Resource fencing using STONITH

Alan Robertson

IBM Linux Technology Center
13750 Bayberry Drive
Broomfield, Colorado, 80020

alanr@us.ibm.com

ABSTRACT. *Clusters of computers which share resources must take steps to protect the integrity of these shared resources (particularly shared storage) in the event of the failure of a node in the cluster. The term which is used for this protection is "fencing". There are classically two different fencing techniques which are commonly used. This article explains the rationale behind fencing, and presents the details of a technique which we refer to as STONITH (**S**hoot **T**he **O**ther **N**ode **I**n **T**he **H**ead), along with its use and implementation in the High−Availability Linux project.*

Rf SUMf . *L'ensemble des consignes rassemblées ci−dessous s'organise en trois rubriques. La rédaction remercie les auteurs pour le strict respect qu'ils accorderont à ces dispositions. La taille de ce résumé ne doit pas dépasser une dizaine de lignes. Il est à composer en Times corps 9 italique, interligné 11 points. Un résumé en anglais doit l'accompagner.*

KEY WORDS: *clusters, fencing, high−availability, I/O, integrity*

MOTS−CLÉS: *un maximum de mots significatifs, en français et en anglais, doivent etre isolés sous forme de mots−clés.*

---

1. Introduction

When a cluster of computers shares resources, there are certain rules which have to be followed regarding the sharing of these resources so that the integrity of these resources is protected. For example, a normal (non−cluster) filesystem cannot be mounted simultaneously on more than one node in the cluster at a time. When this violation of constraints occurs, the results are normally catastrophic − the data on the filesystem is generally destroyed in the process. This violates an important rule high−availability (and life as well): "First, do no harm" [PFI98]. Other types of resources (such as IP addresses) have similar integrity rules − even if the effects of the integrity violations are rarely as severe. Throughout the rest of this paper the examples will be largely shared disk examples, although the conclusions apply to other types of resources as well.

When clusters are operating normally, there are simple protocols which are designed to eliminate this possibility. One such mechanism is quorum − in which shared resources cannot be used without the agreement of the majority of the nodes in the cluster. The much more interesting case occurs when a cluster node which is using a shared resource dies (or appears to die).

At first glance it might seem to be sufficient for the cluster to realize that it still has a majority vote and assign the shared resource to another cluster member and continue from there. After all, if the cluster node is dead it isn't using any cluster resources. If it isn't actually dead, it will notice that it no longer has quorum, and stop using the resource and things will all work out. In practice, this method often works fine for a while in a properly configured cluster. It is unfortunate that it often works, because this can be confused with *reliably* working − which is quite a different matter. Given the severity of the damage done, it must *always* work − as recovering from loss of a filesystem is often difficult, time−consuming, and costly in lost business and customer confidence.

2. Why Quorum Isn't Enough

Quorum is a software−level protocol for sharing resources in a cluster which guarantees that resources are allocated correctly when all nodes in the cluster are operating normally. Unfortunately, it cannot guarantee that resources which were assigned to cluster nodes which are no longer accessible will be released in any particular time frame. When a node cannot be contacted, it is often informally referred to as being dead. However, the only thing which is actually known for certain about the node is that it is incommunicado. All else is speculation.

The problem is that the incommunicado node has shared resources, and we want it to give them up so the cluster can reassign them to another node or otherwise recover them so it can continue providing service. By definition, one cannot ask the node to give up the resource − all communications are inoperable, and indeed, the node itself is often quite sick, or dead.

It is occasionally asserted [LHA01] that once one has waited a period of time, that one can assume that the node is no longer using the resource because it is either dead or has given up the resource voluntarily through quorum. Although this may sound reasonable at first, there are many counterexamples.

For example, someone with administrative privileges or physical access to the machine can stop the machine from processing either deliberately, or accidentally. If a system programmer has entered the kernel debugger then received a phone call, or simply gone off to tea, and left the machine without restarting it − it will be

completely unresponsive for an indefinite period of time, and also quite capable of fully recovering. If there was I/O queued to the shared disk and the programmer resumes from the debugger, the kernel would typically write out this data immediately − before any user−level process is scheduled, and loss of quorum can be detected.

Severe resource shortages, like a "paging storm" can cripple a system indefinitely. Although operating systems try to limit system memory shortages, they sometimes fail to do so, and events like misbehaving applications or ongoing denial−of−service attacks can cripple a machine in a very similar way − with essentially identical results. The system is unable to do any useful work for an indeterminate period of time. If misbehaving applications are killed, or the denial−of−service attack stops, the system will continue on as before − and some amount of time will elapse before the system notices it has lost quorum. During this interval, data can be written to the disk.

It has also sometimes been asserted [LHA01] that if one controlled enough layers of the system software and made them all "cluster−aware" one could eventually reach the place where such effects could be eliminated by software alone.

However, without also controlling the hardware, one cannot assume that the disk controller doesn't have data queued up to the disk device, and is simply waiting for an interrupt to be serviced before writing it all to the disk. In this modern, open environment of off−the−shelf components, this assumption of dedicated hardware is usually inappropriate. In a highly dynamic open source environment like Linux™ [1], such an approach would seriously limit the usefulness of such solutions, in addition to being difficult to manage on an ongoing basis.

3. Resource Fencing

From this discussion, it should be apparent that software−only approaches to dealing with the problem are fraught with difficulties. The problem then is that one must quickly and reliably eliminate incommunicado nodes' access to shared resources without their cooperation. The generic term for this limitation of access is fencing −because hardware is used to build a conceptual fence between the node and its shared resources.

There are two basic approaches to fencing: resource−based fencing, and system reset (or STONITH) fencing. Resource−based fencing will be discussed first.

In resource−based fencing, a hardware mechanism is employed which immediately disables or disallows access to shared resources. If the shared resource is a SCSI disk or disk array, one can use SCSI reserve/release (or better yet persistent reserve/release operations). If the shared resource is a fiber channel disk or disk array, then one can instruct a fiber channel switch to deny the problem node access to shared resources. In general, the errant node itself is left undisturbed, and its resources are instructed to deny access to it. If the node is able to later become part of a cluster with quorum, it will then go through the normal channels to reacquire its resources.

This method has the following characteristics:

- It is minimally disruptive to the problem node
- It has to be implemented differently for each type of resource being protected (SCSI disk, fiber channel disk, IP address, etc.)
- It requires cooperation from the OS, the device drivers involved, (depending on the resource) the controller, and the device being protected.
- These interactions between application, OS, drivers, controller and resource

1 Linux is a trademark of Linux Torvalds

firmware are sometimes complex and difficult to manage. In most cases, each combination of hardware, firmware, drivers and software has to be extensively tested and certified as being mutually compatible. This is, of course, an ongoing expense.

- Booting of cluster members with shared root filesystems can be problematic. If the system is locked out of shared filesystems when it leaves the cluster, this can make it tricky to mount a shared root filesystem − even read only.

Resource−based fencing has often been used in proprietary clustering systems where the hardware vendor is able to constrain the number of different versions of drivers, firmware, software and disk units. Examples of systems which use this approach are IBM's HACMP, and Steeleye's LifeKeeper.

STONITH fencing takes a completely different approach. In STONITH systems, the errant cluster node is simply reset and forced to reboot. When it rejoins the cluster, it acquires resources in the normal way. In many cases, STONITH operations are performed via smart power switches which simply remove power from the errant node for a brief period of time. In other cases, built−in hardware (like Intel's IPMI) on the cluster nodes is used.

The STONITH method has these characteristics:

- It is highly disruptive to the problem node
- It is universal − it operates on all resource types equally well, and simultaneously
- It is very simple in concept and in practice
- There are virtually no support problems or version interactions to complicate development, testing and maintenance
- Overall system availability is often helped by the reboot

Examples of cluster systems which use STONITH methods include Linux−HA (heartbeat) [ROB00], SGI's FailSafe[SGI01], Mission Critical's Convolo cluster[BUR00], and Sistina's Global File System[2][PRE00].

It is worth mentioning some of the reasons why as of now the Linux−HA project has only implemented STONITH fencing:

- STONITH has a low development cost
- Since complexity is the enemy of reliability, STONITH's simplicity is seen as a great virtue
- Linux SCSI reserve/release support is immature and not consistently implemented
- The non−hierarchical nature of Linux development means ensuring consistency between the various components in the system is much harder. This is made worse by developers not appreciating the importance of these features.
- Linux−HA runs on many platforms, and reset mechanisms exist for most computers, and most STONITH implementations are common regardless of platform.

One of the more common objections to using STONITH is that administrators don't want their computers being rebooted automatically. This is dealt with by one the Linux−HA STONITH implementations. There is a version of a STONITH plugin for a manual reboot method − where an operator is informed of the need to reboot a node, and then certifies that they rebooted it. This gives the customer the ability to intervene before takeovers occur. This manual reboot plugin is whimsically called the "meatware" plugin.

2 Although the GFS project calls the method STO**M**ITH with **m**achine substituted for node.

4. STONITH in Two−Node Systems

It is commonly understood that two node systems are a special case in high−availability systems for two reasons:

- Simple quorum mechanisms require an odd number of nodes (although one can use a quorum device to help this out)
- The majority of high−availability systems are two−node systems, since this is the minimum size cluster required to provide failover

Two node systems are attractive and desirable, because they minimize cost and complexity, but quorum mechanisms don't work ideally with two−node systems. An interesting question is "What happens if you have a two node system using STONITH but no quorum mechanism?"

The following cases come up when one operates a two−node system equipped with STONITH, but without quorum:

- One node is operational, and one is disabled or dead
- Both nodes disabled or dead
- Both nodes alive, but incommunicado

Each of these cases will be considered in turn:

4.1 One node operational, one disabled or dead

This is the most common case in properly configured clusters. The operational node will STONITH the other node, which will attempt to reboot. If it is able to reboot, then it will often join the cluster and things will continue on in a reasonably normal way. Reboots clear up a multitude of software, firmware and hardware problems. Of course, if the node does not successfully reboot after being reset, then it will not rejoin the cluster.

4.2 Both nodes disabled or dead

This is the least interesting case. The cluster has suffered multiple simultaneous failures from which no automatic recovery is possible without extra hardware support (watchdog timers, etc.).

4.3 Both nodes alive and well, but incommunicado

This is the most interesting case. If the cluster is configured according to the accepted best practice of having multiple independent communications mechanisms [MIL99] (for example, serial and ethernet), then the system has suffered multiple failures. The cluster may or may not be able to recover from this. But, whether or not it can recover from these multiple failures, protection of the integrity of cluster resources is still of paramount importance. There are a few subcases to consider:

- Communication failure was caused by failure of something which is cured by a system reset
- Communication failure is not caused by something which will be helped by a system reset.

4.3.1 Communications fixed by a system reset

If the node which was reset was the one which had the fault leading to

communications failure, then things will recover and all is well. If the node which was reset was *not* the one with the communications fault, then when it reboots, communications will still be faulty. However, this time, the rebooted node will reset the other (faulty) node and communications will be restored. Although it took an extra reboot cycle to reset the proper node, at no time were the sharing rules for resources broken, so the integrity of the resources was preserved. In order to maximize the probability of resetting the right node in the first place, the FailSafe system has an interesting strategy. If a node wants to reset its peer and the other node in the cluster has been a member of the cluster longer than it has, it waits an extra period of time, to allow the other node the opportunity to reset it first. If a system has had stability problems, this increases the chances that the unstable node (the youngest node) will be reset by the stable node (the oldest node). This strategy is informally described as "the oldest node shoots first". This also has the effect of minimizing the probability that both nodes will shoot each other simultaneously.

4.3.2 Communications not fixed by a system reset

This case is the least satisfactory. In this case, the nodes enter into a behavior with one being reset, then resetting the other, which reboots and resets it and so on. This cycle will continue until the communications problem is resolved or a human being intervenes. This is not a very satisfactory behavior. However, there are several things to keep in mind about this situation:

- The integrity of the resources is still preserved
- Multiple simultaneous failures have occurred − this cannot always be recovered from by any cluster system.
- If the cluster is configured to communicate across those subnets which customers use to reach the cluster as well, then it probably cannot provide useful service until these networks begin operating again.
- If one established quorum through a quorum device (like a router or a shared disk), then service would also not likely be available in these circumstances (although the systems wouldn’t be rebooting each other).

This subcase illustrates a good reason why it is desirable to send cluster control traffic across multiple independent links. When practical, the author recommends that one of the links should be something simple like a serial link − because serial links have few points of failure. They don’t require external devices (like hubs) or additional power, and are a mature and highly reliable technology. As a practical matter, the author also suggests that one screw the cables into the connectors so they cannot fall out or be pulled out by mistake.

It appears from this analysis that one can operate a two−node cluster with STONITH and without a quorum mechanism without endangering the integrity of cluster resources. Earlier we established that quorum was not sufficient for a cluster. In the two−node case, it appears that it is also not necessary − at least for some definitions of the problem.

5. Power−Cycle STONITH and Redundant Power Supplies

As was mentioned, one of the most common implementations of STONITH temporarily removes the power from an errant node to force it to reboot. This is more complicated for resetting systems which have more than one power supply connection. This is quite common in servers chosen for high−availability applications. Many power controllers provide "off", "on" and "reboot" operations.

Normally, STONITH implementations simply use the reboot operation which causes the machine to be power cycled. This is preferable to using an "off" operation followed by an "on" operation. If Node "A" is in the process of resetting node "B", then it is possible that node "B" is simultaneously trying to reset node "A". If both succeed in their "off" operation, then neither will ever succeed in turning the other back on. Although techniques which help this situation were discussed before, they aren't perfect, and this is a highly undesirable state of affairs. The solution to this problem is for each power inlet but the last one issue an "off" command. On the last inlet, issue a "reboot" command. This way, if despite all precautions, both machines issue resets to each other simultaneously, they will both reboot, and not both stay down.

6. STONITH in Linux−HA

As was alluded to earlier, the Linux−HA STONITH system is a library which loads the desired plugin for the type of reset hardware. There are currently 10 different STONITH plugins. Each plugin operates a different type of reset device. One plugin operates a UPS, five operate different brands of power switches, one interfaces to the VACM cluster management software, one is the meatware plugin mentioned earlier, and two are primarily for use in test environments.

The Linux−HA STONITH plugin library was specifically designed not to be tied to any particular cluster system. At the time, the author was working on both the FailSafe and heartbeat clustering systems, and wanted it to be used in either system. Although the operations which have to be performed are quite simple, the API design proved more difficult than anticipated. The author takes this as evidence of the difficulty of designing good, widely−applicable APIs.

The STONITH API [ROB01] simply reflects reset hardware capabilities in a uniform way. It does not try to interact with cluster configuration databases, or provide more capability than the hardware provides. If the reset hardware is network aware, and accessible through the network, then that capability is reflected through the API. If the reset hardware is only accessible through a serial port which is connected to a single machine − that is also reflected through the API. This has made it easy to write new STONITH plugins, since the drivers are isolated from cluster administrative details, and simply reflect the capabilities of the existing hardware. Since the API is cluster independent (and indeed doesn't even depend on being in a cluster), it has been easy to integrate into various cluster frameworks.

7. Linux−HA STONITH futures

The STONITH implementation which Linux−HA developed has been a success as a reusable component. It has been adopted by three cluster systems and has had contributions from several companies and individuals. Although the API itself is reasonably good, there is a perfectly valid criticism of the API which will be remedied in future releases.

This difficulty relates to how the STONITH plugins are configured. In order for a plugin to create a STONITH object for the underlying hardware, it has to be given some configuration parameters. These are passed to the plugin as an opaque string. This is fine in some respects, because it is uniform, simple, and easy to implement. However, when a user interface program wants to configure a STONITH object, it either has to just hope the person can type in the string that's

needed (the plugin API helps with this), or have special case code in it for each type of STONITH plugin.

The first approach is unsatisfactory because the GUI cannot provide the user good feedback on the details − like the opaque string is an IP address, a login and a password, and validate each separately. So, much of the good a GUI can do for a user is defeated. The second approach basically defeats the purpose of the shared development plugin environment. It means that although projects can share STONITH plugins, they probably have to change their GUI for each and every STONITH plugin that is created − minimizing the effects of the code sharing.

So, we intend to take another approach in the future. It is our intent to enhance the API so that each plugin can be queried for metadata information which describes the parameters it needs for its configuration. User interface code could then query a plugin for its configuration metadata and construct an appropriate user interface dialog for configuring the plugin. This is similar in some respects to the approach used in the SANE (a Linux−based scanner system) for configuring its scanner plugins. The interesting part about this solution is that the solution (creating metadata, etc.) is not at all specific to STONITH plugins, but occurs in many contexts − particularly those where plugins are a key part of the architecture.

The Linux−HA STONITH implementation has proven itself to be a reasonably general cluster component, which allows reasonably unrelated open source projects to share development and testing costs effectively. This has inspired a group in the open source area to try to extend this idea into a complete framework of reusable general cluster components. This project (which is just getting organized at the time of this writing) is called the Open Cluster Framework Project[COO01].

8. Conclusions

Clustered computers need to protect the integrity of shared resources. Quorum is a commonly used technique[TWE00], but must be used in combination with resource fencing. Resource−based fencing is commonly used, but STONITH fencing is more commonly used in the Linux community because of the volatility of the Linux environment. STONITH fencing is simple to implement, and widely applicable to a variety of environments and computers. The use of STONITH as a fencing mechanism was discussed in detail, with special attention given to the two−node system. The STONITH implementation in Linux−HA is based on a general plugin architecture, and has proven very successful as an implementation, and as an open source project. The architecture used in the Linux−HA STONITH subsystem is being extended into a broader scope − the Open Cluster Framework Project.

Acknowledgments

The author would like to thank David Brower for the excellent discussions on fencing which he led in the linux−ha mailing list, and the many contributors of STONITH methods: Mike Ledoux, Todd Wheeling, Andreas Piesk, Gregor Binder, Eric Z. Ayers, Joachim Gleissner and Michael C Tilstra. The author would also like to thank Michael Brown for suggesting a straightforward solution to the redundant power supplies problem.

9. Bibliography

[BUR00] Burke, T., "High−Availability Cluster Checklist", *Linux Journal*, no. 80, December, 2000.

[COO01] Cook, F., "Linux High−Availability Working Group", *Linux Weekly News OLS 2001 Feature Article*, 25 July, 2001, http://lwn.net/2001/features/OLS/linuxha.php3.

[LHA01] High−Availability Linux Project Members, *Linux−HA Mailing List Archives*, http://marc.theaimsgroup.com/?l=linux−ha&r=1&w=2

[MIL99] Milz, Harald, "The Linux High Availability HOWTO". http://metalab.unc.edu/pub/linux/ALPHA/linux−ha/High−Availability−HOWTO.html

[PHI98] Pfister, Gregory F., *In Search of Clusters*, 2nd Edition, Prentice Hall PTR, 1998

[PRE00] Preslan, K., et al., "Scalability and Failure Recovery in a Linux Cluster File System", 4th Annual Linux Showcase and Conference, Atlanta, Georgia, October 10−14, 2000, p. 169−180.

[ROB00] Robertson, A., "Linux−HA Heartbeat System Design", 4th Annual Linux Showcase and Conference, Atlanta, Georgia, October 10−14, 2000, p. 305−316.

[ROB01] Robertson, A., "Linux−HA APIs", presented at the LinuxWorld Conference and Expo, New York City, New York, Jan 30−Feb 2, 2001. http://linux−ha.org/heartbeat/LWCE−NYC−2001/index.html

[SGI01] SGI, Inc., "The Linux Fail Safe Project", http://oss.sgi.com/projects/failsafe/

[TWE00] Tweedie, S. C., "Quorum Operations", http://linux−ha.org/PhaseII/WhitePapers/sct/quorum.txt