# DRBD

Philipp Reisner <philipp@linuxfreak.com>

31st August 2000

**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.

## 1 High availability by redundancy

Hard disk mirroring (RAID1) is a well known method to increase the availability of servers. It prevents loss of data in the case of hard disk failure. But mirroring inside a single machine does not contribute anything 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 challange 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 these 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 A)

## 2 Building blocks

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

The scope of this paper and the DRBD kernel module is the synchronisation of the block devices, therefore the “split brain” problem is not part of this paper. I think that the “split brain” problem should be handled in the monitoring subsystem.

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 predetermine 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 arrived in safety 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.

When building a network mirroring block device, it is very critical to signal the completion of IO without compromising correctness on the one hand and delivering good performance on the other.

## 4 Protocols

### 4.1 Protocol A

With protocol A we signal 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.

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

### 4.2 Protocol B

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

Unfortunately there is still a small window of vulnerability in this protocol:

> 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

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

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

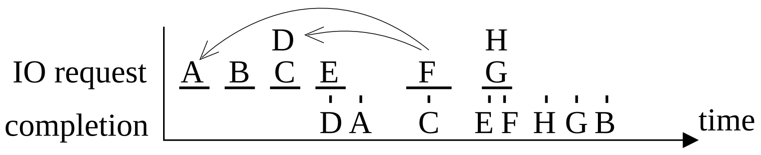
To ensure exactly the same write order as on the primary, we must use the following scheme:

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.

[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.

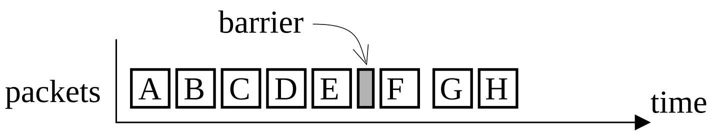
It is possible to lift this restrictive scheme a bit. It is known that there is no dependency between two blocks if there has been no IO completion event for the first write before arrival of the second write.

Example:

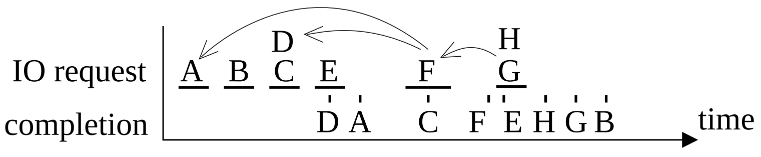


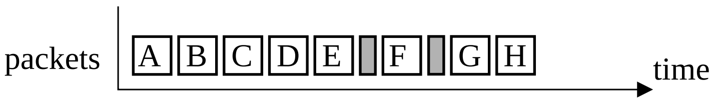
There is no causal dependence between A, B, C, D or E, but F might depend on the IO completion of D and A. F does not depend on C but G might depend on C, and so on.

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 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 synchronize the content of the mirrored disks. This is necessary if one node rejoins the cluster after an outage.

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.

If a node joins the cluster for the first time, every block needs to be copied 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 where 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. Because 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.

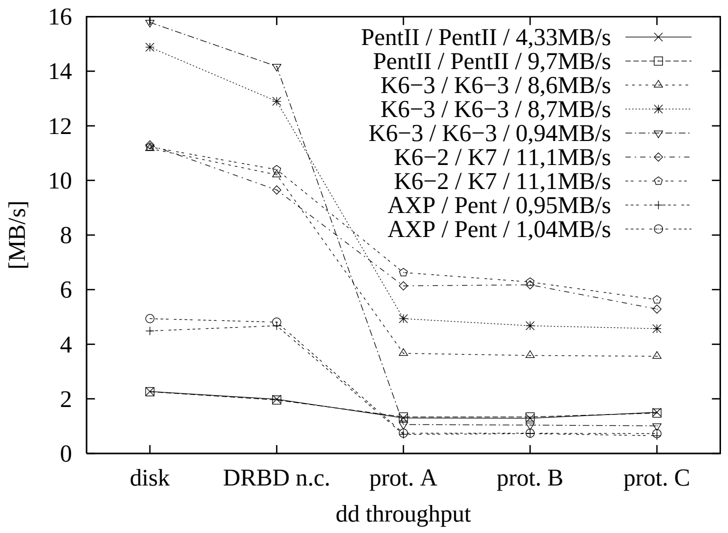
A quick synchronisation can be interrupted and continued by a failure of the standby node, since a bit gets cleared only if the standby node confirms that the block has been written to disk.

## 7 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.



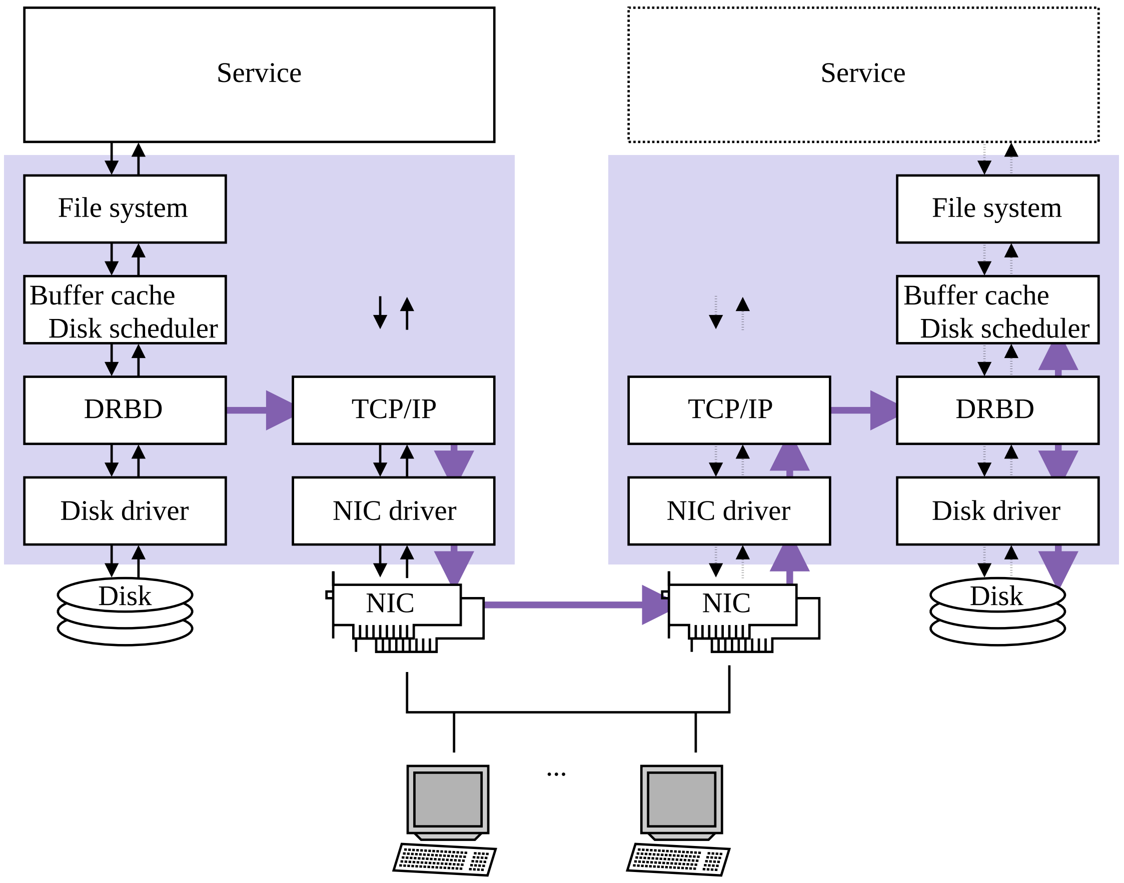
The lines are labelled with the processors of the active and the standby system and the throughput of the interconnecting network.

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

## 8 Status

While DRBD is already used by eartly adopters for production systems, it is still actively developed. At present heartbeat is the only cluster manager for which glue scripts are provided with DRBD. You can find the described software and hands-on documentation at http://www.complang.tuwien.ac.at/reisner/drbd/.

## A System overview



## References

[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

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