DRBD

Philipp Reisner <philipp.reisner@gmx.at>

April 22, 2001

Abstract

DRBD is a kernel module for building a two-node HA cluster under Linux. It supports different protocols in order to meet a broad range of user needs. It is shown that the potential dependencies between written blocks can be easily analysed on the sending node in order to allow limited write reordering on the receiving node. The device reaches between 50 % and 98 % of the maximum theoretical performance. DRBD-based clusters have been used in production since June 2000 and, overall, they have done very well. They are used to mirror various databases and to form a highly available NFS Server.

1 High availability by redundancy

Hard disk mirroring (RAID1) is a well known method to increase the availability of servers. It prevents data loss in the case of hard disk failure. Mirroring inside a single machine, however, does not contribute to availability if a component other than the hard disk is failing. A short distance between the two hard disks also does not protect data from disasters like fire.

The first challenge is solved by so-called HA clusters, where the active server is backed up by a standby machine. Usually these clusters are equipped with shared disks. A shared disk is a hard disk which can be accessed from all nodes of a cluster. In an HA cluster the shared disk is usually a RAID-set of disks. But in shared disk clusters the distance between the disks is still very low, since the disks of the RAID-set are located inside a single case.

DRBD is a device driver for Linux which allows you to build clusters with distributed mirrors, so-called "shared nothing" clusters. This architecture does not only have the advantage that the physical distance between the two copies of the data can be magnitudes greater than with shared disks, but it is also (magnitudes) cheaper than configurations with shared disks. (See also Appendix B)

2 Building blocks

The components needed to build a simple cluster are a mechanism to synchronise the cluster node's disks, a monitoring component, a file system, and finally the daemon providing your service.

The monitoring subsystem monitors the functioning of the primary cluster node. If it encounters a failure, it carries out the failover process.

Thus, the file system needs to be able to go online from any state, since we cannot predict the time of a failover. After a failover the file system must be in the same state as before the failover. An important prerequisite is provided (among others) by journaling (roll forward) file systems.

Finally, the service you provide also needs to handle the failover. It must for instance have a stateless network protocol or a built-in retry mechanism. If the service maintains files, it must be able to tolerate a file which was left in an inconsistent state by the failing instance.

3 Block device drivers under Linux

At first we need to have a closer look at the characteristics of a block device:

- You can pass blocks to the block device.
  But the call (ll_rw_block()) may return before the blocks have safely arrived on the real device.
- The block device may change the order of the write operations.
- You can check if a block has reached safety on the real device (buffer_uptodate()).
- You can wait until a block has reached safety (wait_on_buffer()).

From the block device's point of view, you get blocks (do_request()) and you need to signal the completion of IO (end_request()). You may reorder blocks you get by a do_request() call.

It is very critical to signal the completion of write requests without compromising correctness while delivering good performance.

4 Protocols

4.1 Protocol A

Protocol A signals the completion of a write request as soon as we have written the block to the local disk and sent it out to the network. We signal the completion of the operation without knowing whether the block has arrived or will arrive on the disk of the mirror.

Of course this protocol is not suitable for every application, since it may violate the transaction behavior of the system. Example:

> A database signals its client that the last transaction was completed sucessfully. But the blocks modified by the transaction are still on their way to the standby cluster-node. If the active node is crashing now, the standby machine will roll-back the last trasaction since it was incomplete and continue to offer the service to the clients.

While this protocol is not suited for mirroring your local databases, it is very well suited for long-distance mirroring, since it has the lowest preformance penalty for the sending system, especially on long[1] links.

4.2 Protocol B

Protocol B is better suited for making the database used in the above example highly available. Protocol B considers a write operation complete as soon as we receive an acknowledgement that the block was received by the standby system.

It has to be noted, however, that protocol B still presents a small residual risk, arising when the cluster manager is unaware of protocol B's properties:

> The standby server fails after it has issued an acknowledgement and before it has had the chance to write the blocks to disk. The primary server may signal a client that a transaction was successful and fail afterwards.
> The operating team decides to repair the former standby server, thus the former standby server becomes the new active server. – This server has not got the last transaction, but the client thinks that the transaction was successful.

Protocol B is not very well suited for mirroring a complete data-processing center across the Atlantic, because it will slow down the operation of the sending system on a long-latency network link. This is caused by the limited number[2] of request slots of Linux. If no free request slot is left, an application that tries to issue a further write request is blocked until a request slot is freed (= an older request gets completed).

4.3 Protocol C

Protocol C considers a write operation complete when a block-has-been-written acknowledgement is received from the standby system. – This protocol can guarantee the transaction semantics in all failure cases.

[1] A link with high bandwidth and high latency; a link which stores a lot of data in itself.

[2] Linux's request queue has a fixed length of 128 entries. Only 42 of these slots can be used by DRBD's write requests.

5 Write ordering

Some file systems require that certain blocks hit the media in a determined order, for example a JFS needs to write a transaction (the commit record must be last) into the journal before it does any updates to the home locations.

It does this by postponing the home location updates until it knows that the writes to the journal are on stable storage. (This is done with wait_on_buffer() and/or buffer_uptodate())

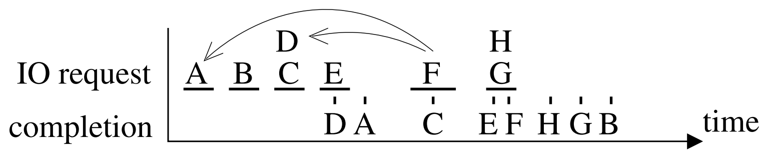
From DRBD's point of view the question is which blocks might be reordered when writing to the secondary's disk.

To ensure exactly the same write order as on the primary, we must use the following scheme:

1. Get a block from the network and put it onto the buffer cache.
2. Write that buffer and wait for IO completion.
3. Continue with 1.

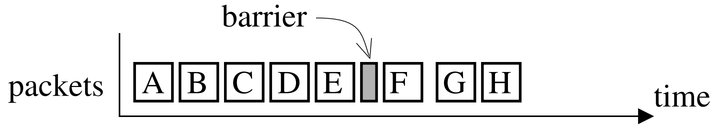
It is possible to soften this restrictive scheme a bit. It is known that there is no dependency between two blocks if there has been no IO completion event for the first write before arrival of the second write.

Example:

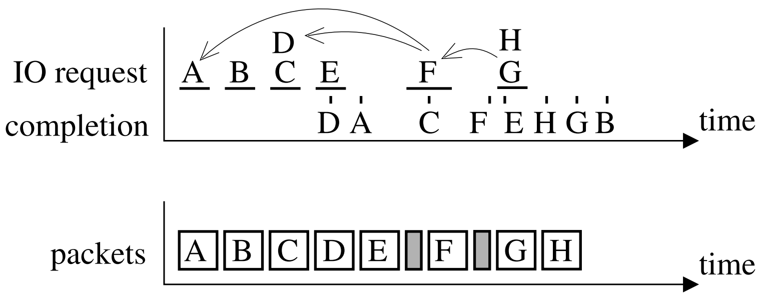


There is no causal dependence between A, B, C, D or E, but F might depend on the IO completion of D and A. F does not depend on C but G might depend on C, and so on.

We could exploit this by adding every block we get to an epoch set. When we signal a write completion event for a block from this epoch, we need to issue a write-barrier message on the wire and clear the epoch set.



Another example:



This would allow to pass blocks within the bounds of two write barriers to ll_rw_block() without needing to wait for IO completion between them. When receiving a write barrier, the write of further blocks must be postponed until all blocks of the former epoch have been written.

6 Cluster wide state management

DRBD provides a virtual shared disk to form a highly available computing cluster. By doing so it relies on a cluster manager software in charge of monitoring the nodes of the cluster and of reconfiguring the nodes of the cluster in case of a node failure.

6.1 Heartbeat

By the time I started working on DRBD, the only cluster manager available to the open source community was Alan Robertson's heartbeat, and therefore heartbeat has so far been used as cluster manager for DRBD-based clusters.

An instance of heartbeat runs on each node of the cluster and monitors all nodes. If a node fails, it simply starts the services which were running on the failed node on the node that is still working. When a failed node rejoins the cluster after having been repaired, the services are automatically failed back to their home node.

6.2 DRBD

Heartbeat does its job as a cluster manager quite well, but DRBD brings new requirements to the task of cluster management, which originally were not met by heartbeat.

Since DRBD is used on all nodes of the cluster, it is vital to ensure that the applications are always running an a node which has the up-to-date data on its disk and that the nodes that do not have up-to-date data are updated as quickly as possible.

6.3 Meta-data

In order to be able to decide which node has the up-to-date copy of the data, DRBD needs to maintain a small amount of meta-data consisting of the **inconsistent-flag** which indicates if the node is the target of a running synchronisation process and thus currently has inconsistent data on its disk and the generation-counter which itself has four parts. The four parts of the generation-counter are:

human-intervention-count This counter is increased when the state of the cluster is changed by human intervention, it has the highest priority of all four parts.

connected-count This counter is increased if the state of the cluster changes and the node is part of a running DRBD cluster, or the secondary node just left the running cluster.

arbitrary-count This counter is increased if the state is modified by the cluster management software when the node is not part of a running cluster.

primary-indicator The active node of a running DRBD cluster sets it to 1 while the standby node sets it to 0.

This meta-data is stored in a non-volatile space on each node. If the nodes can communicate and one node is active, then the active node's generation count overrides the generation-count of the standby-node (of course with the exception of the primary-indicator).

During the node's boot process it needs to look out for its partner node. If it cannot talk to its partner node, it has to wait until it becomes available or an operator decides that it has to become the active node.

In order to identify the node with the up-to-date data, the components of the meta-data are investigated in the order they are mentioned above. If a node with set inconsistenflag is present, the other node has the up-to-date data. If the first components of the generation-counters (the human-intervention-counters) are different, the node with the higher value has the up-to-date data. The second components are considered only if the first components are equal. If even the connected-counts are equal, the next components (the arbitrary-counts) are considered, and so on.

6.3.1 Examples

The values of the generation-counter are written as ordered quadruples.

In the first example we have the situation that node A has the up-to-date data, since it was running longer before both nodes were down, clearly node A has to become the primary after cluster restart:

P	Node A	Node B	Description
1	P<0,0,0,1>	S<0,0,0,0>	Both nodes up, generation-counter is equal.
2	P<0,1,0,1>	-<0,0,0,0>	Node B goes down, node A increases the second component, because it was part of a running cluster.
3	-<0,1,0,1>	-<0,0,0,0>	Both nodes are down.
4	?<0,1,0,1>	?<0,0,0,0>	The cluster is restarted.
5	P<0,2,0,1>	S<0,2,0,0>	A updates B and, again, the second component is increased because the nodes are connected again.

This is the classic example of a highly available cluster, where node B has to take over while the former primary is down:

P	Node A	Node B	Description
1	P<0,0,0,1>	S<0,0,0,0>	Both nodes up, generation-counter is equal.
2	-<0,0,0,1>	S<0,0,0,0>	Node A goes down, node B does not increase the generation-counter.
3	-<0,0,0,1>	P<0,0,1,1>	Heartbeat starts the services on B.
4	-<0,0,0,1>	-<0,0,1,1>	Both nodes down.
5	?<0,0,0,1>	?<0,0,1,1>	Cluster restart.
6	S<0,1,1,0>	P<0,1,1,1>	A synchronises from B and B becomes primary.

This example shows the common power failure, and with the meta-data management as outlined in this section, it is perfectly fine to use protocol B for an application that requires valid transaction behaviour, since the example in section 4.2 assumed a wrong decision by the cluster manager:

P	Node A	Node B	Description
1	P<0,0,0,1>	S<0,0,0,0>	Both nodes up.
2	-<0,0,0,1>	-<0,0,0,0>	Common power failure.
3	?<0,0,0,1>	?<0,0,0,0>	Cluster restart.
4	P<0,1,0,1>	S<0,1,0,0>	The former primary becomes primary again.

This example outlines human intervention:

P	Node A	Node B	Description
1	P<0,0,0,1>	S<0,0,0,0>	Both nodes running.
2	-<0,0,0,1>	S<0,0,0,0>	A's power cable fails; B leaves the generation counter unchanged.
3	-<0,0,0,1>	P<0,0,1,1>	Heartbeat starts up the services on node B.
4	-<0,0,0,1>	-<0,0,1,1>	Unfortunately B's hard disk fails now.
5	?<0,0,0,1>	-<0,0,1,1>	A's power cable is repaired and the operator decides that it is best to restart node A now.
6	P<1,0,0,1>	?<0,0,1,1>	Magically B's hard disk recovers, but ruling of the operator is accepted.
7	P<1,0,0,1>	S<1,0,0,0>	

And finally the worst case scenario, a temporarily network outage:

P	Node A	Node B	N	Description
1	P<0,0,0,1>	S<0,0,0,0>	W	Both nodes running; Network working.
2	P<0,1,0,1>	S<0,0,0,0>	B	Network breaks. A increases the second component, since the secondary left the cluster.
3	P<0,1,0,1>	P<0,0,1,1>	B	Heartbeat on node B starts the services since it assumes that A is down.
4	-<0,1,0,1>	-<0,0,1,1>	B	Power fails.
5	?<0,1,0,1>	?<0,0,1,1>	W	Power and netwok are working again.
6	P<0,2,0,1>	S<0,2,0,0>	W	

6.4 Acknowledgements

I wish to acknowledge Alan Robertson and other members of the DRBD mailing list who originated and helped to develop the idea of keeping persistent metadata in a tuple of counters of different importance to help manage the versioning problem.

7 Synchronisation when a node joins the cluster

Beside the normal operation where data blocks are mirrored as they get written, it must be possible to synchronise the content of the mirrored disks. This is necessary if one node rejoins the cluster after an outage or if the cluster is restarted after an outage of both nodes.

Synchronisation is designed not to affect the operation of the node that runs the cluster's application. It runs in parallel to the normal mirroring. In order to ensure that the application on the active node is not slowed down by the resynchronisation, the resynchronisation may only use a limited amount of the network's bandwidth.

The usual way to update the hard disk of a node is to copy every block from the active node to the standby node. This is called **full synchronisation**.

If a node leaves the cluster for a short time[3], it is only necessary to copy the blocks to the joining node that were updated while the joining node was away. This is called **quick synchronisation**.

Quick synchronisation is implemented with an in-memory bitmap that records all modifications to blocks while the standby node is away. As there are no write acknowledgement packets in protocols A and B, there is an acknowledgement packet for write barriers. In the case of a lost connection, all packets in not-yet-acknowledged epoch sets are immediately marked as out-of-sync in the bitmap.

But there is an unobvious restriction to the use of quick synchronisation. Quick synchronisation may only be used when the standby node has left and rejoined the cluster. Here is an example that shows that it is not possible to do quick synchronisation after an outage of the node in primary state:

[3]The only requirement for a *short time* is that the active node is not restarted during this time.

The application on top of DRBD decides to write a data block to the storage and therefore the block is written to the local disk and sent via the network to the standby node. The block reaches the local disks, and just before it can leave the node via the network interface card the node crashes. The standby node takes over and starts the application and, as long as protocol B or C were used, the write of the block was never acknowledged to the file system and the cluster is in a valid state after the failover.
But if the former primary node would join the cluster (now as standby node), updated only with a quick synchronistaion, it would have that one block on its disk that did not go out to the network before it crashed.

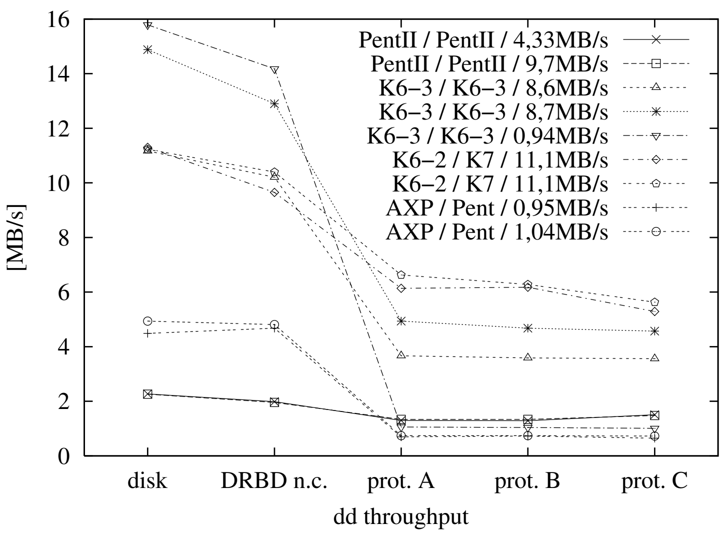
A quick synchronisation can be interrupted and continued by a failure of the standby node, since a bit gets cleared only if the standby node confirms that the block has been written to disk.

8 Performance

The most important constraint on DRBD's performance is of course the performance of the components involved, the two disks and the interconnection network between the nodes.

Read as well as write throughput in degraded mode are 89 % of the throughput of the local disk with a very low deviation of 1.41 %.

Throughput of mirrored writes is affected by the performance of both disks and the performance of the network. This chart gives only a very rough overview of achieved results.



The lines are labelled with the processors of the active and the standby system and the throughput of the interconnecting network.

9 Experiences in production

At CUBiT IT Solutions we have been using DRBD in production for more than nine months now. It is used on two clusters, one of them runs two instances of mySQL databases and a NFS server and the other one runs a PostgreSQL database and a messaging server, which also contains a database.

The PostgreSQL cluster is running stable by itself and has not had a single failover in months, which is probably due to the fact that it is only very slightly loaded.

The other cluster, which is running the mySQL databases, is rather heavy loaded and has caused severe troubles. We had crashes caused by:

- machine check exceptions[4]
  It seems that these are caused by hardware defects in CPUs or mainboards.
- crashes of SCSI busses
  We are using expensive big-name SCSI RAID controllers in these machines, but as one of the hot pluggable SCSI disks died, the whole SCSI bus was not usable any more and the machine crashed hard.
- kernel bugs
  Various Linux 2.2.x kernels have bugs which cause complete systems hangs under high memory pressure. The system keeps on printing "do-try-to-free-page-failed for process XXX" onto the console screen, and does nothing else any more.

With all these crashes the cluster built out of DRBD and heartbeat worked as expected, and kept our customers' applications up and running (of course interrupted by the failover time).

The following shortcomings and problems we discovered:

- Resynchronisation is ways too slow.
- When heartbeat gets confused, it sets both nodes of the cluster into primary state. With the DRBD release which was stable at that time it was not possible to recover from this situation without shutting down the application for a short time.
- Since the stable release does not include the meta-data management nor was heartbeat ready to deal with DRBD's requirements concerning which node needs to become primary, cluster recovery after a common power failure is ways too complicated for use in production.

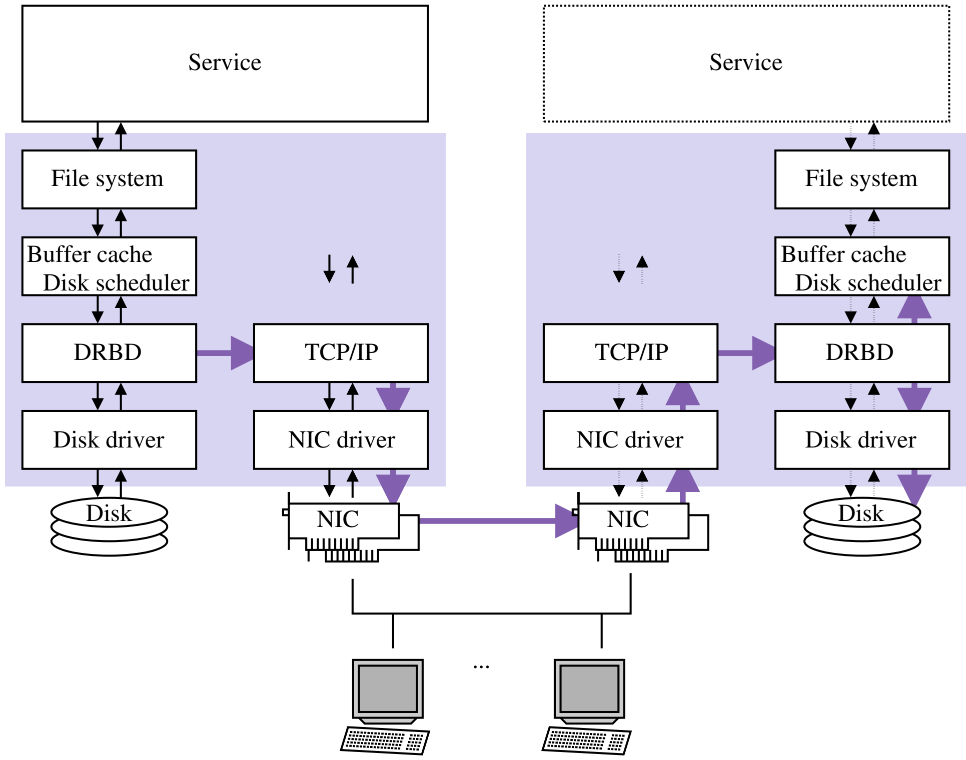
These problems are currently beeing worked on and by the time of the publication of this paper they will hopefully be eliminated.

[4]Machine check exceptions were introduced with the PentiumIII CPU and inform the operating system that the current CPU context is corrupted.

A Resources

Web site	http://www.complang.tuwien.ac.at/reisner/drbd/
Mailing list	http://lists.sourceforge.net/lists/listinfo/drbd-devel

B System overview



References

[Pfi98] Gregory F. Pfister. *In search of Clusters.* Prentice-Hall PTR, Upper Saddle River 1998.

[Rei00] Philipp Reisner. *DRBD Festplattenspiegelung übers Netzwerk für die Realisierung hochverfügbarer Server unter Linux.* Diploma thesis at the Vienna University of Technology. Http://www.complang.tuwien.ac.at/Diplomarbeiten/reisner00.ps.gz

[Rob00] Alan Robertson. *Linux-HA Heartbeat System Design.* Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10-14, 2000, Atlanta, Georgia.
Http://www.usenix.org/publications/library/proceedings/als2000/robertson.html