommended to load the DRBD module with the option `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
```

12.3. 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 mode](#) for 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.

12.4. 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.

12.5. 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.

12.6. 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.

12.7. 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 13. Measuring block device performance

13.1. 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 $TEST_RESOURCE)
# TEST_LL_DEVICE=$(drbdadm sh-ll-dev $TEST_RESOURCE)
# drbdadm primary $TEST_RESOURCE
# for i in $(seq 5); do
#     dd if=/dev/zero of=$TEST_DEVICE bs=512M count=1 oflag=direct
# done
# drbdadm down $TEST_RESOURCE
# for i in $(seq 5); do
#     dd if=/dev/zero of=$TEST_LL_DEVICE bs=512M count=1 oflag=direct
# done
```

This test simply writes a 512M chunk 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 significantly reduced performance on the first `dd` run. This is due to the Activity Log being "cold", and is no cause for concern.

13.2. 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 $TEST_RESOURCE)
# TEST_LL_DEVICE=$(drbdadm sh-ll-dev $TEST_RESOURCE)
# drbdadm primary $TEST_RESOURCE
# dd if=/dev/zero of=$TEST_DEVICE bs=512 count=1000 oflag=direct
# drbdadm down $TEST_RESOURCE
# dd if=/dev/zero of=$TEST_LL_DEVICE bs=512 count=1000 oflag=direct
```

This test writes 1,000 512-byte chunks of data to your DRBD device, and then to its backing device for comparison. 512 bytes is the smallest block size a Linux system (on all architectures except s390) is 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 sector write.

Chapter 14. 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.

14.1. 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 of disks that can be written to in parallel. A single, reasonably recent, SCSI or SAS disk will typically allow streaming writes of roughly 40MB/s to the single disk. 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.



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.

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 protocol (such as 10 Gigabit Ethernet), or by using link aggregation over several network links, as one may do using the Linux **bonding** network driver.

14.2. 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 *minimum* between the two establishes the theoretical throughput *maximum* available to DRBD. DRBD then reduces that throughput maximum by its additional throughput overhead, which can be expected to be less than 3 percent.

- Consider the example of two cluster nodes containing I/O subsystems capable of 200 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 107 MB/s maximum DRBD throughput.

- By contrast, if the I/O subsystem is capable of only 100 MB/s for sustained writes, then it constitutes the bottleneck, and you would be able to expect only 97 MB/s maximum DRBD throughput.

14.3. Tuning recommendations

DRBD offers a number of configuration options which may have an effect on your system's throughput. This section lists 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.

14.3.1. Setting `max-buffers` and `max-epoch-size`

These options affect write performance on the secondary node. `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;
    ...
  }
  ...
}
```

14.3.2. Tweaking the I/O unplug watermark

The I/O unplug watermark is a tunable which affects how often the I/O subsystem's controller is "kicked" (forced to process pending I/O requests) during normal operation. There is no universally recommended setting for this option; this is greatly hardware dependent.

Some controllers perform best when "kicked" frequently, so for these controllers it makes sense to set this fairly low, perhaps even as low as DRBD's allowable minimum (16). Others perform best when left alone; for these controllers a setting as high as `max-buffers` is advisable.

```
resource <resource> {
  net {
    unplug-watermark 16;
    ...
  }
  ...
}
```

14.3.3. 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 512 KiB, or perhaps even more. Send buffer sizes of more than 2 MiB are generally not recommended (and are also unlikely to produce any throughput improvement).

```
resource <resource> {
  net {
    sndbuf-size 512k;
    ...
  }
  ...
}
```

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;
    ...
  }
  ...
}
```

14.3.4. 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 3389;
        ...
    }
    ...
}
```

14.3.5. 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;
        ...
    }
    ...
}
```

Chapter 15. 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.

15.1. 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

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.

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 cache memory available for the disk write cache.

Network latency

Network latency is, in essence, the packet round-trip time () 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 Gigabit Ethernet links, which typically is on the order of 100 to 200 microseconds (μ s) packet RTT.

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.

15.2. 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. 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.

15.3. Tuning recommendations

15.3.1. 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.

15.3.2. Modifying the network MTU

When a block-based (as opposed to extent-based) filesystem is layered above DRBD, 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".



Block-based file systems include ext3, ReiserFS (version 3), and GFS. Extent-based file systems, in contrast, include XFS, Lustre and OCFS2. Extent-based file systems are expected to benefit from enabling Jumbo frames only if they hold few and large files.

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).

15.3.3. 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 in reasonably recent kernel configurations (post-2.6.18 for most distributions).

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 **/boot/grub/menu.lst** if you are using the GRUB bootloader) and add **elevator=deadline** to your list of kernel boot options.

Learning more about DRBD

Chapter 16. 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.

16.1. 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* and *externally*. Which method is used is configurable on a per-resource basis.

16.1.1. 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](#).

16.1.2. 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.

16.1.3. Estimating meta data size

You may calculate the exact space requirements for DRBD's meta data using the following formula:

$$M_s = \left\lceil \frac{C_s}{2^{18}} \right\rceil * 8 * N + 72$$

Figure 10. 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 by issuing `blockdev --getsz <device>`.

The result, M_s , is also expressed in sectors. To convert to MB, divide by 2048 (for a 512-byte sector size, which is the default on all Linux platforms except s390).

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 11. Estimating DRBD meta data size (approximately)

16.2. 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.

16.2.1. 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 a 8-byte, universally unique identifier (UUID).

16.2.2. The generation identifier tuple

DRBD keeps four 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 UUID

This is the UUID of the generation against which the on-disk sync bitmap is tracking changes. As the on-disk sync bitmap itself, this identifier is only relevant while in disconnected mode. If the resource is *Connected*, this UUID is always empty (zero).

Two Historical UUIDs

These are the identifiers of the two data generations preceding the current one.

Collectively, these four items are referred to as the *generation identifier tuple*, or GI tuple" for short.

16.2.3. How generation identifiers change

Start of a new data generation

When a node loses connection to its peer (either by network failure or manual intervention), DRBD modifies its local generation identifiers in the following manner:

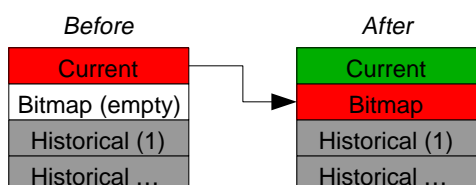


Figure 12. GI tuple changes at start of a new data generation

1. A new UUID is created for the new data generation. This becomes the new current UUID for the primary node.
2. The previous 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, the GI tuple remains unchanged.

Start of re-synchronization

Upon the initiation of re-synchronization, DRBD performs these modifications on the local generation identifiers:

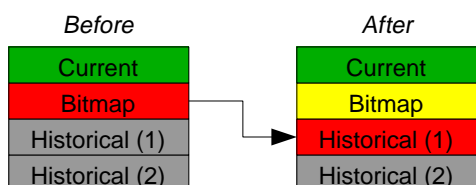


Figure 13. GI tuple changes at start of re-synchronization

1. The current UUID on the synchronization source remains unchanged.
2. The bitmap UUID on the synchronization source is rotated out to the first historical UUID.
3. A new bitmap UUID is generated on the synchronization source.

4. This UUID becomes the new current UUID on the synchronization target.
5. The bitmap and historical UUID's on the synchronization target remain unchanged.

Completion of re-synchronization

When re-synchronization concludes, the following changes are performed:

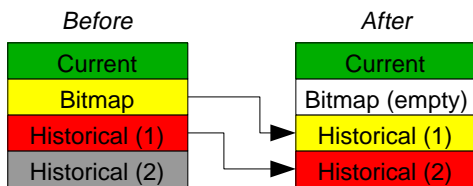


Figure 14. GI tuple changes at completion of re-synchronization

1. The current UUID on the synchronization source remains unchanged.
2. The bitmap UUID on the synchronization source is rotated out to the first historical UUID, with that UUID moving to the second historical entry (any existing second historical entry is discarded).
3. The bitmap UUID on the synchronization source is then emptied (zeroed).
4. The synchronization target adopts the entire GI tuple from the synchronization source.

16.2.4. 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.

16.3. The Activity Log

16.3.1. 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.

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 of 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, that node must be fully synchronized anew 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, rather than the full device. This drastically reduces synchronization time after an active node crash.

16.3.2. 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.

16.3.3. Selecting a suitable Activity Log size

The definition 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 15. Active extents calculation based on sync rate and target sync time

R is the synchronization rate, given in MB/s. t_{sync} is the target synchronization time, in seconds. E is the resulting number of active extents.

To provide an example, suppose our cluster has an I/O subsystem with a throughput rate of 90 MiByte/s that was configured to a synchronization rate of 30 MiByte/s ($R=30$), and we want to keep our target synchronization time at 4 minutes or 240 seconds ($t_{sync}=240$):

Figure 16. Active extents calculation based on sync rate and target sync time (example)

The exact result is 1800, but since DRBD's hash function for the implementation of the AL works best if the number of extents is set to a prime number, we select 1801.

16.4. The quick-sync bitmap

The quick-sync bitmap is the internal data structure which DRBD uses, on a per-resource 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.

16.5. The peer fencing interface

DRBD has a defined interface for the mechanism that fences 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 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 <i>Inconsistent</i> .
4	Peer's disk state was successfully set to <i>Outdated</i> (or was <i>Outdated</i> 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 17. Getting more information

17.1. Commercial DRBD support

Commercial DRBD support, consultancy, and training services are available from the project's sponsor company, [LINBIT](#).

17.2. 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 <https://lists.linbit.com/listinfo/drbd-user>. A complete list archive is available at <https://lists.linbit.com/pipermail/drbd-user>.

17.3. 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](#),
- [#linux-ha](#),
- [#linux-cluster](#).

Getting in touch on IRC is a good way of discussing suggestions for improvements in DRBD, and having developer level discussions.

17.4. Official Twitter account

[LINBIT](#) maintains an official [twitter account](#).

If you tweet about DRBD, please include the [#drbd](#) hashtag.

17.5. 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:

- Lars Ellenberg. 'DRBD v8.0.x and beyond'. 2007. Available at <http://www.drbd.org/fileadmin/drbd/publications/drbd8.linux-conf.eu.2007.pdf>
- Philipp Reisner. 'DRBD v8 - Replicated Storage with Shared Disk Semantics'. 2007. Available at <http://www.drbd.org/fileadmin/drbd/publications/drbd8.pdf>.
- Philipp Reisner. 'Rapid resynchronization for replicated storage'. 2006. Available at http://www.drbd.org/fileadmin/drbd/publications/drbd-activity-logging_v6.pdf

17.6. Other useful resources

- Wikipedia keeps [an entry on DRBD](#).
- Both the [Linux-HA wiki](#) and
- [ClusterLabs](#) 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 8.4. It highlights some important changes to DRBD's configuration and behavior.

A.1. Volumes

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.

A.1.1. 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.

In DRBD 8.3 and earlier, links in `/dev/drbd/by-disk` point to single block devices:

Listing 9. udev managed DRBD symlinks in DRBD 8.3 and earlier

```
lrwxrwxrwx 1 root root 11 2011-05-19 11:46 /dev/drbd/by-res/home ->
    ../../drbd0
lrwxrwxrwx 1 root root 11 2011-05-19 11:46 /dev/drbd/by-res/data ->
    ../../drbd1
lrwxrwxrwx 1 root root 11 2011-05-19 11:46 /dev/drbd/by-res/nfs-root ->
    ../../drbd2
```

In DRBD 8.4, since a single resource may correspond to multiple volumes, `/dev/drbd/by-res/<resource>` becomes a *directory*, containing symlinks pointing to individual volumes:

Listing 10. udev managed DRBD symlinks in DRBD 8.4

```
lrwxrwxrwx 1 root root 11 2011-07-04 09:22 /dev/drbd/by-res/home/0 ->
    ../../drbd0
lrwxrwxrwx 1 root root 11 2011-07-04 09:22 /dev/drbd/by-res/data/0 ->
    ../../drbd1
lrwxrwxrwx 1 root root 11 2011-07-04 09:22 /dev/drbd/by-res/nfs-root/0 ->
    ../../drbd2
lrwxrwxrwx 1 root root 11 2011-07-04 09:22 /dev/drbd/by-res/nfs-root/1 ->
    ../../drbd3
```

Configurations where filesystems are referred to by symlink must be updated when moving to DRBD 8.4, usually by simply appending `/0` to the symlink path.

A.2. 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 not present in prior DRBD releases, 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.2.1. 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 11. 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 12. Pre-DRBD 8.4 configuration example with boolean options in `common` section

```
common {
  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 13. DRBD 8.4 configuration example with boolean options in **common** section*

```
common {
    md-flushes no;
}
resource test {
    disk {
        md-flushes yes;
    }
}
```

A.2.2. **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 14. 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 15. DRBD 8.4 configuration example with **syncer** section replaced*

```
resource test {
    disk {
        al-extents 3389;
    }
    net {
        verify-alg md5;
    }
    ...
}
```

A.2.3. **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 16. Pre-DRBD 8.4 configuration example with standalone **protocol** option*

```
resource test {
  protocol C;
  ...
  net {
    ...
  }
  ...
}
```

The equivalent DRBD 8.4 configuration syntax is:

*Listing 17. DRBD 8.4 configuration example with **protocol** option within **net** section*

```
resource test {
  net {
    protocol C;
    ...
  }
  ...
}
```

A.2.4. 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 18. 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:

```
resource test {
  options {
    cpu-mask ff;
    on-no-data-accessible suspend-io;
  }
  ...
}
```

A.3. On-line changes to network communications

A.3.1. 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 20. Changing replication protocol while connection is established

```
drbdadm net-options --protocol=A <resource>
```

A.3.2. 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.4. Changes to the `drbdadm` command

A.4.1. 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 21. 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 22. 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.4.2. `--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 23. Pre-DRBD 8.4 initial sync `drbdadm` commands

```
drbdadm -- --overwrite-data-of-peer primary <resource>
```

Use the command below instead:

Listing 24. DRBD 8.4 initial sync `drbdadm` commands

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

A.5. 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.5.1. 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.5.2. 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.5.3. 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.5.4. 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 25. 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.5.5. 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.