DRBD

Distributed Replicated Block Device

Philipp Reisner <philipp.reisner@linbit.com>

August 12, 2002

Abstract

DRBD is a kernel module for building a two-node HA cluster under Linux. It is shown that the potential write-after-write dependencies between write requests can be easily analysed on the sending node in order to allow limited write reordering on the receiving node. DRBD clusters are able to boot unattended and can guarantee that no data is lost in case of a single node failure.
This paper describes the 0.6 release of DRBD as well as the improvements that will be available with the 0.7 release.

1 High availability with replicated storage

The obvious architectural advantage of replicated storage over shared storage is the avoidance of a common resource. In order to eliminate this single point of failure, the shared storage uses redundancy within the storage box.

In a high availability (HA) system with replicated storage the complexity is moved out of the storage box and distributed over the system. The particular requirements for such software are:

No loss of data in case of a node failure. At first glance this seems to be obligatory, but in some usage scenarios it is also possible to weaken this point to improve overall throughput.

Node rejoin without service interruption. Since the point is to improve availability, it must be possible to add a node to the running system without interrupting the service offered by the cluster.

Unattended system start process. Since there are multiple copies of the data, it is necessary to find the up-to-date copy of the data at system start time.

Unattended restart of degraded clusters. In case an unattended degraded[1] cluster is restarted, it needs to start into degraded mode without intervention by an administrator.

[1] A degraded cluster is a cluster consisting of only one working node.

2 Software architecture

The software discussed by this paper is DRBD, which is a module for the linux kernel. It provides a block device driver to the kernel. In each cluster node, the DRBD driver is in control of a "real" block device, which holds a replica of the system's data. Read operations are carried out locally, while writes are transmitted to the other nodes in the HA cluster.

TCP was chosen as transport layer since it is available for kernel modules and solves two basic problems of distributed systems: packet reordering and flow control. This decision has simplified the development but limits DRBD-based clusters to two nodes.

At present DRBD's design allows only one node to modify the replicated storage. It is possible to change the nodes' roles, but it is not possible to modify the data on both nodes concurrently. This feature would allow to use file systems designed for shared storage, like GFS.

Each DRBD device may be in primary or secondary state, hereinafter referred to as role. Write access is granted to applications only if the device is in primary state. Only one device of a connected device pair may be in primary state. If this rule is violated, they will break up the connection and form two independent, degraded clusters.

The assignment of the roles to the devices is usually done by the cluster management software which is not discussed in this paper. Heartbeat and FailSafe are cluster managers known to work with DRBD.

The next questions to answer are how closely the systems should be coupled, and if write operations should be carried out synchronously or asynchronously. Since this decision has far reaching effects on the properties of the whole system, DRBD provides three modes of operation. The engineer deploying DRBD can choose which mode to use depending on the workloads for the system to be implemented.

2.1 Protocols

Protocol A completely decouples write operations on the two nodes. The primary node passes the write operation on to its local block device and sends it to its partner node. It signals the completion of IO as soon as the IO operation on the local disk has been finished.

This loose coupling between cluster nodes is an advantage if your network link has long latency. It causes the least slowdown of the primary node.

Protocol A does not satisfy the first requirement stated in section 1, as this example demonstrates:

> An application running on the primary node might signal its clients that a transaction was executed successfully. Should the primary node crash before all the transaction's write operations were received by the secondary node, then the transaction will be incomplete on the remaining node. The application will roll back that transaction as soon as it is started on the remaining node.

Protocol B couples the nodes more closely than the previous protocol. Completion of a write operation is signaled to the upper layers of the operating systems as soon as the local IO is complete and an acknowledgement packet has arrived from the secondary node. This acknowledgement is sent by the secondary node as soon as it receives the write operation.

In **Protocol C** write operations are carried out synchronously. The primary node does not signal the completion of a write request before its local hard disk has finished the request and the block is written to the disk of the partner node. It does this by waiting until a write acknowledgement has arrived, which is issued by the secondary node after the block is written to the local disk.

2.2 Write order constraints

Applications like databases and journaling file systems maintain the consistency of their data by keeping track of modifications of their data sets in special places (logs, journals). In case of a system crash, the information in these journals is sufficient to bring the data set into a consistent state again. The application must ensure that the write operation to update the journal is finished before the update of the associated data set takes place.

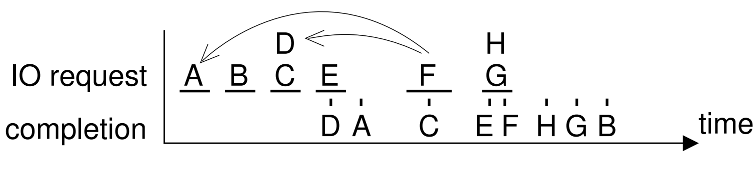
Like most modern general purpose operating systems, Linux also has a disk scheduler. The scheduler reorders pending IO operations in order to optimize the movements of the disk heads by grouping IO operations for adjacent blocks.

The applications use the fsync(2) and/or the fdatasync(2) system calls to impose write order constraints. By using these calls the applications are delayed until all pending write requests have been processed. Kernel modules may use the wait_on_buffer() function for this purpose.

In case the primary node fails, the applications have to restart from the current state of the secondary node's disk. Therefore it is necessary to also obey the write order constraints on the secondary node. With protocol C no further effort is necessary to fulfill this requirement because protocol C write operations are carried out synchronously.

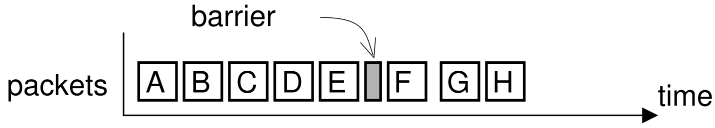
The obvious approach for protocols A and B would be to require write completion of each block before accepting the next block via the network connection, which would render the disk scheduler of the secondary node useless. It would be impossible for the disk driver to set up larger data transfers to the disk, and it would not be possible to use the full capabilities of the disk IO subsystem.

A better approach is to identify write-after-write dependencies on the primary node. These dependencies can be enforced on the secondary node by including reordering barriers in the flow of write requests.



The application does not require a defined order of the write operations of blocks A, B, C, D, and E, since they were issued before any of these IO requests had been finished. But it is possible that the application issued the write request for block F because the write operations of A and D were finished. Only the application knows if this is a real write-after-write dependency, therefore we must not violate a single possible dependency.

DRBD's approach is to keep a history of all recently sent write requests. As soon as a single block of these finishes IO, a write barrier must be issued, and the history (=epoch set) is cleared. In the above example the blocks A, B, C, D, and E are added to the history set. With the completion of D's write operation the history set gets cleared and a write barrier is issued after block E.



The secondary node can now receive blocks and commit them to its local IO subsystem immediately. If it receives a write barrier, it waits until all its pending write requests are finished before it processes any further requests.

2.3 Membership changes

It is necessary to distinguish between two cases when a node leaves the cluster:

Secondary leaves. When the secondary node leaves the cluster, the primary node knows which blocks were in progress of being written to the secondary node's disk, since all blocks that are written to the disk of the secondary node originate from the primary node.

Primary leaves. When the primary node leaves the cluster, the remaining node, which is in secondary state, does not know what the primary was about to do just before it crashed.

To make use of this principal knowledge, the epoch sets (as introduced in section 2.2) are kept in main memory until the secondary node acknowledges that all blocks of that set were written.

When the secondary node leaves the cluster, all blocks from the epoch sets are marked as modified in the bitmap. All write requests in the degraded cluster are also marked in the bitmap. When the failed secondary node rejoins the cluster, resynchronisation based on the marks in the bitmap is performed.

Upon return of a node to the degraded cluster, the contents of its DRBD mirrored disks must be updated to the state of the running cluster. As stated in section 1 this must be performed as a background task while the node with the up-to-date data continues to offer its services.

If the primary node has a valid bitmap, it is sufficient to synchronize only the blocks marked in the bitmap. This is called **quick synchronization**.

If this is not the case, **full synchronization** must be performed.

2.3.1 Pitfalls of resynchronisation

During resynchronisation, DRBD reads the data from the node of the degraded cluster (i.e. source node) and writes them to the joining node (i.e. target node). It cannot use the system-wide buffer cache, since the blocks can be modified by the file system there.

Without careful locking this can lead to concurrency issues, as shown in this example:

> A file system writes a data block, which is almost simultaneously read by the resynchronisation code. The data written by the file system is sent first, immediately followed by the block read by the resynchronisation process. But the disk scheduler might have issued the read request prior to the write request. The two versions of the data block will arrive in the wrong order at the secondary node.

To avoid these race conditions DRBD creates a semaphore if a concurrent access by the resynchronisation process and the application is detected. (These semaphores are stored in a circular list called busy_blocks.)

2.3.2 Further work in this area

The resynchronisation process brings about two major problems:

- Full synchronisation is quite time-consuming and generates a lot of network traffic.
- Apart from that, the current implementation assigns the primary role to the source node and the secondary role to the target node, although role assignment is usually done by the cluster management software. The workaround adopted in DRBD 0.6 is to delay the start of the cluster manager until the resynchronisation is finished.

To reduce network traffic, DRBD 0.7 will offer the possibility to only transfer data blocks with different crc32 hash values. The target node will send a conditional read request tagged with the hash value of its current data block. The source node will respond either with the complete data block or with a *block is in sync* message.

Under release 0.7 role assignment will be done by the cluster manager only. The target node might thus be assigned the primary role. If the application requests a not yet updated block, this request has to be passed on to the source node. Thus the primary node presents a consistent view of the up-to-date data.

2.4 System restart

While all this is about high availability it is also necessary to take a restart of the whole cluster into account. Actually it is very important that the software can deal with this automatically, since in the event of service outage people in charge of running the cluster are exposed to extreme stress.

At system start time, the first task is to find the node with the up-to-date data. To do this, the nodes must establish communication and process information which had been collected before the cluster went down. This meta-data is stored on each cluster node in a non-volatile space.

Since we are only dealing with two cluster nodes, it is possible to list all scenarios of cluster failure.

The cluster nodes fail concurrently. It is quite obvious that the node that was primary before the cluster failure has the up-to-date data. To recover from this kind of failure we need to store a single bit in the meta-data area, which is set if the node is in primary state. This bit is called the **primary indicator**. If such a bit is found at system start time, it marks the node which holds the up-to-date data.

The secondary node fails first. This case is also handled using the information of the primary indicator.

The primary fails first. The cluster manager will bring the remaining node into primary state immediately after the former primary has failed. If we need to restart now, we will find the primary indicator set on both nodes.
To deal with this situation we introduce a counter which is increased every time the cluster manager changes the role of a node while the other node is not present. We call this the **unconnected counter**. At system start time, the unconnected counters are compared and the highest value marks the node with the up-to-date data, if all counters are equal, the primary indicators are considered.

The network fails first. This is the famous split-brain-problem, each cluster node thinks that the other node has crashed. With the meta-data introduced so far we would always select the former secondary node at cluster restart time, because the cluster manager on the secondary node has to take action, change role and therefore increase the unconnected counter.
It is hard to take the right decision in this case since it is possible that both copies of the data are modified. There might be a slightly better chance to do the right thing, if the former primary is considered as the node with the most recent data at cluster restart time.
To achieve this, the **connected counter** is added to the meta-data. It is increased if the state of the cluster changes while communication between the cluster nodes is working. In addition the primary node of a cluster increases the counter if it loses contact to the secondary node.
At system start time the connected counter is considered before the unconnected counter and primary indicator are looked at.

Restart of a degraded cluster. For clusters running in remote places it might be desirable to activate the only remaining node automatically if the latter is restarted. The administrator can enable this feature by setting the *degraded mode startup* time-out. If the partner node does not appear within this timeout and the **connection indicator** is not set and the primary indicator is set, it is assumed that the other node is broken and will not recover at all.

In order to avoid deadlock when one of the nodes breaks during cluster restart, a second time-out is introduced. If the *startup* time-out expires, the **timeout counter** is increased and the system switches to degraded mode. At a normal system start the timeout counter is considered before the connected counter while searching for the up-to-date data.

Both time-outs might be set to zero, which disables the associated functionalities.

A node fails at start time. Before the selection of the node with most recent data may begin, the nodes must be able to communicate. Therefore a node simply waits for the other to appear. In case the partner node is not able to start, an operator can tell the remaining node to assume that it has the most recent data. To ensure that the broken node does not win subsequent selection processes after it has been repaired the **human intervention counter** is introduced with higher priority than all other meta-data components mentioned so far.

A node fails during synchronization. The highest priority of the meta-data has the **consistency flag.** It is cleared as soon as the node has become the target of a synchronization process.

With the exception of the consistency flag, connection indicator and the primary indicator, all parts of the meta-data are synchronized while communication is working. After system start the secondary node inherits the counter values from the newly selected primary node.

The bitmap mentioned in section 2.3 is also tagged with the three counters. If communication to the secondary node is lost, the bitmap inherits the current counters of the node. Since these were synchronized with the other node before the connection was lost, they are a sufficient label for the cluster state. If the secondary node rejoins the cluster again, a quick synchronization is possible if its counters are at the same values as the bitmap's.

2.4.1 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 meta-data in a tuple of counters of different importance to help manage the versioning problem.

3 Designed for Linux

A number of kernel threads are used:

drbd_receiver Incoming packets are handled by this kernel thread. On the secondary node, it allocates buffers, receives data blocks and issues write requests to the local disk. If it receives a write barrier, it sleeps until all pending write requests have been finished.
On the primary node it handles all incoming acknowledgement packets. In case of protocols B and C this includes finishing of write requests.

drbd_sender Data blocks written by an application are sent in the context of that application, which ensures proper flow control. Data blocks that are sent in response to a read request packet are sent by this thread. This has to be done in a thread other than drbd_receiver, otherwise distributed deadlocks would occur. If a resynchronisation process is running, its packets, read requests or conditional read requests are generated by this thread.

drbd_asender Usually hard disk drivers are informed by hardware interrupts about the completion of IO operations. Therefore they tend to activate the IO completion call-backs from interrupt handlers. It is not possible to send acknowledgement packets directly from the call-back, since sending a data packet via a TCP connection may block the caller.
Therefore the call-back only wakes up this dedicated thread, which sends the acknowledgement packets used in protocol C.

3.1 Liveliness of nodes

Each node needs to know if the other node is currently working, as it has to switch to degraded mode if its partner node does not respond. This process implies an update of its meta-data.

It is not possible to tell the liveliness of the other node by observing the packets on the TCP connections. Congestion caused by write barrier processing on the secondary node blocks connection and makes it impossible to exchange packets revealing information on the state of the peer. The exchange of out of band packets also is no option on a TCP connection where send and receive buffers are full.

Therefore it is necessary to use a second connection to see if the other node is still working. Processing of incoming packets on this connection is handled by the drbd_asender thread. Although the thread uses tcp_send() which may block theoretically, it will not block in that context because only small packets are sent through a mostly idle connection.

In the 0.7 release, when DRBD will support a shared mode for use with GFS, this second connection will also be used for acknowledgement packets since both directions of the main connection may then be blocked by congestion.

4 Performance

Throughput measuring described in this section was done using a program that generates a data stream without write-after-write dependencies. All clusters were connected by networks with low latency (<1ms).

Performance measurements of over 10 different cluster systems showed that DRBD under Linux 2.2 caused a drop of performance to 89 % (deviation of only 1.41 %) of the throughput of the local disk. This applies to all read operations as well as to write operations in disconnected mode.

DRBD under Linux 2.4-based clusters shows no measurable performance loss for local IO operations.

The reason for the difference in the performance between Linux 2.2 and Linux 2.4 systems is that on the Linux 2.2 kernel the queuing properties are radically modified by the presence of the DRBD layer. Only half of the system's request slots are available for DRBD, and since there is only one global set of these slots, only the remaining half of slots is available to schedule the local disk. Clustered requests can only be formed at the level of the real device driver, since DRBD cannot deal with them.

When looking at the performance of DRBD in connected state, the throughput is obviously limited by the disks in the nodes and the interconnecting network. Below you will find two data sets as examples (all numbers are MB/sec):

kernel	disk n1	disk n2	n.c. n1	n.c. n2	net MBit/s	prot. A	prot. B	prot. C
2.2.18	24.15	24.08	21.53	20.32	1000	19.08	18.05	19.38
2.4.12	4.15	3.58	4.06	4.04	100	4.04	3.97	4.02

5 Resources

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

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