DRBD

Philipp Reisner <philipp@linuxfreak.com>

2nd February 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's needs. It is shown that the possible causal dependencies between written blocks can be easily analysed on the sending node, in order to allow limited write reordering on the receiving node.

1 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 synchronization 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. (Monitoring of services would be better)

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 from the failing instance in an inconsistent state.

2 Block device drivers under Linux

At first we need to have a closer look on 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 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()).

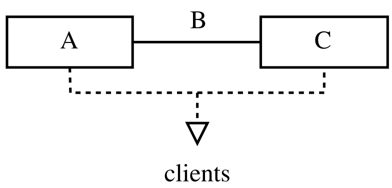
- You can wait until a block has reached safety (wait_on_buffer()).

All current file systems are written on top of these features.

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 one do_request() call.

When building a network mirroring block device signaling the completion of IO without compromising correctness while delivering good performance is very critical.

3 Model



A and C are cluster nodes while B is the interconnecting network. In this model a component can be in working or not-working state.

4 Protocols

4.1 Protocol A

If we want to mask the failure of exactly one component, i.e. the failure of one node or the failure of the connection, we can signal the end of a write operation when the block has left the primary node. This is only possible if the communication channel is reliable by itself. Unfortunately this is not the case with TCP over ethernet.

Failure cases:

- The primary node fails.
  The block arrives at the secondary node, thus the secondary node is in the same state as the primary was in the moment of failure.
  Attention: The service may not be started as long as a block might be in the communication channel.
- The secondary node or the communication channel fails.
  Obvious.

4.2 Protocol B

Since ethernet is not reliable by itself, we must postpone the IO completion event to the time when we know that the block arrived at the secondary node. The secondary node sends a block-received-acknowledgement message when it received the block.

This works correctly for the single failure case.

When omitting the single failure assumption, this solution runs the risk of violating correctness:

> The primary node fails after indicating the the write as complete, the secondary node fails before the block is written to the secondary node's disk and finally, when power returns, the former secondary node comes up first thus becomes the new primary node. In this case the last block would be missing, while the old primary node could already have taken action assuming that the block had been written.

In order to solve this problem, the monitoring subsystem could be required not to assign the primary role to another node after the whole cluster had been down.

But you should keep in mind that by dropping the single failure assumption in a two node cluster, you might lose your service anyway.

4.3 Protocol C

Without the single failure assumption, we must take an even more conservative approach. The secondary node sends a block-written- acknowledgement when the block has reached stable storage. The primary node signals the completion of IO upon arrival of the acknowledgement.

With protocols B and C upon the failure of the connection or the failure of the secondary node, IO completions must be signalled for the outstanding blocks to ensure continued operation of the primary node.

5 Write ordering

Some file systems require that some 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 (to my knowledge) 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())

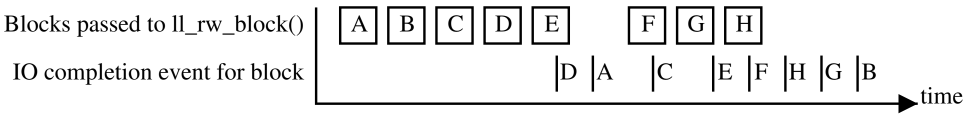
From the drbd's point of view the question is, which blocks might be reordered when writing to the secondary's disk.

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 the IO completion.
3. continue with 1.

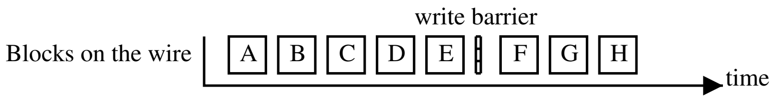
It is possible to lift this restrictive scheme a bit. It is known that there is no causal dependence between two blocks iff there has been no IO completion event for the first write before arrival of the second write.

Example:

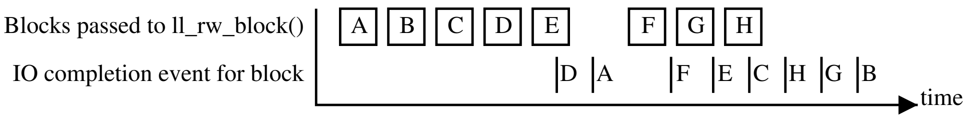


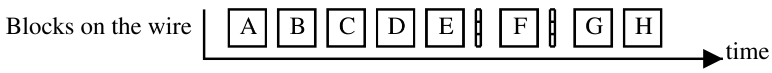
There is no causal dependence between A, B, C, D or E, but F might depend on the IO completion of D and A. F does not depend on C but G might depend on C, and so on...

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 are written.

6 Synchronization when a node joins the cluster

Apart from a synchronization mechanism that copies every block, I have implemented a smarter algorithm which only copies the blocks which where updated since the node left the cluster.

It sits on top of an interface which will also allow file systems to state what needs to be updated. (Think of a file system consuming only 10% of the available blocks)

It is implemented as a bitmap stored in memory, thus it cannot speed up synchronization when both nodes crashed. It is intended to assist sysadmins in taking single nodes down for maintainance.

When the connection is interrupted, the primary node continues operation. It marks the blocks from all not yet acknowledged epochs as out-of-sync. Of course all further write operations, which are only applied to the local disk, are also marked in the bitmap.

Upon rejoining of the secondary node, only the marked blocks need to be copied. The bandwith used by resynchronisation is user-adjustable and should of course not impede normal operation. The resynchronization uses protocol C, which is in this case not a performance limiting factor. The block becomes unmarked upon receipt of the write acknowledgement.

6.1 Two node clusters at their limit

You should keep in mind that when one node is down (be it for maintainance or not) the cluster does not provide you with high availability.

The synchronization as described above bears even more risk. E.g. if the disk of the primary node breaks during the synchronization process, you are left with nothing, because during synchronization the secondary node's disk is not in a consistent state. The order of the updates applied to the secondary node's disk is determined from the scan of the bit map. Of course this order has nothing to do with the *real* order, thus your JFS will not be able to recover from what is left.

7 Disk failures

A failure of the primary node's local disk should cause a failover. Therefore drbd should have the possibility to demand a failover on the primary node.

8 Conclusion

I want to implement protocols A, B and C, and support the write barriers.

I suggest advancing Linux's interface to the IO system in a way that the file system can inform the IO system of the causal dependencies between the written blocks (, maybe with introducing the ll_write_barrier() call).

9 Current Status

Nearly all off the described features are implemented, but comprehensive testing is still to be carried out. Drbd is a block device implemented in a module for Linux 2.2.x and 2.3.x. There is no change of current kernel code required. You can download it from **http://mips.complang.tuwien.ac.at/reisner/drbd/**.

10 Further work

In release 2.0 of drbd I want to support clusters of more than two nodes. Since we are using ARP packets for IP address takeover, we have to have all cluster nodes in a single ethernet segment. Therefore I want to use UDP's multicast feature for an efficient implementation of drbd for bigger clusters.