Implementing Clusters for High Availability

James E.J. Bottomley
SteelEye Technology, Inc.
James.Bottomley@steeleye.com

Abstract

We explore the factors which contribute to achieving High Availability (HA) on Linux, from intrinsic cluster type to those which lengthen the application's uptime to those which reduce the unplanned downtime.

1 Introduction

The venerable Pfister [1] gives a very good survey of the overall state of clustering on commodity machines. From his definitions, we will be concentrating exclusively on High Availability (HA) and excluding any form of clustering to achieve greater computational throughput (Usually referred to as High Performance Computing [HPC]).

The type of HA cluster plays a role in cluster selection (see section 2) since that governs its speed and recoverability but the primary thing to consider is service availability: availability is often measured as the ratio of down time[1] to up time [2]. Thus, in its crudest sense, High Availability is anything that increases Availability to a given level (often called the class of nines).

There are two ways to increase Availability: improve up time and reduce down time. The former can often be achieved by carefully planning the implementation of your application/cluster. The latter often requires the implementation of some type of clustering software.

So, the real question is what do you need to do to increase Availability.

1.1 Class of Nines

When the Availability of a cluster is expressed as a decimal (or a percentage), the number of initial leading nines in the figure is referred to as the "Class of Nines"; thus

- 0.99987 is class 3 (or 3 nines)
- 0.999967 is class 4 (or 4 nines)

and so on. Each class corresponds to a maximum allowable amount of down time per year in the cluster:

- class 3 is no more than 8 hours, 45 minutes
- class 4 is no more than 52 minutes
- class 5 is no more than 5 minutes, 12 seconds

1.2 The Paradigm for a HA Cluster

The standard template for a HA cluster is shown in figure 1; it basically consists of multiple redundant networks (so that heartbeats between nodes don't fail because of network problems), a set of commodity computing hardware (called the nodes) and some type of shared storage.

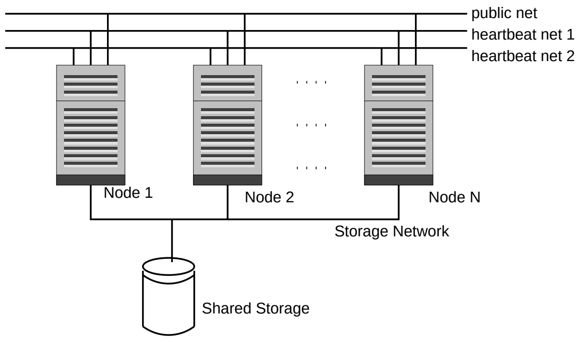

Figure 1: *A Standard Cluster*

2 Types of HA Clusters

The HA cluster market is split broadly into three types:

1. Two Node Only
2. Quorate
3. Resource Driven

The first (Two Node Only) describes any type of cluster whose method of construction does not allow it to expand beyond two nodes. These clusters, once the mainstay of the market, are falling rapidly into disuse. The primary reason seems to be that even if most installations only actually use two nodes for operation, the ability of the cluster to expand beyond that number gives the operator the capacity to add extra nodes at will (whether to perform a rolling upgrade of the cluster hardware, or simply to expand the number of active nodes for greater performance).

2.1 Quorate Clusters

This is often regarded as the paradigm of HA. It describes the cluster mechanism originally employed by the Digital's VAX computers. (The best description is contained in the much later openVMS documents [3]). The key element here is that when a cluster forms, it establishes the number of votes each cluster member has and compares that against the total available votes. If the forming cluster has under half (the quorum) it is unable to perform any operations and must wait until it attains over half the available votes (becomes quorate). Often votes are given to so called "tie break" resources like discs so that the formation of the cluster may be mediated solely by ownership of the tie-breaker resources.

The essential operational feature here is that the cluster control system must first fully recover from the failure (by establishing communication paths, cluster membership, voting and so on) before it may proceed to direct resource recovery.

2.2 Resource Driven Clusters

This type of clustering is not very well covered in the literature, but it has been in use in clustering technologies for the last twenty years. The key element is to divide the resources protected by a cluster into independent groupings called hierarchies. Each hierarchy should be capable of operating independently without needing any other resources than those which it already contains. When an event occurs causing the cluster to re-establish, each node calculates, for each hierarchy, based on the then available communications information whether it is the current master (i.e. it has lost contact with all nodes whose priority is higher for that hierarchy). If the node is the current hierarchy master, it immediately begins a recovery. In order to prevent contention, each hierarchy must contain one own-able resource (usually disk resources), ownership of which must be acquired by the node before hierarchy recovery may begin. In the event of an incomplete or disrupted communications channel, the nodes may race for ownership, but only one node will win out and recover the hierarchy.

The essential operational feature is that no notion of a "complete" cluster need be maintained. At recovery time, all a node needs to know is who is who is preferred over it for mastering a given hierarchy. Operation of a resource driven cluster doesn't require complete communications (or even any communication at all) since the ownership of the own-able resources is the ultimate arbiter of every hierarchy.

2.3 Comparison of Quorate and Resource Driven

The resource driven approach has several key benefits over the quorate approach:

1. the cluster layer can be thinner and simpler. This is not a direct advantage. However, the HA saying is "complexity is the enemy of availability", so the simpler your HA harness is, the more likely it is to function correctly under all failure scenarios.
2. recovery proceeds immediately without waiting for a quorum (or even a full communication set) to form.
3. Recoveries on different nodes, by virtue of the independence of the hierarchies, may be effected in parallel leading to faster overall cluster recovery.
4. May form independent subclusters: In the case where a cluster is totally partitioned, both partitions may recover hierarchies in a resource driven cluster; in a quorate cluster, only one partition may form a viable cluster.
5. recoverability is possible down to last man standing: As long as any nodes remain (and they can reach the resources necessary to the hierarchy) recovery may be effected. In a quorate cluster, recovery is no longer possible when the remaining nodes in a cluster lose quorum (either because too many votes have been lost, or because they can no-longer make contact with the tie breaker).

There are also several disadvantages:

1. For the paradigm to work, own-ability is a required property of at least one resource in a hierarchy. For some hierarchies (notably those not based on shared discs, like replicated storage) this may not be possible.
2. Some services exported from a cluster (like thinks as simple as cluster instance identity number) require a global state which a resource driven cluster does not have. Therefore, the cluster services API of a resource driven cluster is necessarily much less rich than for a quorate cluster.
3. The very nature of the simultaneous multi-node parallel recovery may cause a cluster resource crunch (too many things going on at once).
4. Since each node no-longer has a complete view of the cluster as a whole, administration becomes a more complex problem since the administrative tool must now build up its own view of the cluster from the information contained in the individual nodes.

However, the prime advantages of simplicity (Less cluster glue layer, therefore less to go wrong in the cluster program itself) and faster recovery are usually sufficient to recommend the resource driven approach over the quorate approach for a modern cluster..

Some clustering approaches try to gain the best of both worlds by attaching quorum resources to every hierarchy in the cluster.

3 Availability

As we said previously, Availability is the ratio of uptime to uptime plus downtime. Improving availability means either increasing uptime, decreasing downtime (or both). It is most important to note that any form of fail-over HA clustering can only decrease downtime, it cannot increase uptime (because the failure will mostly be visible to clients). Thus, we describe how to achieve up and down time improvements.

3.1 Increasing Up Time

It is important to understand initially that *no* clustering software can increase up time. All they can do is reduce down time. Generally, there are four reasons for lack of up time:

1. Application failures: The application crashes because of bad data or other internal coding faults.
2. Server failures: The hardware hosting the application fails, often because of internal component failures, like power supplies, SCSI cards, etc.
3. controllable infrastructure failures: things like global power supply failure, Internet gateway failure.
4. uncontrollable failures: Anything else (fire, flood, earthquake).

3.2 Application Failures

These are often the most insidious, since they can only be fixed by finding and squashing the particular bug in the application (and even if you have the source, you may not have the expertise or time to do this). There are two types of failure

Non-Deterministic: The failure occurs because of some internal error depending on the state of everything that went before it (often due to stack overruns or memory management problems). This type of failure can be "fixed" simply by restarting the application and trying again (because the necessary internal state will have been wiped clean). Non-deterministic failures may also occur as a result of interference from another node in the cluster (called a "rogue" node) which believes it has the right to update the same data the current node is using. To prevent these type of one node steps on another node's data failures from ever occurring in a cluster, I/O fencing (see section 6 is vitally important.

Deterministic: The crash is in direct response to a data input, regardless of internal state. This is the pathological failure case, since even if you restart the application, it will crash again when you resend it the data it initially failed on. Therefore, there is no automated way you can restart the application—someone must manually clean the failure causing data from its input stream. This is what Pfister[1] calls this the "Toxic Data Syndrome".

Fortunately, deterministic application failures are very rare (although they do occur), so they're more something to be aware of than something to expect. It is important to note that nothing can recover from a toxic data transaction that the application is *required* to process (rather than one introduced maliciously simply to crash the service) since the application must be fixed before the transaction can be processed.

3.3 Server Failures

The easiest (although certainly not the cheapest) way to get better uptime is to buy better hardware: often vendors sell apparently similar machines labelled "server" and "workstation" the only difference between them being the quality of the components and the addition of redundancy features.

Server redundancy features can be divided into two categories: those which don't and do require Operating System support to function. Of those that don't:

Redundant Fans: Ironically in these days of increasingly reduced solid state components, we still rely on simple mechanical (and therefore prone to wear and failure) devices for cooling: fans. They are often the cheapest separate component of any system, and yet if anything goes wrong with them, the entire system will crash or, in the extreme case of an on-chip CPU fan burn it's way through the motherboard. The first thing to note is that a well engineered box should have *no* on-component fans *at all*. All fans should be arranged in external banks to direct airflow over heat-sinks. The arrangement of the fans should be such that for any fan failure, the remaining fans should be sufficient to cool the machine correctly until the failed fan is replaced.

Redundant Power Supplies: After fans, these are the next most commonly failing components. A good server usually has two (or more) separate and fully functional power supply modules arranged so that for any single failure, the remaining PSUs can still fully power the box.

Those requiring Operating System support are things like:

Storage Redundancy: Both via multiple paths to the storage and multiple controllers within the storage (see section 3.4).

Active Power Management: With the advent of ACPI, the trend is toward the Operating System managing power to the server components. In this scenario, it becomes the responsibility of the OS to detect any power failure and possibly lower power consumption in its system until the fault is rectified.

Monitoring: This is the most overlooked part of the whole Server Failure problem. However much expensive hardware you buy, undetected faults will eventually cause it to die, primarily because the hardware is engineered to withstand a single fault in any subsystem, but a second fault (which will eventually occur) is usually fatal. Therefore, if you are going to run your systems unmonitored, you might just as well have bought the cheaper hardware and let the HA harness take over on any single failure.

3.4 Eliminating Single Points Of Failure

Single Points Of Failure (SPOFs) are one of the keys to controlling uptime. Their elimination is also crucial in cluster components that the HA harness doesn't protect: most often the actual data storage on an external array.

External data protection can be achieved by RAID [4], which comes in several possible implementations:

1. **Software**: using the `md` (or possibly the `evms md` personality). This is the cheapest solution, because it requires no specialised hardware.
2. **Host Based RAID**: This is a slightly more expensive solution where the RAID function is supplied by a special card in the server. This can cause problems clustering though: only some of these cards support clustering in both the hardware and the driver, and even if the card supports it, the HA package might not.
3. **External RAID Array**. This is the most expensive, but easiest to manage solution: The RAID is provided in an external package which attaches to the server via either SCSI or FC.

A particular problem with both software and Host Based RAID is that the individual node is responsible for updating the array including the redundancy data. This can cause a problem if the node crashes in the middle of an update since the data and the redundancy information will now be out of sync (although this can be easily detected and corrected). Where the problems become acute is if the array is being operated in a degraded state. Now, for all RAID arrays other than RAID-1, the data on the array may have become *undetectably* corrupt. For this reason, only RAID-1 should be considered when implementing either of these array types.

Although RAID eliminates the actual storage medium of the data as a SPOF, the path to storage (and also the RAID controller for hardware RAID) still is a SPOF. The simplest way to eliminate this (applying to both software and host based raid) is to employ two controllers and two separate SCSI buses as in figure 2.

Hardware RAID arrays also come with a variety of SPOF elimination techniques, usually in the form of multiple paths and multiple controllers. The down side here is that almost every one of these is proprietary to the individual RAID vendor and often requires driver add-ons (sometimes binary only) to the Linux kernel[2] to operate.

3.5 Infrastructure Failures and Service Export problems

Another key problem to consider is "what exactly is the criterion for a service being available". In the old days, it was enough to know that the service was being run in the mainframe room to say that it was available. How-

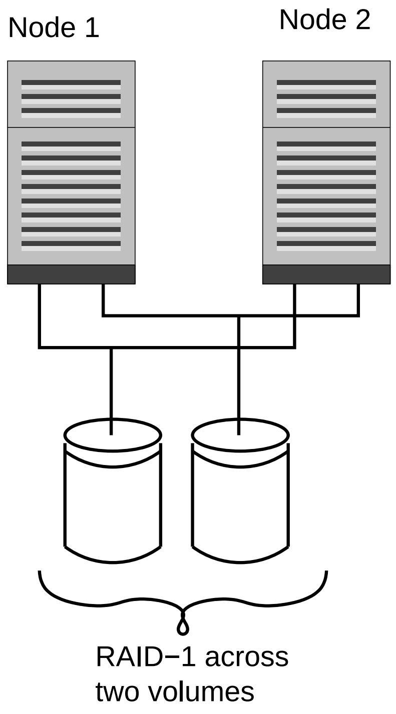

Figure 2: *Achieving no Single Point of Failure*

ever, nowadays, the service's users are more often than not remote from it over the Internet. Therefore, the availability of the service may be affected by factors beyond the control of a HA cluster.

To control vulnerability to these external factors, one must consider the SPOF reduction program as extending into the Internet domain itself: Your external router and your ISP may also be SPOFs, so you may wish to consider provisioning two of them. The expense of doing this for two full blown T1 or higher lines is likely to be prohibitive. However, one can consider the scenario where the primary Internet line is backed by a much cheaper alternative (like DSL or cable modem) so that if the primary fails, the service becomes degraded, but not non-functional.

Even within a cluster, it may be possible apparently to recover the service in a manner which makes it practically useless. For example, a web server exporting a service to the Internet should not be recovered on a node which cannot see the Internet gateway.

For this reason, a utility function per hierarchy could be calculated (measuring the actual usefulness of recovering the hierarchy on a given node) and taken into consideration when performing recovery.

4 Reducing Down Time

By and large this is recovering as quickly as possible from a failure when it occurs. In order to reduce the Down Time to a minimum, this recovery should be automated. This automation is often done by a High Availability Harness.

The cardinal thing to consider is the time it takes to restore the application to full functionality, which is given by:

$$T_{\text{Restore}} = T_{\text{Detect}} + T_{\text{Recover}} \quad (1)$$

The detection time, T_{Detect}, is entirely driven by the HA Harness (and should be easily tunable). The application recovery time, T_{Recover}, is usually less susceptible to tuning (although it can be minimised by making sure necessary data is on a journaling file-system for example).

4.1 Linux Specific Problems

One of the major problems with Linux distributions can be the sheer number of kernel's available (usually with distribution proprietary patches), so any HA package that depends on kernel modifications is obviously going to have a hard time playing "catch up". Thus, although kernel support may be standardised by the CGL specification [5], currently it is a good idea to find a HA package that doesn't require any kernel modifications at all (except possibly to fix kernel bugs detected by the HA vendor). Unfortunately, protection of certain services (like NFS) may be extremely difficult to do unaided; however, if your vendor does supply kernels or modules, make sure they have a good update record for your chosen distribution.

The greatest (and currently unaddressed) problem within the Linux kernel is the so called "Oops" issue where a fault inside the kernel may end up only killing the process whose user space happens to be above it rather than taking down the entire machine. This is bad because the fault may have ramifications beyond the current process; the usual consequence of which is that the machine hangs. Such hangs are inimical to HA software if they cause the machine to respond normally to heartbeats but fail (in a locally undetectable manner) to be exporting the service.

4.2 Replication

This is a useful technology both for Disaster Recovery and for shared storage elimination. Currently, Linux has two candidates for providing replication: `md/nbd` which places a RAID-1 mirror over a network block device[6] and is available in the kernel today and `drbd` which is available as a separate package[7].

Some of the cluster packages listed in the appendix can make use of replication for shared storage replacement.

4.3 2.6 Kernel Enhancements for HA

The most impressive enhancement in 2.6 (although, obviously this wasn't done exclusively for HA) is the improved robustness of the OS. It seems much less prone to emit the dreaded Oops (although when it does, it still erroneously tries to recover rather than doing fast failure).

The primary new availability feature is the proposed multi-path solution using the device mapper. Hopefully when this is implemented by the vendors it will lead to a single method of controlling and monitoring storage availability rather than the current 2.4 situation where each vendor rolls their own.

Finally, there are the indirect enhancements: those that improve Linux acceptance in the enterprise (where HA is often a requirement). Things like:

- Large Block Device (LBD) support, which allows block devices to expand beyond two terabytes.
- Large File and File-system support which takes advantage of LBD to expand file-systems (and files) beyond the two terabyte limit.

4.4 The HA Harness

Every piece of current HA software on the market is structured as a harness that wraps around existing commodity applications. This is extremely important point because the job of current clusters is to work with commodity (including software), so the old notion of writing an application to a HA API to fit it into the HA System simply doesn't fly anymore. This approach also plays into choosing a HA vendor: you need to choose one with the resources to build these harnesses around a wide selection of existing applications that you might now (or in the future) want to use.

Choosing such a harness can be very environment specific. However, there are several points to consider when making this choice.

- **Application monitoring**: All applications may fail (or even worse, hang) in strange ways. However, if the harness doesn't detect the failure, you won't recover automatically (and thus the down time will suffer).
- **In Node Recovery**: If an application failure is detected, can the harness restart it *without* doing a cross node fail-over. (The application and data are often hot in the node's cache, so local restarts can often be faster).
- **Common Application Protection**. HA packages usually require an application "harness" to interface the given application to the HA software. You should make sure the HA vendor has a good range of pre-packaged harnesses for common applications, and evaluate the vendor's ability to support

custom applications easily.

4.5 Considering More than Two Nodes

The availability defined in the introduction is simply

$$A = \frac{T_{\mathrm{Up}}}{T_{\mathrm{Up}} + T_{\mathrm{Down}}} \tag{2}$$

One would like simply to replace T_{Down} by T_{Recover} and have that be the new Availability. However, life isn't quite that simple. In an N node cluster, the Availability A_N is given by

$$\begin{aligned} A_N =& T_{\mathrm{Up}} \left(T_{\mathrm{Up}} + T_{\mathrm{Down}}(1-A)^{N-1} + \right. \\ & \left. T_{\mathrm{Recover}}(1-(1-A)^{N-1}) \right)^{-1} \end{aligned} \tag{3}$$

So if really high Availability values are important to you, more than two nodes becomes a requirement.

However, the most important aspect of more than two node support is the far more prosaic cluster operation scenario: as the number of services ($\equiv$ hierarchies) in your cluster increases, the desire to increase the computing power available to them usually dictates larger clusters with one or two services active per node.

5 Clusters and Service Levels

When all is said and done, anyone implementing a cluster has likely signed off on an agreement to provide a particular level of service. There are many ways to measure such a service, and it is important to consider what you really are trying to achieve before signing off on one.

5.1 Fault Tolerance v. Fault Resilience

Pfister[1] long ago pointed out that the tendency by the marketing departments to redefine HA terms at will makes Humpty Dumpty[3] look like a paragon of linguistic virtue. To save confusion, we will define:

Fault Tolerance to mean that any user of the service exported from the cluster does not observe any fault (other than possibly a longer delay than is normal) during a switch or fail over, and

Fault Resilience to mean that a fault may be observed, but only in *uncommitted* data (i.e. the database may respond with an error to the attempt to commit a transaction, etc.).

These distinctions are important, because it is possible to regard a fault tolerant service as suffering *no* down time even if the machine it is running on crashes, whereas the potential data fault in a fault resilient service counts toward down time.

5.2 Converting Fault Resilience to Fault Tolerance

Given the definitions above, it is apparent that the client the user employs to make contact with the service may also form part of the overall experience. Namely, if the client gets the observable failure, for example the error on transaction commit, but then itself simply retries the complete transaction (i.e. the client must be tracking the entire transaction) and receives a success message back because the service has been fully recovered, the user's experience will once again be seamless.

The moral of this is that if you control the construction of the client, there are steps you can take outside of the server's high availability environment that will drastically improve the users experience, converting it from one of Fault Resilience (user observes failure) to Fault Tolerance (user observes no failure).

5.3 Is it Availability you want?

The standard service level agreement is usually phrased in terms of availability. However, as we've seen, availability can be a tricky thing to determine and can also be very hard to manage since it depends on uptime which is outside the capability of any clustering product to control.

However, consider the nature of most modern Internet delivered services (the best exemplar being the simple web-server). Most users, on clicking a URL would try again, at least once if they receive an error reply. The Internet has made most web users tolerant of any type of failure they could put down to latency or routing errors. Thus, to maintain the appearance of an operational website, uptime and thus availability are completely irrelevant. The only parameter which plays any sort of role in the user's experience is downtime. As long as you can have the web-server recovered within the time the user will tolerate a retry (by ascribing the failure to the Internet) then there will be no discernible outage, and thus the service level will have met the user's expectation of being fully available.

In the example given above, which most user requirements tend to fall into, it is important to note that since uptime turns out to be largely irrelevant, then any money spent on uptime features is wasted cash. As long as the investment is in a HA harness which switches over fast enough, the cheapest possible hardware may be deployed. [4]

5.4 Important Lessons

The most important observation in all of this is that it is possible to spend vast amounts of money improving cluster hardware and uptime, and yet be doing very little to solve the actual problem (being that of your user's experience).

Therefore *before* even considering buying hardware or implementing a cluster, make sure you have a good grasp on what you're trying to achieve (and whether you can also make service affecting improvements in other areas—like the design of the service client).

6 I/O Fencing

Since clusters may transfer the services (as hierarchies) among the nodes, it is vitally important that only a single copy of a given service be running anywhere in or outside of the cluster. If this is violated, both of these instances of the service would be accessing and updating the same data, leading to immediate corruption.

For this reason, it is simply not good enough for a reformed cluster to conclude that any nodes that can't be contacted is passive and not accessing current data, the cluster must take action to ensure this.

A primary worry is the so called "Split Brain" scenario where all communication between two nodes is lost and thus each thinks the other to be dead and tries to recover the services accordingly. This situation is particularly insidious if the communication loss was caused by a "hang" condition on the node currently running the service, because it may have in-cache data which it will flush to storage the moment it recovers from the hang.

6.1 Stonith Devices: Node based fencing

Stands for Shoot The Other Node in the Head, and refers to a mechanism whereby the "other node" is unconditionally powered off by a command sent to a remote power supply.

This is the big hammer approach to I/O fencing. It is most often used by quorate clusters, since once the cluster membership is categorically established, it's a simple matter to power off those nodes who are not current members. Stonith is much less appropriate to resource driven clusters, since they often don't have sufficient information to know that a node should be powered off.

The main disadvantage inherent in stonith devices is that the situation in a split brain situation caused by genuine communications path failure, then the communication path to the remote power supply used to implement stonith is also likely to be disrupted.

6.2 Data based Fencing

Instead of trying to kill any nodes that should not be participating in the cluster, Data Fencing attempts to restrict access to the necessary data resources so that only the node legitimately running the service gets access to the data (all others being locked out)

Data based fencing gives a much more fine grained approach to data integrity (and one that is much better suited to resource driven clusters). Fencing is most often implemented as a lock placed on the storage itself (via SCSI reservations or via a volume manager using a special data area on storage). This means that if the node can get access to the data, it can also be aware of the locking.

The disadvantage to data based fencing is that it cannot be implemented in the absence of a storage mechanism that supports it (which occurs when the storage is replicated).

7 Conclusion

You can get a long way toward High Availability simply by taking steps to lengthen uptime. However, this doesn't protect against unplanned outages, so automation in the form of a HA harness is a prerequisite for this.

Knowing the right questions to ask when choosing a HA harness is often more important than the choice itself because it gives you a fuller understanding of the limitations of the system you will be implementing.

A Linux Cluster Products

Here we briefly summarise the major clustering products on Linux and their capabilities

A.1 SteelEye LifeKeeper

Closed Source, Resource driven cluster, scales to 32 nodes, includes active monitoring and local recovery. Uses SCSI reservations for Data Based fencing and also has support for Stonith Devices. Uses open source kernel modifications for HA NFS and data replication only (absent the requirement for these features, LifeKeeper will run on an unmodified Linux kernel). Supports replication using `md/nbd`.

`http://www.steeleye.com`

A.2 Veritas Cluster Server

Closed Source, Resource driven cluster, scales to 32 nodes, includes active monitoring and local recovery. Uses SCSI reservations or the Veritas Volume Manager for Data based fencing. Uses closed source kernel modules (which are only available for certain versions of Red Hat) for SCSI reservations and Cluster Communication. Cluster server will only run on a kernel with proprietary modifications. No support for replication on Linux[5].

`http://www.veritas.com/Products/van?c=product&refId=20`

A.3 Red Hat Cluster Manager

Open Source, Quorate cluster, scales to 6 nodes, limited active monitoring, no local recovery. Uses stonith devices for Node fencing. Uses open source kernel modifi-

cations (which are integrated into Red Hat Kernels only) to support HA NFS. No support for replication.
`http://www.redhat.com/software/rha/cluster/manager/`

A.4 Failsafe

Open Source, Quorate cluster, scales to 32 nodes, full active monitoring and local recovery. Uses stonith devices for Node fencing. Project has not been updated for a while. No support for replication.
`http://oss.sgi.com/projects/failsafe/`

A.5 Heartbeat

Currently Two node only. Uses other available components for active monitoring and local recovery. Uses stonith devices for Node fencing. Supports replication using `drbd`.
`http://www.linux-ha.org`

Notes

1. Often this excludes *planned* downtime

2. A framework for multiple paths to storage in the 2.6 kernel has been proposed, but so far there have been no implementors

3. “When *I* use a word”, Humpty Dumpty said, in a rather scornful tone “it means just what I choose it to mean—neither more nor less.”[8]

4. although the increased probability of failure of such hardware increases the probability of an unrecoverable “double fault” where both nodes in a two node cluster are down at the same time because of hardware failure.

5. Replication is available with the Veritas Cluster Server on non-Linux platforms.

References

[1] Gregory F. Pfister *In Search Of Clusters*, 1998, Prentice Hall.

[2] Matthew Merzbacher and Dan Patterson, *Measuring end-user availability on the Web: Practical experience*, Proceedings of the International Performance and Dependability Symposium (IPDS), June 2002, `http://roc.cs.berkeley.edu/papers/Merzbacher%20-%20Measuring%20Availability.pdf`

[3] Digital Equipment Corporation, *OpenVMS Clusters HandbooK*, Dcoument EC–H220793, 1993

[4] D. A. Patterson, G. A. Gibson, R. H. Katz *A Case for Redundant Arrays of Inexpensive Disks (RAID)* Proceedings of the International Conference on Management of Data (SIGMOD), June 1988, `http://www-2.cs.cmu.edu/~garth/RAIDpaper/Patterson88.pdf`

[5] Open Source Development Lab, *Carrier Grade Linux Requirements Definition*, version 2.0, Chapter 6, `http://www.osdl.org/lab_activities/carrier_grade_linux`

[6] J. E. J. Bottomley and P. R. Clements, *High Availability Data Replication*, Proceedings of the Ottawa Linux Symposium (2003) pp. 119–126

[7] Philipp Rensner, *DRBD*, `http://www.drbd.org`

[8] Lewis Carroll *Alice Through the Looking Glass*, 1872 `http://www.cs.indiana.edu/metastuff/looking/lookingdir.html`