High Availability Data Replication

Paul Clements
SteelEye Technology, Inc.
http://www.steeleye.com/
paul.clements@steeleye.com

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com/
james.bottomley@steeleye.com

Abstract

This paper will identify some problems that were encountered while implementing a highly available data replication solution on top of existing Linux kernel drivers. It will also discuss future plans for implementing asynchronous replication and intent logging (which are requirements for performing disaster recovery over a WAN) in the Linux kernel.

1 Introduction

The first part of this paper (Section 2) will discuss some issues in the 2.2 and 2.4 Linux kernels that had to be overcome in order to implement a replication solution using raid1 over nbd.

The second part of the paper (Section 3) will present future plans for implementing asynchronous replication and intent logging in the md and raid1 drivers.

2 Fixing the existing problems

We've done considerable work over the past 3 years testing, debugging, and finally fixing several problems in the md, raid1, and nbd drivers of the Linux kernel. We ran into several bugs in these drivers, primarily due to the fact that we're using them in an unusual fashion, with one of the underlying disk devices of the raid1 mirror being accessed over the network, via a network block device (see Figure 1). Our usage of raid1 in conjunction with nbd led to the increased occurrence of several race conditions and also caused the error handling code of the drivers to be stressed much more than a normal, internal disk only, raid1 setup.

The following is a brief summary of some of the problems we've uncovered and solved:

- eliminating md retries in order to avoid massive stalls when a device (in our case, a network device) fails
- correcting SMP locking errors and allowing an nbd connection to be cleanly aborted when problems are encountered
- fixing various bugs in the raid1 driver:
  - mistakes in the error handling code
  - incorrect SMP locking
  - IRQ enabling/disabling bugs
  - non-atomic memory allocations in critical regions
  - block number "off by one" error[1]

At the time of this writing, patches for all of these problems have been accepted into the latest releases of the mainline 2.4 and 2.5 kernel

[1] This problem was actually corrected by Neil Brown after our initial bug report to him.

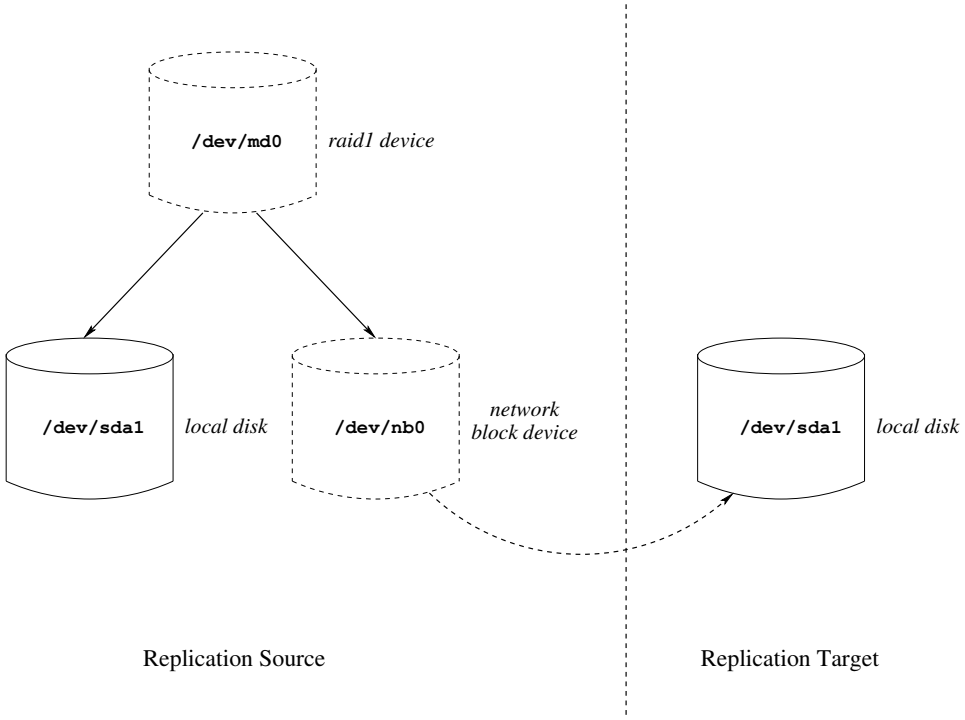

Figure 1: Data replication using raid1 over nbd

series. For information about all of SteelEye's open source patches or to download the source code for the patches, visit the SteelEye Technology Open Source Website[1].

2.1 Eliminating md retries

This was one of the first problems that we encountered back in the fall of 2000. At the time, we were working with the 2.2 Linux kernel. The md resynchronization code (`md_do_sync`) was written, at the time, to always retry any failed I/O (read or write) no less than 4096 times![2] On a network failure, this caused raid1 and nbd to spin in tight loops for several seconds, hanging the entire system. Our stopgap solution (read, hack) was to strategically insert `schedule` calls into the error handling code of those drivers.[3] Needless to say, the md resynchronization code got a major overhaul before the 2.4 kernel was released and this issue was fixed.

2.2 Allowing nbd connections to be aborted

After initially fixing a few trivial bugs in nbd having to do with missing or incorrect spin lock calls, we realized that we could not afford to wait for TCP socket timeouts when we needed to abort a network connection, or when the network went down. We needed to have the ability to terminate nbd connections at will, so that our high availability services could have complete control over the data replication process. To fix this, we unblocked `SIGKILL` during the network transmission phase of nbd so that we could send a signal from user space to terminate an nbd connection. We also needed to add code to ensure that nbd's request queue was

[2] `PAGE_SIZE`$*(1 << 1) * 2/$ `sizeof(struct buffer_head *)` (md.c, c. 2.2.16 kernel)

[3] We did not have the option of modifying md, since it is compiled into the kernel in most Linux distributions, and we did not want to be in the business of distributing entire rebuilt kernels.

cleared and all outstanding I/Os were marked as failed when a connection was terminated.

Our patches for nbd have been accepted into the mainline 2.5 kernel (c. 2.5.50) and were backported and accepted into the 2.4 kernel (c. 2.4.20-pre).

2.3 Fixing various bugs in the raid1 driver

The raid1 driver, by far, has been the biggest thorn in our side... we've made many fixes to raid1 over the past few years in order to increase its robustness. Neil Brown has simultaneously been performing a lot of cleanup and bugfix work in the md and raid1 drivers that was beneficial to our cause, as well.

The first set of problems that we ran into with raid1 was related to handling failures of the underlying devices.[4] To correct the problems, we added code to detect device failures during resync and during normal I/O operations. The additional code correctly marks the device "bad," fails all outstanding I/Os, and aborts resync activities, if necessary.

After fixing this initial set of problems, we were able to stress the raid1 driver much more heavily than we previously had been able to (without it falling over dead). Unfortunately, this heavier stress uncovered a whole raft of new problems. We were able, however, to eventually pin-point and solve each of these new problems. Many of the problems turned out to be due to some fairly common kernel programming mistakes, such as:

- **"nested" `spin_lock_irq` calls** – Failure to use the `save` and `restore` versions of the spin lock macros with nested calls (i.e., `spin_lock_irq` called while another `spin_lock_irq` is already in force) leads to the CPU flags being improperly set. This means that interrupts could be enabled at inappropriate times, causing deadlocks to occur. The rule of thumb here is that it's best to avoid nesting spin locks whenever possible, and to always use the `irqsave` and `irqrestore` versions of the macros, instead of the simple `irq` versions, if deadlocks are a concern.
- **sleeping with a spin lock held** – There were cases where the driver was doing non-atomic memory allocations or calling `schedule` with a spin lock held, which caused deadlocks to occur. To avoid the deadlocks, the code was rearranged so that a spin lock was never held while calling `schedule` and a few `kmalloc` calls were changed to use the `GFP_ATOMIC` flag rather than the `GFP_KERNEL` flag.
- **"off by one" error** – This was a simple case of differing block sizes being used in the md and raid1 drivers resulting in one of the block counts used in the resync code being shifted incorrectly. This bug caused resyncs to hang, leaving the raid device in an unusable state.

Our patches for raid1 have been accepted into the Red Hat Advanced Server 2.1 kernel (2.4.9-ac based) and an alternate version of the fixes (authored by Neil Brown) has been accepted into the mainline 2.4 kernel (c. 2.4.19-pre). The raid1 driver in the 2.5 kernel is not believed to suffer from any of the aforementioned problems.

3 Future enhancements

We are planning to enhance the md and raid1 drivers of the Linux kernel to support asynchronous data replication and intent logging.

[4] Since we use nbd underneath raid1, device "failures" are quite a common occurrence (e.g., when the network goes down).

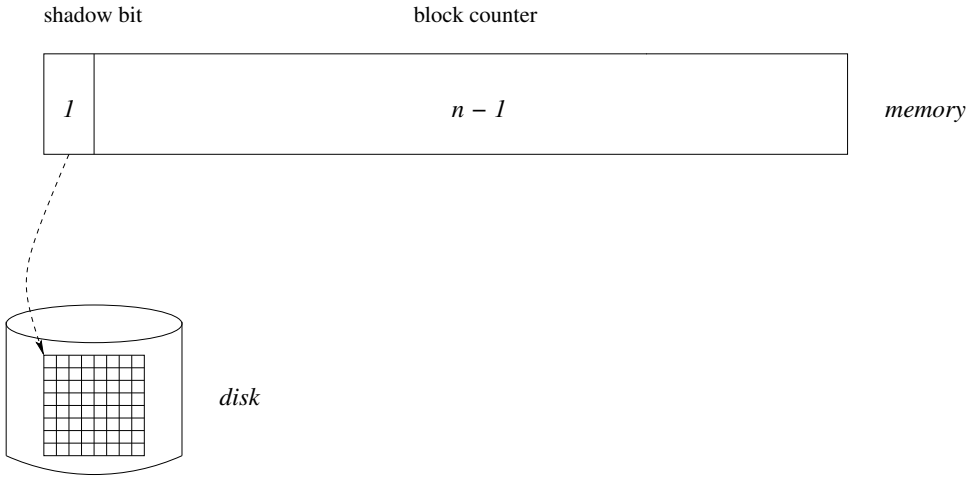

Figure 2: In-memory and on-disk bitmap layout

Our strategy for implementing these changes will be to place the bulk of the code into the md driver, in a manner that will allow all the underlying raid drivers to take advantage of it. We will also add the necessary code to raid1 to call the md driver hooks.

We plan to leverage some of the implementation and design of Peter T. Breuer's *fr1* code[2], which was recently published[3]. The *fr1* driver implements intent logging and asynchronous replication as an add-on to the raid1 driver. We will make the following, additional changes to the *fr1* code, to produce a final solution:

- disk backing for the bitmap (intent log)
- addition of a daemon to asynchronously clear the on-disk bitmap
- conversion of single bits to 16-bit block counters (to track pending writes to a given block, so as not to prematurely clear a bitmap bit on disk)
- allow rescaling of the bitmap (i.e., allow one bit to represent various block sizes—the current code is restricted to one bit per 1024-byte data block only)
- make the code fully leveragable by all the raid personality drivers
- add some additional configuration interfaces for the new features

3.1 Intent Logging

In a data replication system, an intent log is used to keep track of which data blocks are out of sync between the primary device and the backup device. An intent log is simply a bitmap, in which a set bit (1) represents a data block that is out of sync, and a cleared bit (0) represents a data block that is in sync. The use of an intent log obviates the full resync that is normally required upon recovery of an array.

3.1.1 Bitmap Layout

We will store the bitmap both in memory and on disk, in order to be able to withstand failures (or reboots) of the primary server without losing resynchronization status information.

We will use a simple, one-bit-per-block bitmap for the on-disk representation of the intent log, while the in-memory representation will be

slightly more complex. The reason for this additional complexity is the need to track pending writes, so as not to clear a bit in the bitmap until all pending writes for that data block have completed[5]. The write tracking will be handled using a 16-bit counter for each data block. One bit in the counter will actually be used as a "shadow" of the corresponding on-disk bit, reducing the usable counter size by one bit (see Figure 2). The counter will be incremented when a write begins and decremented when one has completed. Only when the counter has reached zero, can the on-disk bit be cleared.

In order to conserve RAM, the in-memory bitmap will be constructed in a two-level fashion, with memory pages being allocated and deallocated on demand (see Figure 3). This allows us to allocate only as much memory as is needed to hold the set bits in the bitmap. As a fail-safe mechanism, when a page cannot be allocated, the (pre-allocated) pointer for that page will actually be hijacked and used as a counter itself. This will allow logging to continue, albeit with less granularity,[6] during periods of extreme memory pressure.

The bitmap will also be designed so that it is possible to readjust the size of the data "chunks" that the bits represent. This will be handled by translating from the default md driver I/O block size of 1KB to the chunk size, whenever the bitmap is marked or cleared. So, with a chunk size of 64KB, for example, the I/O to 64 contiguous disk blocks will be tracked by a single bit in the on-disk bitmap (and the corresponding in-memory counter).

[5] clearing the bit prematurely could result in data corruption on the backup device if a network failure coincides

[6] On x86, with 32-bit pointers and 4KB pages, the granularity is reduced to roughly 1/1000 the normal level.

3.1.2 Bitmap Manipulation

To make use of the bitmap, we will make modifications to two areas of the raid1 driver:

1. Ordinary **write operations** will require a bitmap entry be made (and synced to disk) before the actual data is written—the bitmap entry will be cleared once the data has been written to the backup device.
2. **Resynchronization operations** will no longer involve a full resynchronization of the backup device, but rather a resync of just the "dirty" blocks (as indicated by the bitmap).

3.1.3 Write Operations

The sequence of events to write block *n* on a raid1 device with an intent log is as follows:

1. set the *n*th shadow bit in the in-memory bitmap and increment the counter for block *n* (both can be done as a single operation since the shadow bit and counter are contiguous)
2. increment the "outstanding write request" counter for the array[7] (and disallow further writes to the device if the counter has exceeded the configured limit)
3. sync the shadow bit to disk, if the on-disk bit was not already set
4. duplicate the write request, including its data buffer
5. queue the write request to the primary device

[7] This counter is really only used when the array is in asynchronous replication mode. For more details, see Section 3.2.

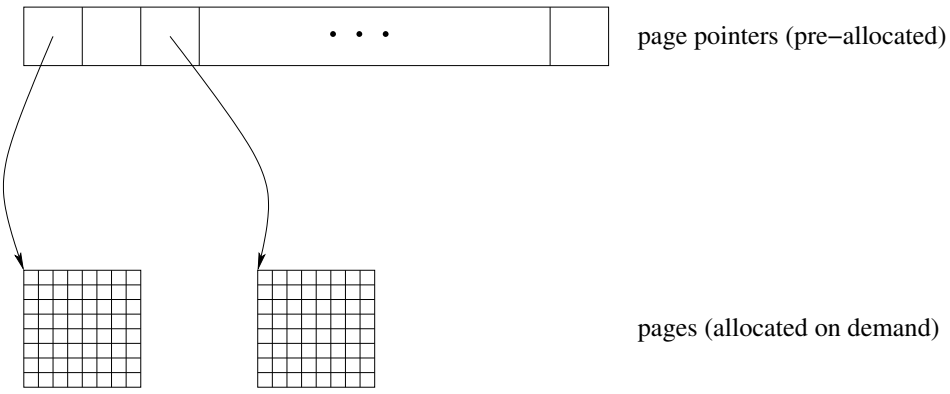

Figure 3: Two-level, demand-allocated bitmap

6. queue the duplicate request to the backup device

We then allow the writes to complete asynchronously. After each write is completed, the raid1 driver is notified with a call to its `b_end_io` callback function (`raid1_end_request`). This function is responsible for signalling the completion of I/O back to its initiator. In synchronous mode, we wait until the writes to both the primary and backup devices have completed before acknowledging the write as complete. In asynchronous mode, the write is acknowledged as soon as the data is written to the primary device.

After the write has been acknowledged, the callback function is responsible for decrementing the block counter and, if the counter's value is 0, clearing the shadow bit in the in-memory bitmap. Whenever a shadow bit is cleared, a request will also be placed in a queue to indicate that the on-disk bit needs to be cleared.

The bits in the on-disk bitmap will be cleared asynchronously, by a dedicated kernel daemon, `mdflushd`. The daemon will periodically awaken and flush all the queued updates to disk.[8] The interval at which the daemon will awaken and flush its queue will be tunable (with a default value of 5 seconds).

Clearing the bits in the on-disk bitmap in a lazy manner will help to reduce the number of disk writes, and will also ensure that any bits that happen to correspond to I/O "hotspots"[9] will simply remain dirty, rather than causing a constant stream of writes to the on-disk bitmap.

3.1.4 Resynchronization Operations

The resynchronization process of the md driver is fairly straightforward. Following recovery from a failure, the driver will attempt a complete resync of the backup device. We will modify this process slightly, so that for each data block that is to be resynchronized, we will first check the appropriate shadow bit in the in-memory bitmap and then, either:

- resync the block (if the bit is set), or
- discard the resync request and indicate success (if the bit is cleared)

Once a block has been resynced, its shadow bit will be cleared and its block counter zeroed. An update request will then be queued

[8] unless the shadow bit has been reset in the meantime, in which case the update is simply discarded and the on-disk bit is left set.

[9] areas of the disk that are continually written, such as an ext3 filesystem journal

to tell `mdflushd` that the on-disk bit should be cleared.

3.2 Asynchronous Replication

In an asynchronous replication system, write requests to a mirror device are acknowledged as soon as the data is written to the primary device in the mirror. In contrast, in a synchronous replication system, writes are not acknowledged until the data has been written to all components of the mirror. Synchronous replication works well in environments where the mirror components are local. However, when the backup device is located on a network, the write throughput of a synchronous mirror decreases as network latency increases. An asynchronous mirror does not suffer this performance degradation since a write operation can be completed without waiting for the write request and its acknowledgement to make a complete roundtrip over the network. To achieve reasonable write throughput in a WAN replication environment, an asynchronous mirror is generally employed.

3.2.1 Outstanding Write Request Limit

In an asynchronous mirror, there can be several outstanding (i.e., in-flight) write requests at any given time. In order to limit the amount of data that is out of sync on the backup device during normal mirror operation, it is necessary to keep the number of outstanding write requests fairly low. Therefore, we will place a limit on the number of outstanding write requests. However, to avoid degrading the write throughput of the mirror, this limit must be adequately high. Since the limit will need to be tuned appropriately for each environment, it will be made a user configurable parameter.[10] When the limit for outstanding writes has been exceeded, the driver will throttle writes to the mirror until another write acknowledgement returns from the remote system (i.e., the mirror will degrade to synchronous write mode). A message will be printed in the system log when this event occurs, to warn system administrators that they should adjust the relevant parameters. The outstanding write request limit will default to a reasonable value (which will be determined through testing).

3.2.2 Device Tagging

In synchronous replication mode, there is no real need to differentiate between primary and backup devices, since writes must be committed to all array components before being acknowledged. However, in asynchronous mode, the component devices of a raid1 array will need to be tagged as "primary" or "backup" to ensure that the bitmap is handled correctly, and to ensure that read requests are always satisfied from the primary device. To accomplish this, we will need an additional `/etc/raidtab` directive to enable a device to be tagged as a "backup." Devices tagged as backups will be placed into a special "write-only" mode that exists in md.

4 Conclusion

With the recent bugfix and cleanup work that has been done, and with the upcoming additional features that are in the works, the Linux kernel md driver will finally be an enterprise-class software RAID and data replication solution: robust, and capable of being used for many different applications, from simple internal disk mirroring and striping, to LAN data replication, and even disaster recovery over a

[10] To avoid overflowing the block counters in the in-memory bitmap, we will make it impossible to set this limit higher than the maximum value for those counters.

WAN.

5 Acknowledgements

We would especially like to thank Peter T. Breuer and Neil Brown for their outstanding and ongoing work in the Software RAID (md) subsystem of the Linux kernel. Without their contributions, we would not have been able to undertake such a huge endeavor.

References

[1] SteelEye Technology, Inc. *SteelEye Technology Open Source Website* `http://licensing.steeleye.com/open_source/`

[2] Peter T. Breuer. *Fast Intelligent Software RAID1 Driver* `http://www.it.uc3m.es/ptb/fr1/` `http://freshmeat.net/projects/fr1/`

[3] Peter T. Breuer, Neil Brown, Ingo Molnar, Paul Clements. *linux-raid mailing list discussions on raid1 bitmap and asynchronous writes* `http://marc.theaimsgroup.com/?l=linux-raid&b=200302` Jan-Apr 2003

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.