

DRBD v8

Replicated Storage with Shared Disk Semantics

Philipp Reisner, Lars Ellenberg

August 6, 2007

Abstract

Errata: edited the Concurrent Write Detection Algorithm to match reality. Original paper: 18th September 2005, Proceedings of the 12th International Linux System Technology Conference.

DRBD is a well established Linux software component to build HA (high availability) clusters out of common off-the-shelf hardware. DRBD's key point is to replace shared storage systems by shared-nothing networked mirroring.

The most outstanding new feature is certainly our new "shared disk" mode, i.e. support for shared-disk file systems and other shared storage-aware applications. Provided that you use one of OpenGFS, GFS, OCFS2, this means that applications have read and write access to the mirrored data set on both nodes at the same time, since you can mount the storage on both peers simultaneously.

Besides that, DRBD 8 supports effective resource level fencing. It prevents the divergence of the nodes' copies of the data when DRBD's communication path fails.

Although DRBD has been widely used and under active development for 5 years now, it turned out that the method to determine the node with the up-to-date data was suboptimal. It was replaced by a new scheme based on UUIDs. This new scheme also works correctly in multi-node setups.

1 Overview

The DRBD software consists of a Linux kernel module and some tools to control the operation of the module. To file systems and user space this is just another block device. However, all write requests to this block device are written to the backing storage devices on both nodes in the cluster. Read requests are carried out locally for performance, mainly latency, reasons. Unless, of course, the local disk is broken, in which case reads are done via the network, too.

Usually DRBD is controlled by a cluster manager software called heartbeat, which initiates the failover process, in case the active node leaves the cluster unexpectedly (crashes).

Each peer of a DRBD resource acts in one of two roles, it may be either secondary or primary. All modifications of the data must be initiated by a peer acting as primary. This implies that when acting as secondary, no write access to the data is provided to user space, and no file systems might be mounted on the DRBD device. Only in primary role write access to the data is granted. The role assignment is usually done by the cluster management software.

2 Shared disk semantics

With the success of storage area networks (SAN) in the market place, applications for shared disks are readily available. Linux file systems that make use of shared disks are OCFS2, GFS and openGFS.

As the name *shared disk* implies, a shared disk is a shared resource. There are not only processes competing for this resource, these processes are even distributed over several computer systems.

If there are uncoordinated writes to the same area of a shared disk by several of the connected computer systems, the resulting on-disk data is some “random” combination of the original writes. Would the participating nodes read the data afterwards, all of them would read the same random mixture of the previously written data. Users of a shared disk therefore typically have some sort of “distributed lock manager” (DLM) to coordinate their accesses.

In order to allow the deployment of shared disk file systems (OCFS2, GFS, openGFS etc.) DRBD would not need to implement this aspect of shared disks, because uncoordinated writes from a shared disk file system can only happen if the file system’s lock manager does not work.

2.1 Replicated storage systems

Now, DRBD is *replicated*, not *shared*. We need to make sure that both copies are always¹ identical. And we do not want to introduce a distributed locking mechanism ourselves.

Regardless of whether the locking mechanisms of our users (GFS, OCFS2 etc.) do work or not², if we mirrored competing writes naively, we would get inconsistent data on the participating nodes.

In this example system N1 and N2 issue write requests at nearly the same time to the same location. In the end, on N1 we have the data that was written on N2, and N2 has the data of N1. On both nodes the remote version overwrites the local version.

With a real shared disk, both nodes would read the same data, even though it would not be deterministic which version of it. With the naive approach, depending on the exact timing of events, DRBD would end up with diverging versions of data on the nodes.

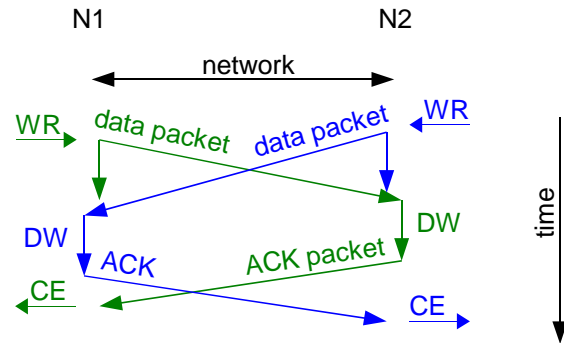


Figure 1: uncoordinated writes;
WR: write request; DW: disk write; CE: completion event

3 Shared disk emulation

Write accesses to a replicated storage can be seen as the timing of 5 events (figure 1):

1. A write request is issued on the node, which is sent to the peer, as well as submitted to the local IO subsystem (WR).
2. The *data packet* arrives at the peer node and is submitted to the peers io subsystem.
3. The write to the peers disk finishes, and an *ACK packet* is sent back to the origin node.
4. The *ACK packet* arrives at the origin node.
5. The local write finishes (local completion).

Events 1. to 4. always occur in the same order. The local completion can happen anytime, independently of 2. to 4. The timing can vary drastically.

If we look at two competing write accesses to a location of the replicated storage, we have two classes of events with five events each, shuffled, where we still can distinguish four different orderings within each

¹If DRBD is connected, not degraded, and we look at it in a moment when no requests are on the fly.

²Of course we expect them to work. But there will always be someone trying to do this with reiserfs, believing DRBD will magically make it cluster aware. We have to guarantee that we scramble both replication copies in the same way...

class. Expressed mathematically, the number of different possible timings for the naive implementation is $\frac{\text{number of combinations}}{\text{number of indistinguishable combinations}}$, which is $\frac{(5+5)!}{\frac{5!}{4} \times \frac{5!}{4}} = 4032$, or 2016, if we take the symmetry into account.

This quite impressive number can be reduced into a few different cases if we “sort” by the timing of the “would be” disk writes: writes are strictly in order (trivial case; 96 combinations); the writes can be reordered easily so they are in the correct order again (remote request while local request is still pending; 336 combinations); the conflict can be detected and solved by just one node, without communication (local request while remote still pending; 1080 combinations); the write requests have been issued quasi simultaneously (2520 combinations).

3.1 The trivial case

If at first all five events of one node happen, and then all events of the other node, these two writes are actually not competing, and require no special handling. This is mentioned here only for the sake of completeness.

The same applies for all combinations where the write request comes in after the ACK for the last mirror request was sent, as long as the ACK arrives before the data packet, and there is no more local write pending (there are no crossing lines in the illustration; 96 of 4032).

3.2 Remote request while local request still pending

In the illustrations (Figure 2 to 7) we contrast the naive (on the left) versus the actual implementation.

A data packet might overtake an ACK packet on the network, since they use different communication channels. Although this case is quite unlikely, we have to take it into account. We solve this by attaching sequence numbers to the packets. See figure 2.

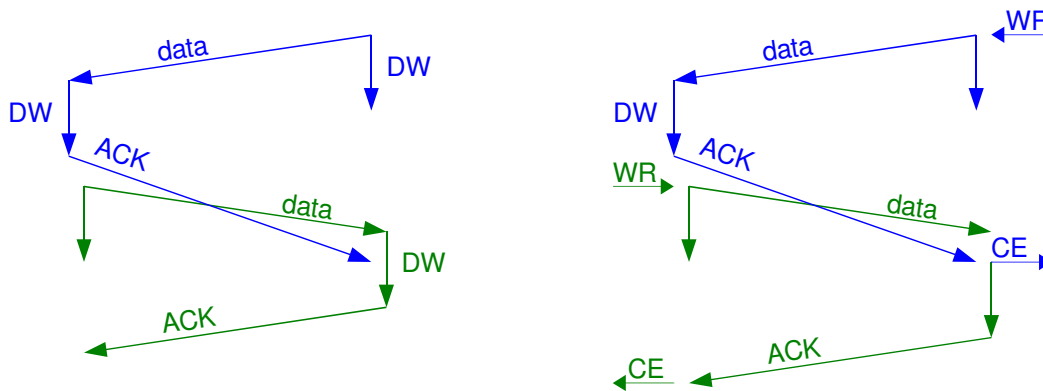


Figure 2: A data packet overtakes an ACK packet; 126 of 4032

High disk latency on N2. This could happen by IO reordering in the layers below us. The problem is that we would end up with N1’s data on the disk of N1, and N2’s data on the disk of N2. The solution is to delay the on-behalf-of-N1-write on N2 until the local write is done. See figure 3.

3.3 Local request while remote request still pending

The conflicting write gets issued while we process a write request from the peer node. We solve this by simply discarding the locally issued write request, and signal completion immediately. See figure 4.

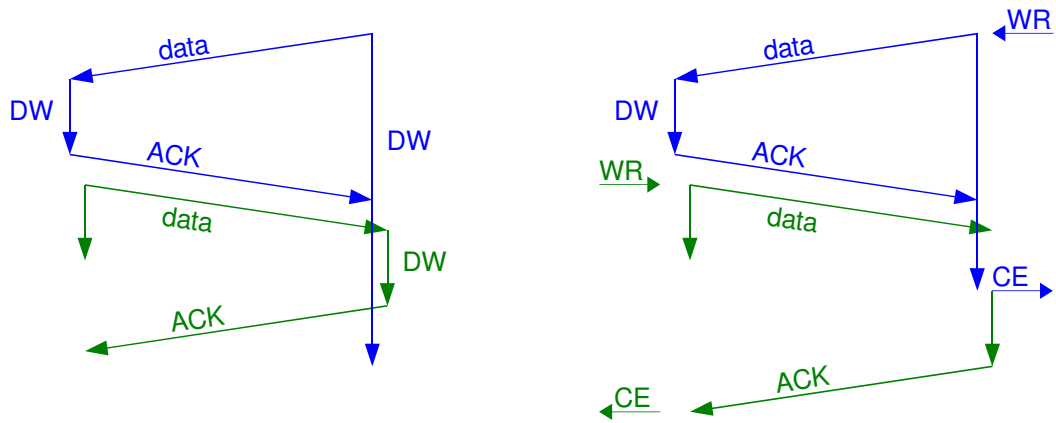


Figure 3: High disk latency; 210 of 4032

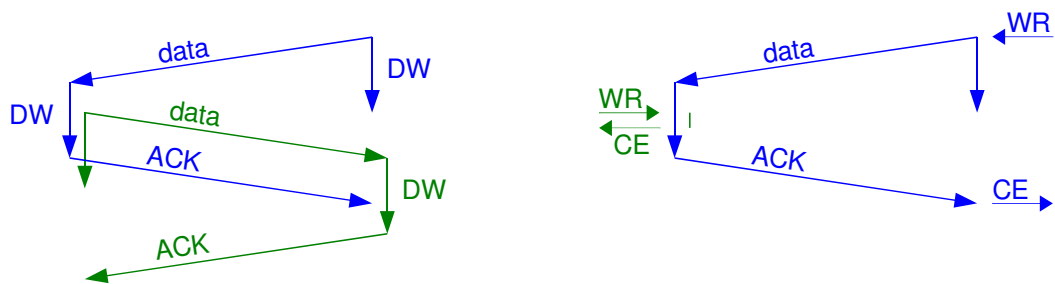


Figure 4: Concurrent write while writing to the backing storage; 1080 of 4032

3.4 Quasi simultaneous writes

All these cases have in common that the write requests are issued at about the same time, i.e. the nodes start to process the request and do not know that there is a conflicting write request on the other node. So both data packets get shipped. One of the data packets has to be discarded when received. This is achieved by flagging one node with the discard-concurrent-writes-flag.

Concurrent writes, network latency is lower than disk latency

As illustrated, in the end each node ends up with the respective other node's data. The solution is to flag one node (in the example N2 has the discard-concurrent-writes-flag). As we can see on the right side, now both nodes end up with N2's data. See figure 5.



Figure 5: Concurrent write with low network latency

Concurrent writes, high latency for data packets

This case is also handled by the just introduced flag. See figure 6.

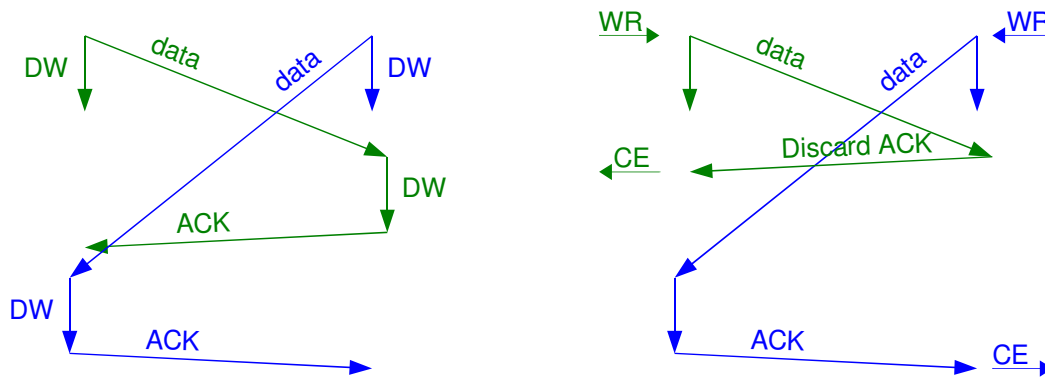


Figure 6: Concurrent write with high network latency

Concurrent writes with high latency for data packets

The problem now is that N2 cannot detect that this was a concurrent write, since it got the ACK before the conflicting data packet comes in. To solve this, we defer the local submission of the received data, until we see the DiscardACK from the peer. See figure 7.

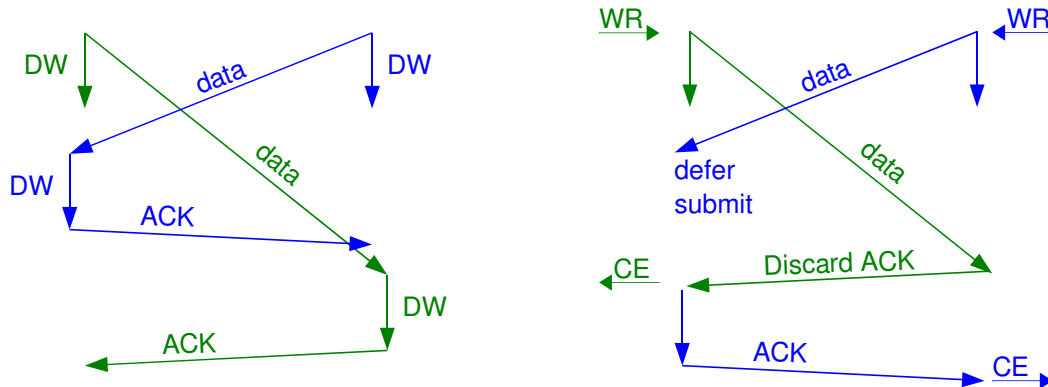


Figure 7: Concurrent write with high network latency

3.5 The Algorithm for Concurrent Write Arbitration

- We arbitrarily select one node and mark it with the discard-concurrent-writes-flag. Each data packet and each ACK packet get a sequence number, which is increased with every packet sent.

A data packet comes in

1. If the sequence number of the data packet is higher than $\text{last_seq}+1$, sleep until $\text{last_seq}+1 = \text{seq_num}(\text{data packet})$ [see 3.2 and figure 2].
2. Do I have a concurrent request, i.e. do I have a request to the same block in my transfer hash tables? If not, write now.
3. If a concurrent request was detected:
 - (a) if I have the "discard-concurrent-write-flag": discard the data packet, and send a DiscardACK back to the peer.
 - (b) if I do *not* have the "discard-concurrent-write-flag": defer local submission of this data packet. Wait for local io-completion [see 3.2 and figure 3] and until any pending ACK (respective DiscardACK) has been received [see 3.4, figures 5, 6, 7], then submit locally. Upon local completion, send normal ACK.

An ACK packet is received

- If I get an ACK or DiscardACK, I store the sequence number in last_seq , and wake up any possibly waiting receiver.

A write request gets issued by the upper layers

- If I am currently writing a block from the peer to the same location, discard the request and signal completion to the issuers [see 3.3 and figure 4].

Each time we have a concurrent write access, we print an alert message to the kernel log, since this indicates that some layer above us is seriously broken!

4 Avoiding data diversion

One of the real world problems with DRBD/heartbeat based clusters is that the cluster management software does not know about DRBD's states. The issues that arise from this fact are mainly at concern when DRBD's communication link is broken, but heartbeat still has other communication links.

Here is an example to illustrate the issue:

DRBD's link is broken, each node tries to reach the other node. The cluster manager would happily migrate the services, including the primary role from one node to the other node in the cluster. This of course would cause data diversion.

The idea is, that a primary node needs to mark its peer node as outdated, as soon as communication is lost. Put in an other way, when a degraded cluster node wants to become primary it needs to know that its peer node is marked as outdated before it actually becomes primary. The outdated mark is of course stored permanently in the meta-data area.

On the other hand, a node that is marked as outdated, refuses to become primary. Still, manual override is possible: An administrator might decide that this is the best surviving version of the data, and therefore might forcefully change the role of the node to primary, dropping the out-date mark with that action.

In order to mark the peer node as outdated, the user has to provide an executable to DRBD that runs the out-date command on the peer. In the DRBD distribution an implementation of such a callback shell script is provided, which uses SSH with passwordless keys for this task. An alternative approach would be to use heartbeat's communication layers, but as of the time of writing, this has not yet been implemented.

5 Data Generation UUIDs

The here presented algorithm based on UUIDs replaces the generation-counter based algorithm [Rei01], which was used in DRBD up to now.

5.1 Cluster restart time

At restart of a cluster, which means that both nodes were down, DRBD needs to be able to determine if the two copies of the data are equal, or if one of the copies is outdated. It needs to know which copy is outdated, and it needs to know if the two data sets are related at all.

By all means the most unpleasant case is if the two data sets were related in the past, but both evolved into different versions. This usually happens in a complete split brain situation, where both nodes started their services.

DRBD offers automatic and manual recovery strategies.

Every time the connection is established, DRBD needs to decide whether a resynchronisation process is needed, and who receives the update.

5.2 Meta-data

In a degraded cluster, i.e. a node that is not connected to its peer, DRBD needs to store permanently the user's intent to modify the data set. This information is written to the meta-data area, before the user is allowed to actually open the block device for write accesses.

In the meta-data area we store various flags and some random numbers (UUIDs³) which identify the data generations.

The first time the user changes the role of a disconnected secondary node to primary, the current-UUID gets copied over to the bitmap-UUID, and a new current-UUID is generated. For all actual write accesses to the data, activity-log [Rei04] entries, and later entries in the bitmap, are set, to mark the changes.

If the device gets switched to primary role a second time, there is already a bitmap-UUID. In this case no new current-UUID gets generated, we just need to continue to record all modifications in the activity log and the bit map data structures.

This also happens on a node in primary role, that gets disconnected from its peer. It moves its current-UUID to the bitmap-UUID and creates a new current-UUID.

When we start a resync process, the SyncSource generates a new bitmap-UUID. This same UUID becomes the new current-UUID of the SyncTarget, which also sets its inconsistent-flag.

The bitmap-UUID gets cleared when the resynchronisation process finishes, i.e. all the marks in the bitmap are cleared, and now both data sets are in sync.

5.3 History UUIDs

One of the motivations to develop the UUID based scheme was that the generation based scheme was not able to deal with more than two nodes.

The UUID approach needs a history-UUID slot for each node beyond two nodes. The history-UUID slots get the old value of the bitmap-UUID, when the bitmap-UUID gets cleared.

In the following example UUID values are written as ordered quadruples, prefixed with the node's current role: role<current,bitmap,history1,history2>. In place of the random UUIDs adjacent numbers are used to illustrate the algorithm.

#	node1	node2	node3	comment
1	P<5,0,0,0>	S<5,0,0,0>		node1 connected to node2
2	P<6,5,0,0>	S<5,0,0,0>		node1 is disconnected from node2
3	P<6,5,0,0>		S<0,0,0,0>	node1 gets connected to node3
4	P<6,7,5,0>		S<7,0,0,0>	full resync n1 ⇒ n3
5	P<6,0,7,5>		S<6,0,7,5>	after the resync, node1 is connected to node3
6	P<8,6,7,5>		S<6,0,7,5>	node1 gets disconnected from node3
7	P<8,6,7,5>	S<5,0,0,0>		node1 is configured to talk to n2 again
8	P<8,9,7,5>	S<9,0,5,0>		full resync n1 ⇒ n2
9	P<8,0,9,7>	S<8,0,9,7>		node1 is connected to node2

As of the time of writing I decided to use two history-UUIDs in DRBD v8, therefore DRBD works correctly as long as you do not establish connections between more than 3 nodes.

Note that DRBD only mirrors the data between two nodes. The main improvement here is that the resynchronisation never runs in the wrong direction, regardless of sequence of events on the cluster nodes.

5.4 Split-brain

In a so-called split-brain situation all communication paths in the cluster failed, and each node of the cluster assumes that it is the last surviving node. In such a situation both nodes activate their DRBD resources and therefore we have two different data sets that share a common ancestor.

In the UUID set we observe this as equal bitmap-UUIDs (the common ancestor) but different current-UUIDs. It is still possible in this situation to determine all differing blocks by combining the bitmap information of both nodes.

³universally unique identifier

5.4.1 Automatic split-brain recovery strategies

The manual way to recover from a split-brain situation, is an option named `-discard-my-data` that the user may pass to the command that instructs DRBD to establish a connection to the peer.

There are configurable automatic recovery strategies for all possible role assignments of the connecting peers.

after-sb-0pri Both nodes are secondary, DRBD has the highest freedom of choice.

disconnect Automatic recovery is disabled, simply disconnect.

discard-younger-primary Resynchronize from the node that was primary before the split-brain situation happened.

In order to implement this recovery strategy one of the UUID bits is used to store the role of the node at that time.

discard-least-changes Auto sync from the node that touched more of its blocks during the split-brain situation.

This is implemented by comparing the number of set bits in the bit map.

after-sb-1pri One node is primary, when the two nodes connect for the first time after split-brain.

disconnect Do not try fancy things in this situation, simply disconnect.

consensus Discard the version of the secondary if the outcome of the after-sb-0pri algorithm would also destroy the current secondary's data. Otherwise disconnect.

discard-secondary Overwrite the data set of the current secondary.

panic-primary Always honour the outcome of the after-sb-0pri algorithm. In case it decides that the current secondary node has the right data, panic the current primary node.

after-sb-2pri This is the most likely situation after a real split-brain situation.

disconnect Both nodes go into stand alone mode.

panic Evaluate the after-sb-0pri algorithm, try to become secondary on the resync-target node. In case that fails, e.g. because a file system is mounted on DRBD, that node triggers a kernel panic.

The Algorithm

#	condition	action if condition is true
1	$C_s = 0 \wedge C_p = 0$	No Resynchronisation
2	$C_s = 0 \wedge C_p \neq 0$	Set all bits in the bit map; become target of resync
3	$C_s \neq 0 \wedge C_p = 0$	Set all bits in the bit map; become source of resync
4	$C_s = C_p$	No Resynchronisation necessary
5	$C_s = B_p$	I am target of resync
6	$C_s \in \{H1_p, H2_p\}$	Set all bits in the bit map; become target of resync
7	$B_s = C_p$	I am source of resync
8	$C_p \in \{H1_s, H2_s\}$	Set all bits in the bit map; become source of resync
9	$B_s = B_p \wedge B_s \neq 0$	Split brain detected; try auto recovery strategies
10	$\{H1_p, H2_p\} \cap \{H1_s, H2_s\} \neq \emptyset$	Split brain detected; disconnect
11		Warn about unrelated data; disconnect

C_s is the current-UUID if the host (self); C_p is the current-UUID of the peer node; B stands for the bitmap-UUID and H1 and H2 for the history-UUIDs.

6 Write Barriers and TCQ

Database systems and journaling file systems offer atomic transactions to their users. These software systems provide this property by clever grouping their write requests. They usually write what they intend to do to a journal or log. When this is done, they update the data in their home locations. In case the system crashes at any time during this process, the transaction is either backed out (e.g. the record in the journal was incomplete), or completely carried out (the record from the journal gets applied to the home location once more at recovery time).

6.1 DRBD's Write Barriers

DRBD generates write barriers in the stream of packets it sends to its peer. These barriers are used to express all possible write-after-write dependencies in a stream of write requests that were possibly imposed by the timing of the arrival of write requests and delivery of write completion events. See [Rei01] for an in-depth discussion of the topic. The semantics of DRBD's write barriers is to divide the stream of write requests in two reorder domains.

All requests both before and after the barrier may get re-ordered.

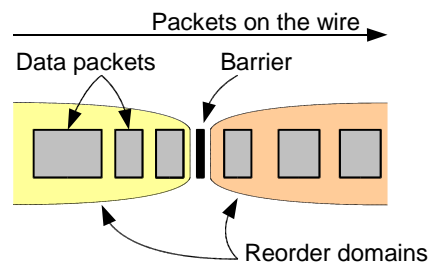


Figure 8: DRBD style barriers

6.2 Linux's Write Barriers

Finally with Linux-2.6 write barriers were introduced in Linux's API. The semantics of these barriers are equal to those of SCSI and SATA. Write requests with a `SIMPLE_QUEUE_TAG` may be reordered by the target device as it wishes. A Write request with an `ORDERED_QUEUE_TAG` splits the stream of write requests into three sets: the requests before the ordered one, the ordered one itself and the requests after the ordered one. This is also known as tagged command queuing (TCQ).

The Linux API allows one to set the `BIO_RW_BARRIER` on individual BIOs, which is then mapped to its counterpart in the SCSI or SATA layers.

As DRBD device we always expose a minimum of capabilities of our two backing storage devices to the rest of the kernel. I.e. if one of our backing storage devices does not support TCQ, this DRBD device as a whole does not support TCQ.

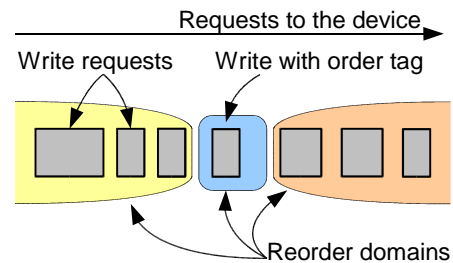


Figure 9: SCSI style barriers

6.3 Write Barrier semantic mapping

As long as it was not possible to express write barriers in the kernel, DRBD had to do a workaround to express these write after write dependencies. In case we get a barrier, we wait until all pending write requests are finished. Then we continue to accept data packets and to issue new write requests.

It is still necessary to implement this case in nowadays kernels as well, because not all block devices support TCQ.

But when the backing device supports TCQ we can express DRBD's write barriers as Linux's write barriers, therefore offloading the whole logic to the backing storage device. The motivation behind this is of course to give the backing storage device more freedom in optimising the data transfer.

For each DRBD-type write barrier we have to issue an ordered write request on the receiving side. As an optimisation we coalesce two DRBD-type write barriers that enclose a single write data block into a single ordered write request.

The Algorithm

- When we receive a barrier packet
 - If we have no local pending requests, we send the barrier ACK immediately.
 - If the last write was an ordered write, mark it. When the IO completion handler finds that mark, it sends the barrier ACK after the data block ACK.
 - If we have local pending requests, we set a flag that the next data packet has to be written with an order tag.
- When receiving data packets, we need to check if we should do an ordered write now.
- When a write completes and it was an ordered write, also send the barrier ACKs (either only one in front, or one in front and one after the data block ACK).

References

- [Rei01] Philipp Reisner. *DRBD*. Proceedings of UNIX en High Availability May 2001, Ede. Page 93 - 104.
- [Rei04] Philipp Reisner. *Rapid resynchronization for replicated storage – Activity-logging for DRBD*. Proceedings of UK UNIX USERS GROUP's winter conference 2004, Bournemouth.

Resources

Papers, documentation and the software are available online at <http://www.drbd.org/>.