High Availability Data Replication

James Bottomley

and

Paul Clements

SteelEye Technology

24 July 2003

What is Replication and What are its Uses?

- Replication is the ability to duplicate data in real time across a network.
- The characteristics of this replication usually determine its use.
- It has two primary uses:
  - Replace expensive shared storage in cheap local clusters.
  - Provide disaster recovery capabilities for existing installations.
- These two uses may be characterised roughly as the extremes of the market.

Replication Characteristics

- The primary determining factors for replication characteristics are network latency and bandwidth.
- In a low latency, high bandwith (i.e. essentially a LAN or MAN), the replication is usually
  - Synchronous: data isn't reported to the application as being committed until it has made it to the disc surface on both the primary and its replica.
  - This provides a high degree of data integrity because the replica is an exact mirror of the primary if a recovery has to be done (i.e. no data is lost).
- this is local replication for shared storage replacement.

Replication Characteristics Continued

- In a high latency, low bandwidth environment (i.e. essentially a WAN), the replication is usually:
  - Asynchronous: data is reported to the application as being committed when it reaches the surface of the primary only. It may still be in-flight to the secondary.
  - This still provides data integrity, but doesn't guarantee not to lose data on a failure.
  - the lost data usually represents the last few in-flight transactions.
- This is the disaster recovery scenario, which is primarily what this talk will cover.

Asynchronous Replication

- The distance in bytes between the primary and the replica is called the "high water mark" of the replication.
- If replication is to be asynchronous, how high should the "high water mark" be?
- for Disaster recovery, the question becomes "How much data can you afford to lose?"
- Another obvious calculation is to multiply the bandwidth by the latency to get the "pipe capacity". This is the amount of data that can be held in the pipe without having to have a cache on the primary.

Asynchronous Replication Continued

- However, when the mark is reached, the data is not allowed to be committed to the primary until some data has made it to the secondary, thus slowing operations.
- Therefore, an additional cache on the primary may be desirable.
- The only required characteristic of the cache and the network pipe should be that they strictly preserve the order of the I/O blocks (thus automatically preserving the integrity of any underlying transactions).

What Happens when the Network Connection is Lost?

- Once the link breaks, the mirror is also broken—the primary operates independently as a single disc without regard to the secondary.
- When the connection is restored, the data must be resynchronized.
- Obvious simple way is to send *all* the data from the primary.
- Can speed this by compression, or simple mechanisms to recognise empty blocks.
- Can also compare block md5sums or secure hashes over the network to see if they differ.

Total Replay

- This is slow, cumbersome and may even be impractical for gigabytes of data over a narrow WAN.
- Other obvious problem is that the replay is done without regard to the transactions in the data (usually just sweep up the volume from beginning to end).
- Thus, the replica is corrupted until this replay is finished.
- If this is a Disaster Recovery scenario, you are completely unprotected while replay is going on. If something happens to the primary, you just lost everything.

Minimizing Replay or Eliminating Corruption Entirely

- If you make a log of the changes while you're disconnected from the secondary, you then need replay only the changes when communication is restored.
- Such fall broadly into two distinct types:
  - Transaction logs: The actual data in the change is recorded in order.
  - Intent logs: Only the location of changed sectors is recorded, not the actual details of the change itself.
- Each type of log has useful characteristics and provides different benefits and disadvantages.

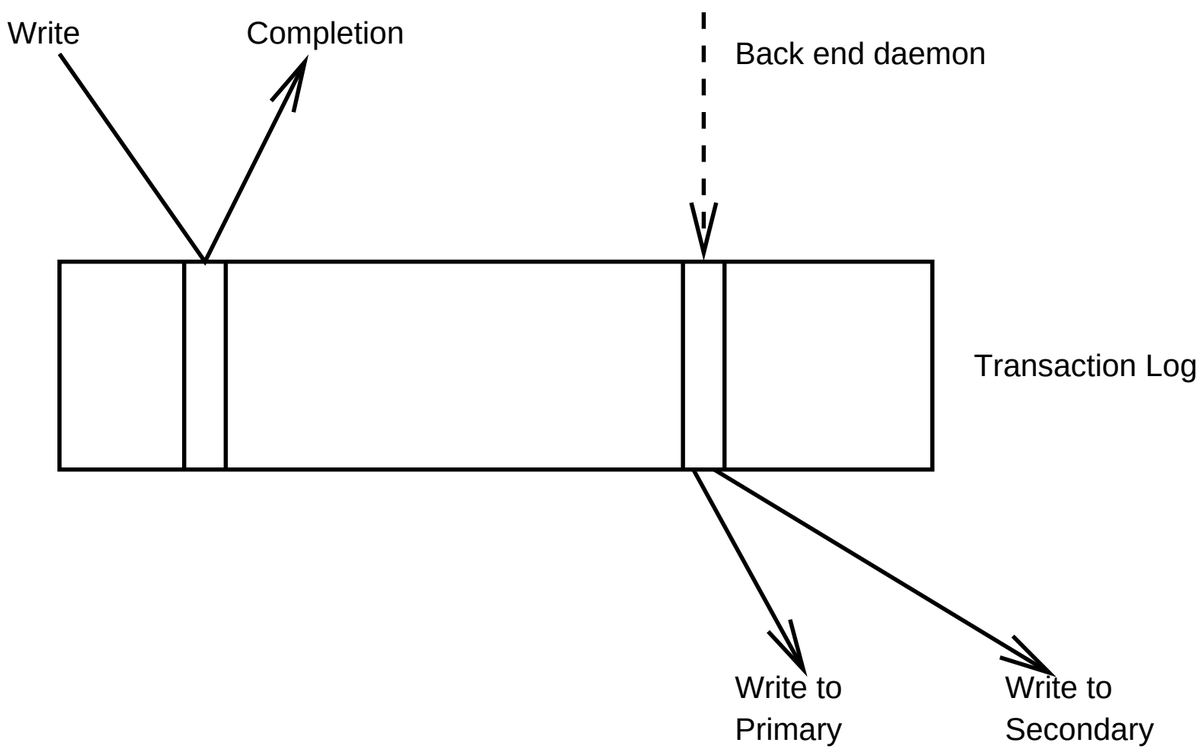
Transaction Logs

- Since these contain all the data, they may grow huge (and without bound) as the changes to the primary continue.
- Therefore, for prolonged outages, the transaction log will overflow available storage.
- However, on restoration of connection, since the transactions are ordered in the log, replaying the log brings the replica up to date without it's **ever** being corrupt.
- If you are doing Asynchronous replication, the transaction log may form part of the primary data cache.

Transaction Logs Continued

- Thus to write data, you write **only** into the cache for it to be committed (i.e. single write of data).
- You have back end daemons writing data from the transaction log (separately) to the real storage on the primary and to the remote replica.
- The "high water mark" is still the distance between the replica and the beginning of the log.
- An entry may be erased from the log when it is sent to both the primary and the replica.
- Thus, the critical write path is to get the data into the log only.

Transaction Logs Contined



Intent Logs

- Usually, you divide the device up into “chunks”.
- Each chunk being a multiple of the underlying block size.
- Every chunk gets a bit in the transaction log to indicate whether it is dirty (needs to be sent to the replica) or not.
- since the log is just a bitmap covering the device, it is usually pretty small, and a fixed size (i.e. never overflows).
- However, the log contains no transaction implementation, and thus, the replica is still corrupt as the intent log is replayed.

Intent Logs Continued

- The write path for an intent log involves **two** strictly ordered writes:
  - The first to set the dirty bit in the log.
  - The second to write the data to the primary.
- This seems to give a significant disadvantage in the write critical path.
- However, if the log entry is already dirty, this reduces to a single write (like the transaction log case).
- Thus, it pays an intent log deliberately to avoid cleaning up dirty bits after the data has been committed to the replica.
- Ideally, you simply want the data hot spots to remain dirty.

Fixing Transaction Log Overflow

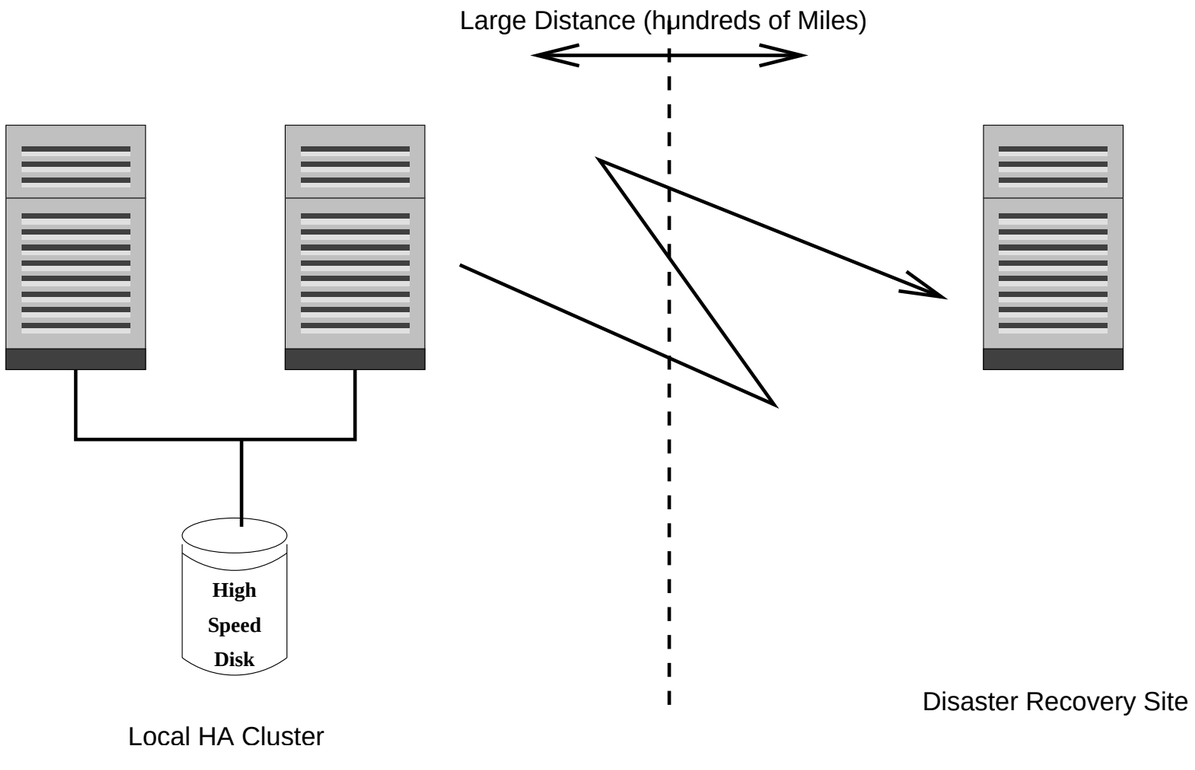
- One of the chief disadvantages of the transaction log is overflow on prolonged disconnect.
- This can be fixed by using a "hybrid" log approach:
  - The log begins normally as a transaction log
  - on disconnect it continues as a transaction log until it approaches the overflow point
  - at this point, the primary is paused and the log is converted to an intent log
  - it continues as an intent log until connection to the secondary is restored

Log Volatility

- A volatile log may be stored in memory. A non-volatile log must be stored somewhere permanently.
- A transaction log, if it is used as part of the primary data cache, is **required** to be non-volatile (you must not lose it if the primary crashes, otherwise you lose transactions from the primary.
- An intent log is not required to be non-volatile to preserve primary transactions, but may usefully prevent full replay to the secondary in the event of a primary crash if it is non-volatile.
- Thus for disaster recovery scenarios non-volatility of the log is a requirement.

Stretch Clusters

- This is a marketing buzzword.



Stretch Clusters Continued

- It means a local HA cluster where the application is
  - protected locally in the cluster but also
  - backed up remotely via replication to provide disaster recovery.
- Key point is that as the application undergoes local recovery within the cluster, the replication must follow the application.
- This **requires** a non-volatile log.

Other Issues: recovery after initial failure

- Once the Application has failed and been recovered onto the replica, one of the key problems is how to transfer it back to the primary without incurring a huge replication replay.
- Can log all the data on the secondary, but still won't know what differs on the primary because of lost in-flight transactions.
- If you're using an intent log, may replay from the secondary the union (or) of the primary and secondary intent logs to guarantee data integrity.
- doesn't work for a transaction log, but could convert both to intent logs before replay

Available replicators for Linux

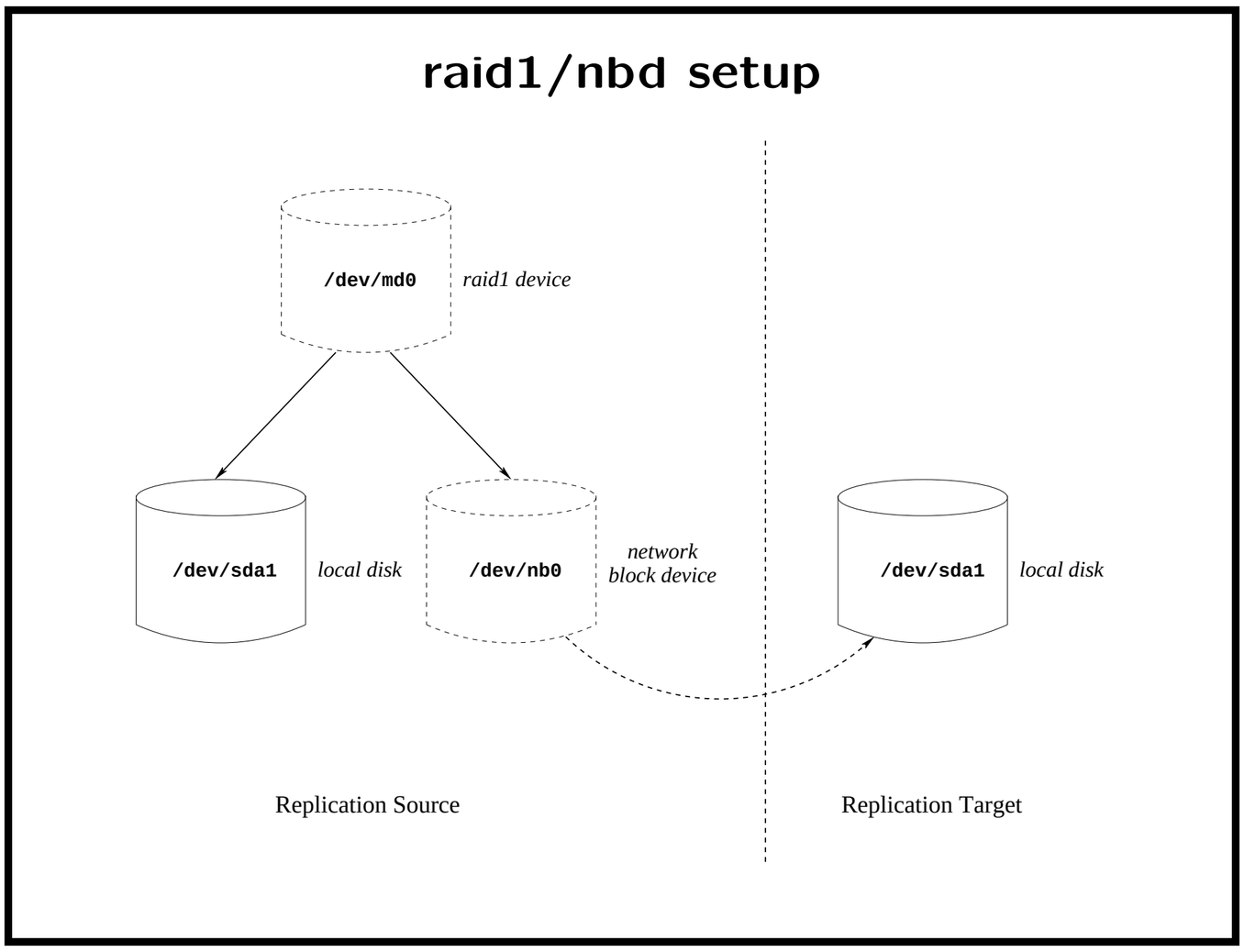
- Excluding proprietary solutions, there are two public domain replicators:
- **drbd**: by Phillip Reisner; is a complete system, includes an in-memory (i.e. volatile) intent log.
  - Main problem is lack of robustness, but various organisations are working on this
- **raid1/nbd**: May set up remote leg of a standard raid1 mirror using nbd.
  - Today, no intent log
  - however, is more robust because most of the internals are in raid1 and are shared with local mirroring (i.e. wider testing pool).

Enhancing raid1/nbd

- Goal is to add non-volatile intent logging to md itself.
- Then make use of it in the raid1/nbd configuration for disaster recovery.
- Key advantage is that logging becomes available to (and tested by) all users of md with redundancy.
- Effectively a project of mutual advantage.
- Starting point is the raid1 logging code by Peter Breuer (originally this was for volatile intent logging of raid1 only).

Why raid1/nbd?

- Essentially because we previously picked this for the SteelEye local replication product.
- We have two years experience finding and fixing bugs in this, so we've actively been ensuring that it is robust.
- Also, we know the code, so development goes faster with a familiar system.
- A good side benefit is that the work may be useful to more than just replication (well, logging at least, asynchronous mirroring is probably only useful to replication).

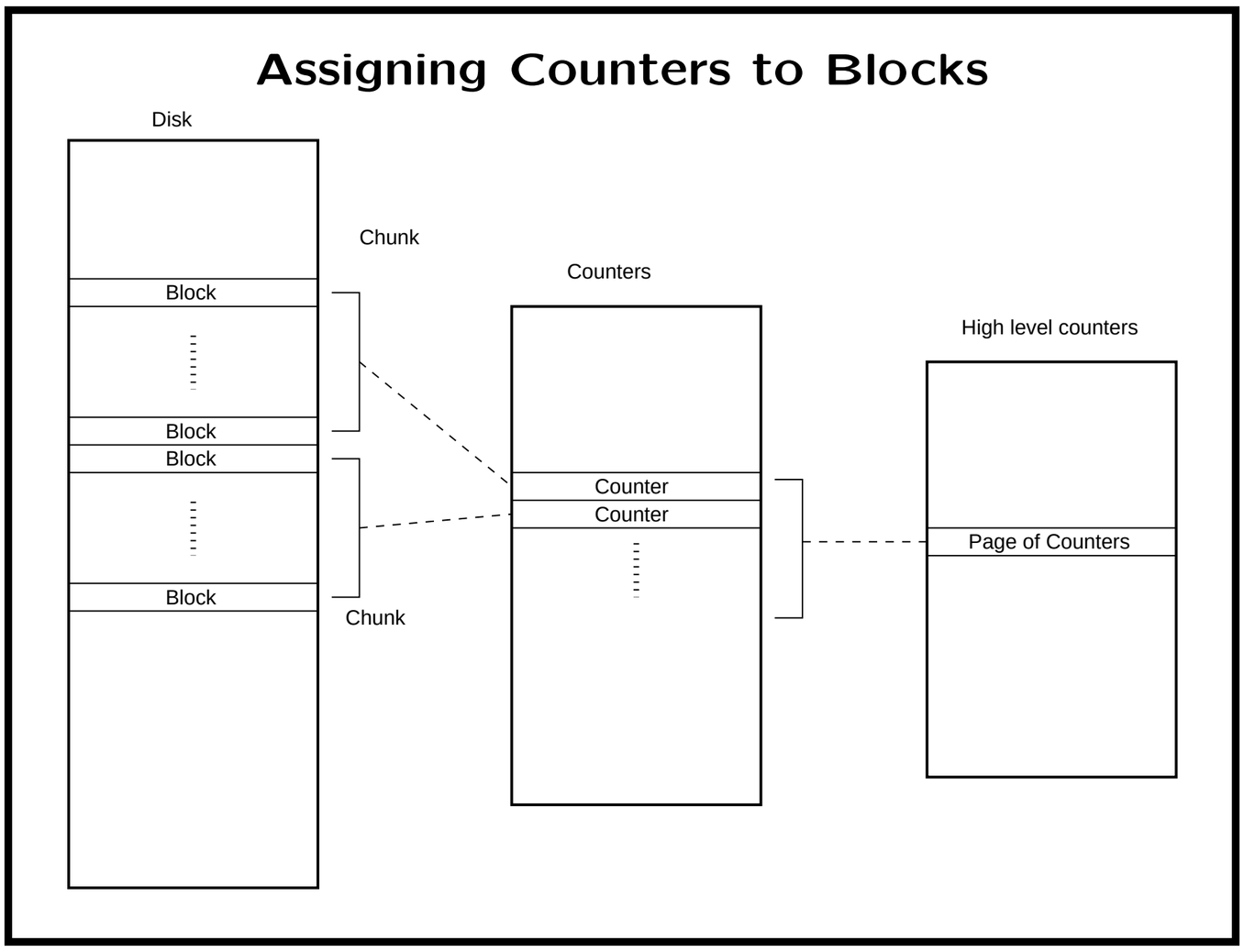

raid1/nbd setup
/dev/md0
raid1 device
/dev/sda1
local disk
/dev/nb0
network
block device
/dev/sda1
local disk
Replication Source
Replication Target

Adding intent logging

- Intent logging is the simplest form of log, thus it makes for the easiest addition.
- Clearing the intent log is done asynchronously by a kernel thread.
  - on-disk representation is a bitmap with a single bit per chunk (with set being dirty).
  - This introduces a tuneable clearing delay to permit the device to get into its optimized working set.
- We also have an in-memory map with a counter per chunk.
  - A counter is required because the on-disk bitmap may only be cleared when all outstanding writes to the chunk have completed.

In-Memory Log

- The counter is sixteen bits wide (you'd have to have a huge clearing delay to overflow this).
- The memory used by the in-memory log is allocated lazily using a two level model.
- A page of counters (2048) is assigned to a superchunk.
- A superchunk may be either a pointer to page of chunk counters or a counter itself for the superchunk.
- Initially only space for the superchunks is allocated (for a 4kb underlying block size, one page of superchunks covers 8MB of disk space).
- chunk pages are allocated and deallocated on the fly. Even if the allocation fails, the system may still continue using the superchunk counter.


Assigning Counters to Blocks
Disk
Chunk
Block
Block
Block
Block
Chunk
Counters
Counter
Counter
High level counters
Page of Counters

Non-Volatile Intent Log

- Non-Volatile log is simply a file in the filesystem.
- As soon as counter increments, the chunk must be marked dirty in the log.
- The log may be cleaned as soon as the count drops to zero
- However, we delay the log cleaning in the hope that we'll get another I/O to that chunk.
- Clearing is done by a back end kernel thread
  - simple algorithm: sweep over every five minutes and clear all the necessary bits.
  - Could make this more intelligent (something like LRU clearing for the bits instead of simple clearing).

Asynchronous Replication

- This is very easy. You simply have a global counter for the in-flight chunks.
- Maximum possible value is set so that per-chunk counter **cannot** overflow (i.e. 65535 for 16 bit counters).
- This rule keeps the two-level counters working correctly at all times
  - you **know** that the sum of the counters in a page cannot overflow 16 bits

Intelligent Switch Back

- Since intent logs are simple files, when the primary comes back you can locate the old primary log
- From user space, you can combine (or operation) the primary and replica logs.
- Once the combination is complete you can replay from the primary to the secondary using the combined log (simply bring the mirror up with this log).
- All of this is done from user space—no kernel help required.

Conclusions

- Asynchronous and Non-Volatile intent logging is eminently doable and makes possible both cheap HA clustering and Disaster Recovery.
- The code should be going across the linux-raid mailing list "any day now".
- Code is all open sourced and will be usable in any HA/DR project.