DRBD 8.0.x and beyond
Shared-Disk semantics on a Shared-Nothing Cluster

Lars Ellenberg

August 10, 2007

Abstract

So, you have an HA-Cluster. Fine. What about when your storage goes down?

DRBD – the Distributed Replicated Block Device, as developed mainly by Philipp Reisner and Lars Ellenberg and their team at LINBIT `http://www.linbit.com` – makes your data highly available, even in case a storage node goes down completely. No special requirements, off-the-shelf hardware and some standard IP-connectivity is just fine.

I'll outline the problems we try to solve, how we deal with it in DRBD, and why we deal with it the way we do, explaining the design ideas of some algorithms we have developed for DRBD, which I think may be useful for software raid or snapshots as well (dm-mirror and friends), or in cluster file systems or distributed databases.

There is an overview of typical and not-so-typical usage scenarios, and the current limitations, and some hints about how you can (ab)use DRBD to your advantage, e.g. when upgrading your hardware.

There is also mention of benchmarks, as well as upcoming features, like cache-warming on the receiving side, improved handling of cluster-partition/rejoin scenarios with multiple-primaries (interesting for cluster file systems), or the (as yet only conceptual) possibility to scale-out to many (simultaneously active) nodes (N > 2).

1 What is DRBD

DRBD is the Distributed Replicated Block Device, implemented as a Linux kernel module.

Block device something like `/dev/sda`; something you can put a file system on

Replicated any local changes are copied to additional nodes in real time to increase data availability

Distributed reduce the risk of catastrophic data loss, spread the data to storage nodes in different locations

The purpose of DRBD is to keep your data available (business continuity), even if your storage server fails completely (high-availability fail-over clustering), or if your main site is flooded or otherwise catastrophically destroyed (disaster recovery). DRBD is not to be confused with grid-storage.

2 What you should know

You should be able to tell the difference of file system and block device, write() and fsync(), write ordering, dependencies and reorder domains, write-back and write-through, cache, coherency and corruption, up-to-date, clean and consistent, switch-over and fail-over, right and wrong, black and orange, blues and jazz, ... [some of this is briefly explained below, anyways].

When I say "cluster" I usually mean "High Availability Cluster": several distinct computers (nodes) acting together to provide maximum reliability and uptime for the services in question.

There also are compute (high performance) clusters, load-balancing clusters, ... Most cluster have at least some aspects of the other cluster varieties, too. Classification and taxonomy is more about the primary focus, the main operational directive, of the cluster.

DRBD only provides the infrastructure, to actually do HA-clustering you need a cluster manager. Heartbeat is the cluster manager most commonly used together with DRBD. If you are not familiar with some HA-clustering vocabulary, e.g. fail-over, switch-over, fencing, STONITH, SPOF, split-brain, ..., their project page `http://www.linux-ha.org` might be a good starting point.

There are a number of other cluster managers that work just fine with drbd, too, including the RedHat Cluster Suit, as well as "home grown", tailored solutions at telcos or storage providers.

3 The Problem: Data Availability

Regardless of how you deploy your cluster, and regardless of what services you provide, if the cluster manager decides that some cluster node shall take over the services formerly provided by some other cluster node, it needs access to the data corresponding to the service, including most recent changes.

The obvious solution is to have the data not stored on the cluster nodes – where the node failure makes it inaccessible defeating the purpose of having the cluster in the first place – but at a central location accessible from all cluster nodes. This may be a remote database, NAS (e.g. NFS-server), SAN (storage box with shared SCSI, Fiber Channel, iSCSI, the various Linux "network block devices" nbd and successors, there is even "ATA over Ethernet", ...), or something similar. Such storage devices (disk) accessible from (shared by) more than one node, are commonly called “shared disk”.

But this does not really solve the problem, as long as this central data location (shared disk) is in itself a Single Point of Failure.

We need to get the storage highly available, too.

3.1 RAID, rsync and Backups

To guard against *hard-disk failure*, you can use backups.

In case you ever really need a full restore from backups, you obviously have lost the changes since your most recent backup. This is better than nothing, but it certainly is not good enough in many cases.

So you use RAID, with mirroring (RAID-1) or checksumming (RAID-5, RAID-6, etc.) to protect against single disk-failures. Certain RAID levels even survive multiple simultaneous failures.

(You still need the backups to guard against the "accidental `rm -rf /`").

To guard against *storage-node failure*, you may replicate the data to an other node, not unlike RAID-1 is mirroring changes to two or more disks. Each node itself should then have a local RAID, which in turn should be composed of reliably hardware...

Some suggested to use scheduled synchronization (aka rsync cronjobs) or file-system event driven synchronization (project idea: combine FUSE with rsync/librsync or csync2, triggering a sync on certain file operations). Because its inherently asynchronous nature, again, once you need the mirror, you lost the changes done since the most recent sync.

While this is nice for infrequently changed directory hierarchies, or for a staging, hardlink-based on-disk backup, it does not work too good for file/news/mail servers. Much less so for databases.

While some databases have their own (asynchronous? synchronous?) replication framework, or can be abstracted to multiple mirrored instances by some middleware, we seek a more generic solution to the data availability problem.

4 Synchronous Data Replication

We chose to *replicate* WRITEs at the block device layer as generic solution to cope with a storage-node failure. DRBD is implemented as a stacked block device driver[1].

To avoid losing the most recent changes in case the failing node was active (=currently writing to the storage), we have to replicate synchronously, in real time, to the other mirror nodes[2].

[1]Before you ask "why is DRBD an out-of-tree kernel module?" At the time of writing this (July 2007), there are efforts underway to get DRBD included into mainline. If I find the time to “beautify” our code base sufficiently in time, we might get into mainline later this year.

[2]hahum. OK, this should read "the other node". Singular :(. But see also section 13

We have to defer completion events for WRITE requests and wait for all[3] mirror nodes to complete the request, before we may relay completion to upper layers (file system, vm, database using drbd as "raw-device" with direct-io, . . .).

Since by replicating the data, each node has its own data set, they do not share any hardware: this is called a “shared-nothing” cluster.

5 Sounds Trivial?

So we basically “only” need to ship READs directly to the local block layer, and ship WRITEs to local storage as well as over TCP to the other node. Which will send back an ACK once it is done.

To get this working is almost trivial, right?

Well, yes. But to *keep* it working in face of component failures is slightly more involved. In the following, all-capital "DRBD" shall denote the driver and the concept, "drbd" means a single virtual block device instance.

"Primary" in DRBD-speak means a node that accepts IO-requests from upper layers; as opposed to

"Secondary", which will refuse all IO-requests from upper layers, and only submit process requests as received from a Primary node. We may change the terminology to Active resp. Standby/Passive someday.

A healthy drbd consists of (at least) two nodes, each with local storage, and established network connection.

The network link may become disconnected, disks may throw IO-errors, nodes may crash... We survive any single failure, and try to handle most common multiple failures as good as possible.

We guarantee that in a healthy drbd, while no WRITE-IO is in-flight, the DRBD-controlled local backend storages of the nodes are exactly bit-wise identical.

If we lose connectivity, or get local IO-errors, we can no longer do this: we become "unhealthy", degraded. To become healthy again, we need to make the various disks contents identical again. This is called resync, and has interesting problems in itself, which are detailed in the section 6.

When a Secondary node detects IO-error on its local storage, it will notify the other nodes about this and detach from its local storage. No harm done yet, we are merely degraded. Operator gets paged, disk gets replaced, resync happens.

When we lose connectivity, DRBD will not be able to tell if it is a network problem or a node-crash. Enter the cluster manager, which should have multiple communication channels, and the smarts to decide whether the Primary may keep going, or the Secondary needs to be promoted to Primary and start up services, because the former Primary is dead, really. In any case, data is still there, we wait for the connection to be established again, and resync.

When a Primary node detects IO-error on its local storage, it will notify the other nodes and detach from its local storage. Failing READs will be retried via the network, new READs will now have to be served via the network by some other node. WRITEs have been mirrored anyways.

Still no harm done, upper layers on the Primary won’t notice the IO-error.

But at the nearest convenient opportunity, services should be migrated to an other node (switch-over). Because if you don’t, Murphy has it that the network will soon get flaky as well, we won’t have access to good data any longer, so we must fail any new IO-request coming in. And to have the services available we’d have to do a fail-over, anyways. Which will happen at the most inconvenient time, obviously. . .

6 resync: magic healing

Now, lets assume we are degraded. How to become healthy again?

Once all nodes can communicate (again) and have good disks (again), we need to resync: we have to retransmit any blocks which (might) differ meanwhile.

To keep resync times down, We try to retransmit only the blocks which in fact *are* different. But data integrity first, rather transmit a couple gigabytes too much than one bit too few.

[3]again, see section 13: "all" may become "number of nodes necessary for a certain quorum"; also see section 10.1

Two questions to be answered:

- which blocks, and
- which direction?

6.1 network hiccup

Lets keep it simple: One Primary, one Secondary, it has been just a network hick up, no node-role changes involved. We know the direction: Primary -> Secondary. We know which blocks to transfer, because the Primary keeps an in-memory bitmap, which is dirtied whenever a WRITEs is completed to the upper layers without being acknowledged by the Secondary.

This may transfer some blocks more than necessary, because of the granularity of the bitmap, and because for some blocks only the ack was lost, but the data had been written correctly.

6.2 node crash

If a Secondary node had crashed and was revived, the procedure is just the same as above.

If the Primary was rebooted while the Secondary was down, we'd lose the information stored in the dirty bitmap, so we do keep a copy of it in some reserved meta-data area on disk, where we can initialize the in-memory bitmap from, once we are configured again.

If a Primary node had crashed, we have a different problem.

There could have been in-flight io, and we have no idea whether that made it to disk or to the network, or to both. Even though only very few blocks will be different, we have no idea which ones, we have to assume that any block might be different.

For the sake of data-integrity, we would have to retransmit the entire disk, just to be sure...

6.3 Full Sync? No Way.

To avoid this, we could dirty the on-disk bitmap with each incoming write request, submit the write, and clear it after it has been successfully completed.

This would make three requests out of one. Worse, to be correct, the dirty write would have to be synchronous, we'd have to wait for it to complete before we could submit the application write.

We are smarter than that.

6.4 Peanuts . . .

To reliably keep track of the target blocks of in-flight IO, while minimizing the required additional io-requests for this housekeeping, we came up with the concept of the "Activity-Log".

Think of your storage as a huge heap of peanuts. Sisyphus has tagged them all with a distinct block number. There are many people running around, taking some of the peanuts in their pockets (that is the in-flight io), and throwing them back on the heap (that is the io-completion). Painting them blue is allowed, these are WRITEs we are missing the acknowledgment of the other node for (dirty bits). Eating peanuts is strictly forbidden, as is re-tagging.[4]

Blocks corresponding to the in-pocket peanuts have to be retransmitted, those corresponding to the heap don't need to (but it would do no harm if some of them are).

Our mission is to know at each given moment as precisely as possible which peanuts are NOT in the pockets of those people (and not painted blue, yet), because if we know that, we can avoid retransmitting the corresponding blocks after Primary crash.

First, we get into control of the situation. We structure the heap, and put the peanuts in order into boxes (activity-log extents) which in turn are numbered. We draw a line in the sand.

[4]Some do that, anyways; call them Eh-i-oh and Silent Corruption ;)

We prepare a number of parking lots on one side, and get ourselves as many little red wagons (activity-log slots). People cannot reach the peanuts on the other side of the line. Only we are allowed to move the wagons from the other side into the parking lots on this side. People are free to take from this side of the line (the in-memory activity-log).

Now we know that everything not in the activity-log is stable, and does not need to be retransmitted - apart from the blue ones, the tags of which we jotted down (to the on-disk bitmap) whenever we dropped an extend out of the activity-log.

To further structure this, we line up the extents in the activity-log in three LRUs: those boxes where some peanuts are missing (the active lru, currently target of in-flight io), those which are complete (the unused lru, zero in-flight io), and those wagons with no boxes at all (the free list).

We start with an empty activity-log.

Now, if someone wants a peanut from a special box which is not on display yet, we have some work to do. We check whether there is an empty wagon, if yes, we go, fetch the box, and write the box number down (to the on-disk activity-log). If there is no slot available on the free list, we check the unused lru. If there is an unused slot, we exchange its content with the requested one, and write both numbers down (to the on-disk activity-log).

If there is neither an empty nor an unused slot available (activity-log is starving), we have to be rude, and block further WRITE requests until enough of the outstanding requests have completed and one slot becomes unused again, which we then will take and exchange, again writing both numbers down (you know where).

When someone comes by and want some of the boxes on display already, we just move them around to keep the lru order, no further action required. This is the caching effect of the activity-log.

For performance reasons (write latency) you want to avoid to block, so you want to have a reasonable number of slots to make sure there is always some unused ones. But you do not want to have a huge number of slots either, because on Primary crash, that is the area we need to retransmit, because that is the granularity with which we know what *might* have been changed.

Whenever we need to “write down the number", we have to do so transactional, synchronously, before we start to process the corresponding WRITE that triggered this meta-data transaction. Since we may crash during such a transaction, we write to an on-disk ring buffer to avoid corrupting the on-disk representation. With each such transaction, we cyclically write down a partial list of unchanged members as well. We dimension the number of slots and the size of the on-disk ring buffer appropriately, so we can always restore the exact set of extents which have been in the activity-log at the time of crash. When a drbd is attached to its backing storage, it detects whether it has been cleanly shut down. If it determines it had been a crashed Primary, it will set all the bits corresponding to the extents recorded in the activity log as dirty.

The number of slots in the activity-log is tunable, the trade-of is larger activity-log, less frequent meta data transactions, less likely to introduce maximum latency because of activity-log starvation, but also longer minimum resync time after Primary crash.

6.5 . . . aka Activity-Log

We just managed to completely decouple the total storage size (number of peanuts) and the amount of unavoidable retransmission after Primary crash, making minimum resync time a configurable constant, independent [O(1)] of the amount of total storage. The Activity-Log concept could be applied to a number of other problems as well. Software raid rebuild is an obvious example. Also the duration of an fsck could probably be trimmed down this way, journalling file system or not.

In fact, any problem where you need to keep track of the changes to a huge set of items, but do *not* want to throttle performance with O(N) housekeeping or worse, is a good candidate.

Now we answered the first question: which blocks (not) to retransmit. During the connection handshake, DRBD transfers the bitmaps, bit-or them together, and then retransmit all corresponding blocks in order.

But wait... In which direction?

7 resync direction: You or Me?

To determine the direction of the resync, we tag data generations, similar maybe how a version control system might tag certain "project generations". (Currently we tag with "randomly generated UUIDs", but that may change.)

Whenever a node modifies its data without being able to communicate the modifications to its peer(s), it first generates a new data generation ID tag.

We keep a short history of such generation IDs, so we have a "current", a "bitmap", and a few "history#" IDs (`[C:B:H1:H2]`). These ID tuples will be stored together with a few status flags in the on-disk meta-data, too. During connection handshake, we compare these ID tuples, and determine which node has the "best" data, and thus will become SyncSource.

A node which is currently Primary may not become SyncTarget, because that would cause cache coherency issues and probably data corruption. It will try to become Secondary, and if that is not possible because it is currently in use, call a policy handler in user space, or disconnect (go “StandAlone”).

Also, during resync, the sync target will be inconsistent, because it does not receive the data in the original order, but as a linear sweep over all the determined to be retransmitted blocks.

If we assume that during resync the chosen SyncSource is the only UpToDate copy of your data, and it happens to crash before the resync is done, you are worse of than if you had not started the resync, when you’d have at least a consistent, while out-dated, copy of the data. To avoid this, you could take a snapshot on the SyncTarget just before you start to resync.

When an Active node lost connectivity, or a Passive node is made Active while communication is down, it will copy its current ID to the bitmap ID (unless that is *already* set), and generate a new data generation ID. Let me visualize the algorithm:

```
  Primary [X;0;A;B] <-----> [X;0;A;B] Secondary
```

communication down – Primary re-tags, and continues writing...

```
          [Y;X;A;B]    |    [X;0;A;B]
```

communication up – X is direct ancestor of Y (version control equivalent: fast-forward).

Because we are going to modify it now (we start the resync), we re-tag it as a new data generation during resync. And we mark the SyncTarget as Inconsistent.

```
          [Y;z;A;B] >>>>>>> [z;0;A;B]
```

If we lose communication during this resync, we just start over; content and IDs changed, but not the overall pattern. resync finished We reset the bitmap ID, rotating its value out into the history. The SyncTarget adopts the current ID of the SyncSource, and sets itself consistent again.

```
          [Y;0;z;A] <-----> [Y;0;z;A]
```

Similar for other cases. This way we reliably detect differing data sets with or without common ancestry, too: `[G;X;A;B] <???> [K;X;A;B]`; split-brain; version control system equivalent: manual merge required.

We offer a number of auto-recovery strategies when split-brain is detected, but the default policy is to give up and disconnect, and let the operator sort out how the merge shall be done, or which change-set to throw away.

A version control system would assist you with a three-way-merge. Unfortunately, we cannot. Because we don’t have the parent data set anymore, to generate the necessary deltas against. And even if we had, there simply is no generic three-way-binary-merge.

Unlike with a "real" shared disk, a split-brain situation with DRBD does not lead to data-corruption.

With a shared disk, if there is no (or malfunctioning) arbitration, once multiple nodes write uncoordinated to the same disk, you can reach out for your backups, because that disk is now scrambled.

With DRBD, because it is shared-nothing, each node has exclusive access to its own storage. When they don't communicate, the end result is "just" diverging data sets, each consistent in itself.

This may be even worse than a scrambled disk, though, since now you possibly spend considerable time trying to merge them, before you get frustrated and reach out for your backups, to start over, anyways...

8 Cluster File Systems: You and Me!

But what about these "shared disk semantics"? I want my drbd to be scrambled as well...

Besides providing central storage (shared disk) for fail-over clustering with normal file systems and exclusive access of at most one node at any given time, shared disks have an other interesting use case: If some nodes access a shared disk in a coordinated fashion, they can use it at the same time, sharing data, possibly for load-balancing applications, or producer-consumer like data mining setups.

Cluster file systems like OCFS2 or GFS2 already do the necessary coordination (using a distributed lock manager, DLM) to deal with cache coherency and avoid scrambling the disk.

We want to use cluster file systems with DRBD, too.

Since the cluster file systems (should) do all the necessary coordination already, we just need to allow more than one Primary, and be done with it, right? What if the user then "accidentally" uses a standard, non-cluster file system? Despite the fact that the now scrambled content will be thrown away, anyways, since DRBD is *replicated*, not *shared*, DRBD still has to make sure that the content is scrambled on all disks in the same way!

Lets assume two nodes write to the same offset at the same time (conflicting writes). Lets further assume that both nodes finish their own write before receiving and writing the other nodes data. We end up with diverging data sets, each node with the foreign data. This is further illustrated below.

8.1 Concurrent Write-Access arbitration

We need to make sure that both copies are always[5] 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 not[6], 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.

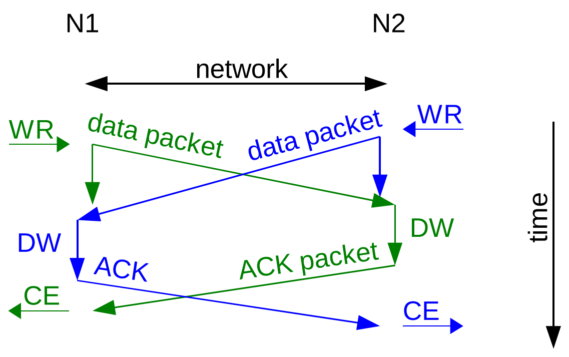

Figure 1: uncoordinated writes;
WR: write request; DW: disk write; CE: completion event

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

[6]Of 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...

8.2 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 peer's 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 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).

8.3 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).

8.4 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[7]. Although this case is unlikely, we have to be prepared. We attach sequence numbers to the packets. See figure 2.

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

8.5 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, pretending it had

[7]We have two tcp-sockets per drbd replication link: the data-socket for streaming of replication data, and the meta-data socket to communicate out-of-band state information, avoiding the additional latency such communication would suffer on a congested data socket. Since we have it, we use it for ACK packets as well. We don't care for ACK packets overtaking data packets, but as explained, we have reason to ensure that processing of DATA packets does only happen after processing of previously sent ACK packets.

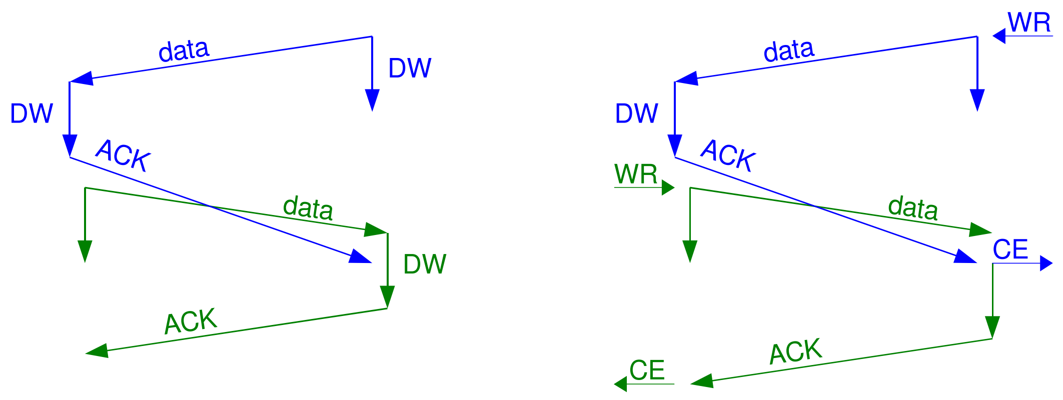

Figure 2: A data packet overtakes an ACK packet; 126 of 4032

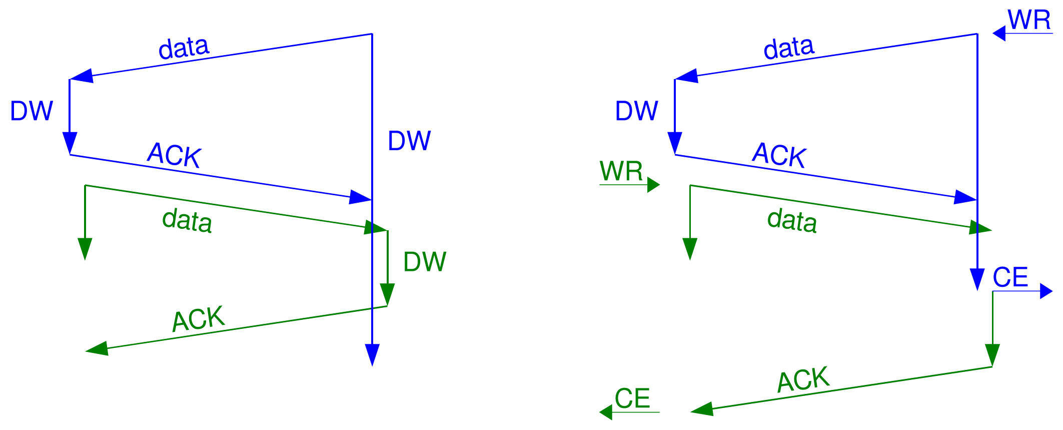

Figure 3: High disk latency; 210 of 4032

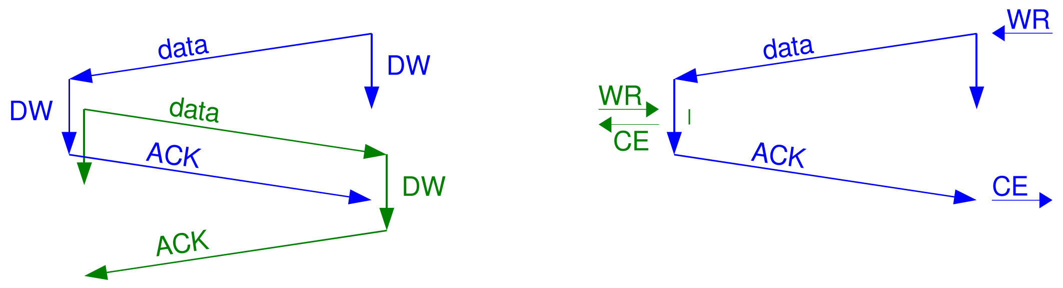

Figure 4: Concurrent write while writing to the backing storage; 1080 of 4032

been written successfully. See figure 4.

We also discard local requests, if we detect that an other local request to the same area is still on-the-fly, since we could not guarantee the write ordering to be the same on both nodes. Even on the local IO-stack only, the result of such behavior would be “undefined”.

8.6 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

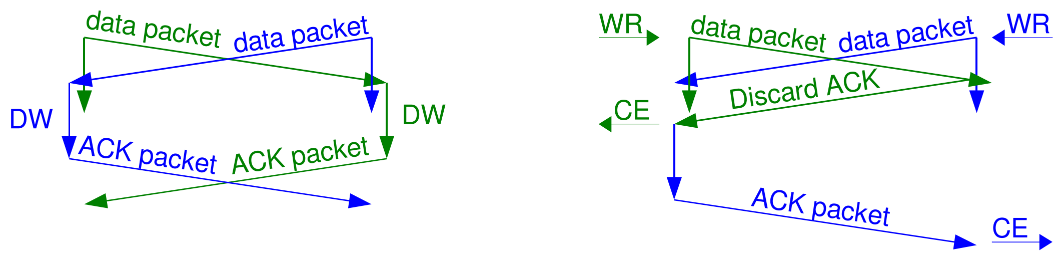

Figure 5: Concurrent write with low network latency

As illustrated, in the end each node would end up with the respective other node’s data. We flag one node (in the example N2) with the discard-concurrent-writes-flag. Now both nodes end up with N2’s data. See figure 5.

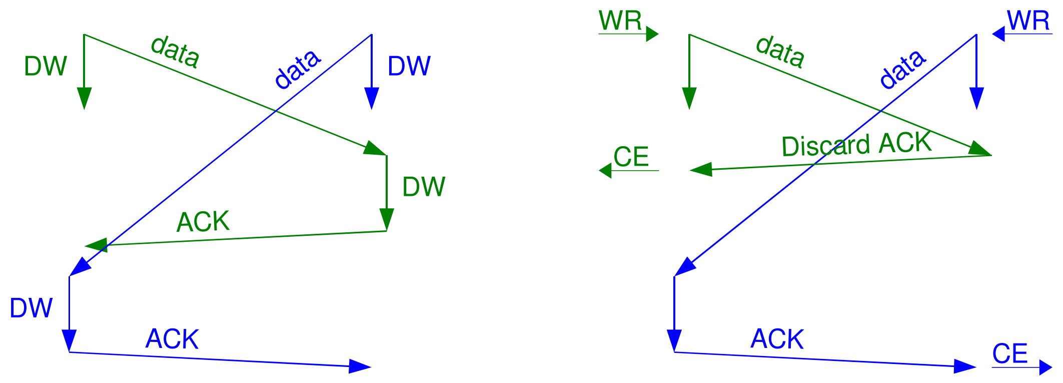

Figure 6: Concurrent write with high network latency

Concurrent writes, high latency for data packets This case is also handled by the just introduced flag. See figure 6.

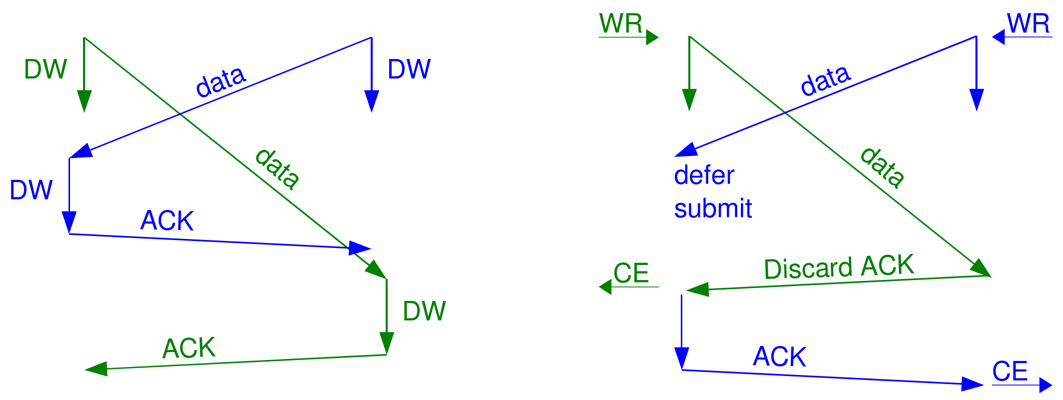

Figure 7: Concurrent write with high network latency

Concurrent writes with high latency for data packets

Given unfortunate timing, N2 may not be able to detect that this was a concurrent write, since it got the ACK for its own request, before the conflicting data packet comes in. We defer the local submission of the received data, until we see the DiscardACK from the peer. See figure 7.

8.7 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

- If the sequence number of the data packet is higher than last_seq+1, sleep until last_seq+1 = seq_num(data packet) [see 8.4 and figure 2].
- Do I have a concurrent request, i.e. do I have a request to the same block in my transfer hash tables? If no concurrent request was detected: write now.
  Otherwise, if a concurrent request was detected:
  - if I have the "discard-concurrent-write-flag":
    discard the data packet, and send a DiscardACK back to the peer.
  - if I do *not* have the "discard-concurrent-write-flag": defer local submission of this data packet. Wait for local io-completion [see 8.4 and figure 3] and until any pending ACK (respective DiscardACK) has been received [see 8.6, 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 8.5 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!

9 Avoiding data diversion (resource-level split-brain)

To avoid data corruption with a real shared disk, it would be mandatory to configure STONITH, i.e. whenever a node starts to access the shared disk, without being able to communicate (coordinate) with the other node(s), it would need to use brute force to make sure any presumably dead node is in fact dead ("node-level fencing").

With DRBD, you have an additional failure scenario over shared disks:
Assume DRBD's replication link was broken, while the cluster manager can still communicate via a serial cable. Not knowing the internal state of DRBD (which can be described as resource-level split brain in this case), the cluster manager might happily migrate the services, including the DRBD Primary role from one node to the other node in the cluster. This of course would cause data diversion.

To avoid this, we can do "resource-level fencing": when we lose connection, we call out to a user-space helper, which can use alternative communication channels to mark the non-active data set as outdated (which is also stored in persistent meta-data). 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. When a degraded cluster node wants to become primary, it needs to first make sure that its peer node is marked as outdated.

One possible implementation is the "drbd outdate peer daemon" (dopd) in the heartbeat package, which uses heartbeat's communication layers.

This is an orthogonal thing to STONITH: a different problem, a different (less intrusive) solution.

10 performance, tunables and benchmarks

There are many tunables in DRBD, which are configured with the rest of drbd in `/etc/drbd.conf`. Some have impact on "everyday performance", some reflect tradeoffs between different preferences, some settings are conservative to limit resource consumption on the "average" system, and some may only have impact in specific situations or setups.

If you want to increase overall performance with DRBD, you should first find the bottleneck. Often you get more out of tuning your network and local IO-stacks first (e.g. chose io-scheduler settings that suit your needs!), before you start fiddling with DRBDs settings.

10.1 DRBD-wire-protocols, Latency, Fail-over and Write-Ordering

Since DRBD does synchronous replication, once you add DRBD to your IO-stack, you will experience performance degradation, resulting from the DRBD-internal housekeeping, as well as from the additional network IO and remote IO stack. Obviously the resulting maximum sustainable bandwidth is the minimum of the bandwidth of local and remote IO-subsystem and bandwidth of the replication link. The resulting worst-case latency is the sum of local, network and remote latency plus any latency introduced by DRBD-internal housekeeping meta-data transactions.

For performance reasons (mainly latency), you can chose to do (slightly) *asynchronous* replication with DRBD as well. We allow for

protocol A asynchronous; completion of request happens as soon as it is completed locally and handed over to the local network stack. In case of a Primary node crash and fail-over to the other node, all WRITEs that have not yet reached the other node are lost.

protocol B quasi-synchronous; completion of request happens after local completion and remote RecvAck. Rationale: by the time the RecvAck has made its way to the Primary node, the data has probably reached the remote disk, too. Even in case of a Primary crash and fail-over to the other node, no data has been lost. For any completed WRITEs to be actually lost, we'd need to have a *simultaneous crash* of both nodes after RecvAck has been already received on the Primary, but before the data reached the disk of the Secondary – *and* the Primary has to be *irreparably* damaged, so the only remaining copy would be the Secondary's disk, which lacks these WRITEs. Not impossible, but unlikely.

protocol C synchronous; completion of request only happens after *both* local completion *and* remote WriteAck.

When the former Secondary takes over after a Primary crash and fail-over, its data is *consistent*. But it may not be *clean*. To become clean and consistent, a file system would need to do a journal-replay (or fsck), a data base would need to additionally replay transaction logs etc., so to the applications and services it would just look like an extremely fast reboot.

Now, actually, it is not as simple as that.

Since we do replication, there are two io layers involved, there are two io-schedulers, two disk subsystems that may reorder WRITEs, and may chose to order differently. To avoid that, we could do strictly synchronous replication, only. But we can also define reorder-domains, and separate them with in-protocol barriers.

Any WRITE that is submitted before a previously submitted WRITE has been completed may be reordered. Every time we complete a request to upper layers, we create a new "epoch" in the "transfer log" (a ring-list structure on the Primary), which then contain all requests submitted between two application-visible completion-events. On the wire, different epochs are separated by in-protocol DRBD-barrier packets inserted into the data stream. Once such a barrier is received, the receiving side waits for all pending IO-requests to complete before sending a BarrierAck back, and submitting the next one. This could be improved upon using tagged command queuing (TCQ), which would avoid the additional latency introduced by waiting for pending IO.

To still be able to reliably track (possibly) changed blocks, even with the asynchronous protocols, and mark them for resynchronization if necessary, completion of a request happens in two phases. First it gets completed to the upper layers. Then, when the corresponding epoch is closed by the receipt of a BarrierAck, it gets cleared from the transfer-log (in protocol C, closing an epoch is almost a no-op). When we lose connection, we scan the transfer log for any entries in not-yet closed epochs, and mark the corresponding bits as dirty, even when the corresponding application requests have been completed already.

10.2 limit resource consumption

To limit the impact of resynchronization on application IO, we throttle the resync-rate. This has no effect on "everyday performance", though.

To limit memory consumption on the receiving side, we allow only a certain number (`max-buffers`) of in-flight io-requests. If your IO-subsystem takes a huge number of requests in-flight, you want to increase this setting.

Similar, to limit housekeeping structures on the sending side, we only allow `max-epoch-size` entries, before we close an epoch, regardless of application-visible completion. Since this currently introduces a "flush and wait" on the receiving side, this may have a performance penalty. Again, if your IO-subsystem is happy with a huge number of in-flight requests, increase this.

For good TCP performance, you may want to increase the size of your tcp-window and other related settings (`sys.net.ipv4.tcp_[rw]mem` and friends). You may also want to increase the drbd data-socket send-buffer (`sndbuf-size` in `drbd.conf`). You can verify whether this is necessary by doing something like

```
watch -n1 'cat /proc/drbd; netstat -tnp | grep :7788 | grep ESTABLISHED'
```

during sequential writes, and see whether it ever runs against the socket buffer limits.

Some IO-subsystems like to be kicked (unplugged) often, some like to really only decide themselves. You can tune this using "`unplug-watermark`", which tells the receiving side to unplug its lower level device whenever it has more than thus many in-flight requests (yes, there is some sort of hysteresis in place).

10.3 find the bottleneck: systematically benchmarking

If this still did not help, maybe you should start over. When hunting down bottlenecks, we like to benchmark linear WRITEs. There is a small and stupid tool named "dm" (I don't remember why dm; has nothing to do with device mapper) in the DRBD source tree, which we use for this purpose[8]. First, get a scratch partition or LV, then benchmark local io throughput, using direct-io, and writing through vm. Next, put a file system on top of that, and benchmark writing to a file in the file system. Next, put DRBD on top of the partition, again first write directly to the (as yet unconnected) drbd, then into a file on a file system on drbd. Do the same on the other nodes. Then connect DRBD on the nodes, and wait for the resync. Maybe you want to also use netcat and dm, to pipe reads from one node through the network to writes on the other node, thus generating a load very similar to that of DRBD-resynchronization. Watch the network buffers. Flood-ping at the same time, and watch the ping times (and packet loss?) increase during load.

For some real-world production system data see figure 8; the 1GBit replication link is maxed out, and is obviously the bottleneck here.

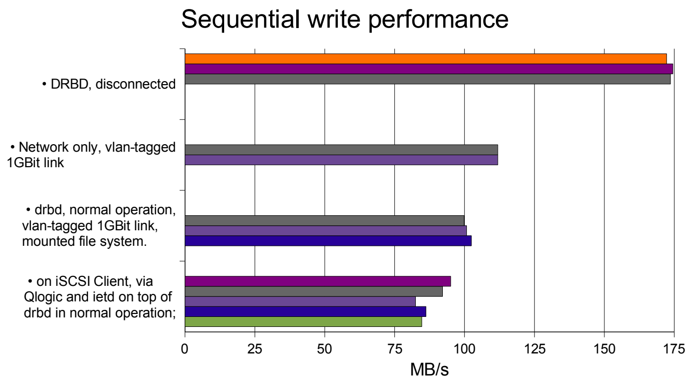

Figure 8: performance data produced by multiple runs of
`dm -a 0 -o /mnt/somwhere/big_file -s 1G -b 1M -y -m -p`
or, for network only:
Sink: `netcat -l -p 1234 > /dev/null`
Source: `dm -a 0 -o >(netcat 192.168.1.2 1234) -s 1G -b 1M -m -p`

You tried everything to get track your bottlenecks and tune the system to get decent performance out of it, but still no luck? Ask for help on our mailing list, or get a support contract…

11 typical setups

11.1 Active/Standby

You have one drbd, containing a file-system, backing the data area of some service (web/mail/news...). You can also have several of such resource groups, but you usually have all active on one node, the other acting as hot-standby only, ready to take over in case of Primary crash.

[8]You are free to use whatever tool you like, but don't forget about fsync!

11.2 Active/Standby + Standby/Active

If you have several drbd and related resource groups, you may also have some active on each node. This balances the average utilization on the available hardware better than the strict active/standby setup. But it may disguise resource shortage for too long. Keep in mind that on node-failure, still one node must be capable to handle all resources. You may have started with a lower utilization, but if your average utilization during normal operation is 60% on both nodes (which you easily can overlook: both still seem to be idle “most” of the time), then after a node crash, you have one overloaded box left, which you probably just sent thrashing. Not exactly the purpose of an HA-cluster.

11.3 Active/Active with cluster file system

This is a very interesting thing to do, even though there are many caveats (see section 13). There are not too much services that can make use of this: two instances of postfix cannot share the same spool directory on one node – why should they be able to share it when running on two nodes?

But you could have postfix deliver to maildirs on one node, and serve them via IMAP on the other node. Or use the Samba cluster. Or use as backend for XEN domain live-migration. Or for load-balanced multipath iSCSI. I expect much more interesting possibilities here to be discovered still.

Again, keep an eye on the average utilization: you should be able to still run both on one node, though maybe with poor performance during peak periods.

11.4 Disaster-Recovery

You could also simply not use a cluster manager, and have a remote site be strictly Secondary, mirroring all changes from the main site. If the main site is catastrophically destroyed, you can move the box from the DR-site physically into a new site, and make it Primary there.

11.5 … + Disaster-Recovery

Of course you could combine any of the above two-node clusters with a third node for disaster recovery.

11.6 DRBD on ramdisk

Huh? Why would you want to synchronously mirror a ramdisk?

This may be your apache/tomcat/whatever session store. You get rid of the performance penalty for persistent on-disk session store, but you still have the session data “persistent”, surviving a server crash: the other node has it in its ramdisk, too, and just takes over. Ok, if both servers go down at the same time, you lost the session data. But then you have other things to worry about.

12 and what you can do with it

Besides using DRBD and high-availability clustering for its primary purpose, namely to protect you against node failure, once you established this shared-nothing HA cluster, there are many additional goodies that come along. Here are just a few examples. Remember that you get all this even out of your off-the-shelf hardware, no special system requirements at all. Though definitely some enterprise-class storage backend won’t hurt.

12.1 add RAM or storage capacity “at runtime”

Your strategic monitoring indicates that you will run out of RAM or disk space soonish?

Using DRBD and switchover, you can do hardware maintenance/upgrade without downtime.

Take your secondary down, upgrade the hardware, boot, test, resync. Switch-over, wait a few seconds for the services to stabilize and any clients to finish their reconnect logic if necessary. Upgrade hardware on the other node, boot, test, resync. Added storage? Tell DRBD about it (`drbdadm resize`), then tell the file system (`xfs_growfs`/`resize2fs`).

Virtually no downtime, and your database is running with doubled RAM, your backup server has tripled its available capacity, and no user has even noticed...

12.2 "online" kernel or firmware upgrade

Similarly, you can upgrade vital system components like kernel, bios or firmware, or install high-priority software upgrades for your services without user-visible downtime, by upgrading in stages, and switch-over half way through.

12.3 reports or backups on the Secondary

If the lower level storage of DRBD is a logic volume (LV), you can do a snapshot on the Secondary, and mount it.

This way you can access (a snapshot of) your data on the Secondary, regardless of used file system. You then can drive database reports or backups from there, without affecting the performance of the Primary (well, almost; the additional write latency introduced by the copy-on-write for the snapshot will propagate back).

12.4 "online" database scale-out

Say you really hit with a good product, and your database backends outgrow the initially estimated capacity quickly. Your database design would allow you to split databases to different backends, increasing the number of backends and the overall system throughput. With DRBD, you could split the cluster, forcefully introduce what normally would be considered "split-brain" and make both Primary, drop different halves of your dataset on the respective nodes, add two new boxes as new Secondaries, ending up with two new clusters. Repeat as necessary.

Similarly for file servers.

If well prepared, this is scale-out without downtime.

13 current limitations, future prospects

Many Nodes

The most inconvenient limitations is currently that DRBD supports only two nodes natively. You can stack DRBD on top of DRBD, but to maintain such a setup gets cumbersome very quickly[9].

We will develop a multi-node DRBD, with configurable symmetry. To deal with the increased number of possible component failures and global failure scenarios, we have to design and implement a better state engine, and have to rework our meta-data (activity-log and persistent dirty bitmap) considerably. The good news is that the conceptual work is done, what is left is mostly straight forward work, though time consuming. Even the algorithm for lock-less concurrent write arbitration can be generalized for $n > 2$ with few modifications. For "read-mostly" applications, the combination of many-node DRBD and cluster file-systems is (will be) the killer scale-out solution: during normal operation, all reads are local, and will be able to saturate the respective "n" local IO-stacks.

[9]Among other improvements – which are folded back to the GPL'ed DRBD irregularly whenever we have an other such improvement as new selling point –, the commercial variant (DRBD+) makes three-node-setups slightly less cumbersome. But effectively you pay for the support to let people who do it every day deal with the details and hide them from you...

Also, think of this as your chance to support the further development of DRBD, and to influence feature development and priorisation.

DRBD-Proxy, deferred asynchronous replication, consistent resynchronization

As explained above, during resynchronization, the SyncTarget is inconsistent. This is undesirable, especially in disaster-recovery setups. What is more, for disaster-recovery setups you want a geographically dispersed storage, which may imply high latencies and low bandwidth.

We can overcome both limitations with what currently has the working title "DRBD-proxy". We will introduce a user-space proxy for the DRBD protocol, which can do large-scale buffering, transfer-compression and encryption of the DRBD-payload. The buffering can obviously not increase the bandwidth for streaming data, but will smoothen out any peaks for random IO-load to get the mid-term *average* throughput below the physically available network bandwidth. An additional benefit is, that we could use the buffer to consistently replay the buffered changes during resync, provided the changes still fit into the available buffer space. If we keep the buffer available for as long as possible even after we mirrored its content, we could potentially even use it to "rewind" the block device (or parts thereof), guarding against said "accidental `rm -rf /`", as long as that is detected in time.

To not lose the High-Availability properties of DRBD, it should be combined with a local "conventional" DRBD cluster.

This will be implemented in several phases. A first production-ready prototype will only do in-memory buffering and compression, and is expected to be available early 2008 (or much sooner, depending on user demand, and how we prioritize development effort).

Write-Quorum

In the Two-Primaries case, currently whenever DRBD loses the connection, you get effectively a split-brain situation immediately, with slightly diverging data. Which then needs to be resolved by "discarding" the changes on one of the nodes, possibly using the provided auto-recovery strategies.

Since this is annoying big time, one of the next features will be the implementation of an (optional) Write-Quorum, so that starting with a *Write-Quorum* of two, any lonely Primary (after connection loss) would first freeze all IO, then wait for either re-established communications (and then re-transmit requests and resume), or an administrative (operator or cluster manager induced) request to reduce the Write-Quorum to one (and then generate the new data generation tag and do the rest as explained above).

You (your cluster manager) would only reduce Write-Quorum after making sure that the other node(s) won't fiddle with their copy of the data.

Cache warming (not only) for database fail-over

Databases often perform comparatively poorly during the "cache-cold" time after restart. This may impact the perceived (quality of the) availability of the database after a fail-over (note that without fail-over it would simply be not available, though).

We can improve this by pre-heating the cache on the Secondary, thus avoiding "cache-cold" effects, or at least considerably reducing them. For databases operating near the limit of their hardware, this should drastically cut down on the time needed after fail-over to reach the optimal performance level again. We need to ship hints to the Secondary for reads (which usually are only done locally), so it in turn can hint the VM to try and fault those blocks in. For writes, we should directly insert/replace the pages into the buffer-cache.

We are working on this. It should be available by the end of the year, if this is at all possible with the available kernel API (I think it is).

Resources

More publications and papers, documentation, and the software, are available online at the DRBD Project Hompage `http://www.drbd.org`.

You may want to subscribe to the mailing list `http://lists.linbit.com/listinfo/drbd-user`.

You should browse (and help to improve) the FAQ `http://www.linux-ha.org/DRBD/FAQ`.