

Clones, Failovers and Migrations

Background

Whilst working at a customer site, I recently had a requirement to design a mechanism whereby a very large legacy database server could be failed over from one site to another. Over the years several methods of doing so had been investigated, and subsequently abandoned for various financial, technical and political reasons.

Years after the initial service was implemented, the system was still a major single point of failure for the customer, and scheduled to still be in active duty for anything up to eighteen months.

During this time, the size and criticality of the system meant that the risk of implementing a creative solution was deemed to be inappropriate, until a single, significant event forced the client to reassess the situation.

The solution had to be constructed around the existing system, keeping change to a minimum, and absolutely avoiding any direct interference with the database and application.

We had a healthy budget to implement, however, we felt that a competent solution could be delivered whilst utilising a fraction of the available funds.

This article is both a quick discussion on the available technologies used during this exercise, and the methods employed to ensure consistent and reliable failover.

Legacy Systems

We briefly considered constructing a solution based on a clustering product (Veritas Cluster Server is heavily used by this particular client, and would be the product of choice).

VCS is a great framework for encompassing the complicated dependencies for starting/stopping and monitoring applications. As an aside, I notice that the Solaris 10 SMF provides a similar, perhaps an even better framework for this task, albeit without the multihost capabilities of VCS – perhaps an interesting future direction.

This particular 64-way E10k has been around for ages. It has been upgraded, broken, fixed, patched, unpatched, crashed and recovered – it even has custom written kernel patches provided by Sun because we are running it beyond the designed capabilities of the somewhat geriatric Solaris 2.6.

Regardless of whether or not we ended up using VCS (as it happens, we didn't, opting instead for a manual ten-step failover procedure which could be invoked by any local sysadmin), the biggest concern that I had whilst considering this project was to ensure that the contingent host had a suitable and accurate replica Operating System, and that once we had cloned the system, future changes to the production system were replicated accurately.

Basic Failover Mechanics

There are two basic infrastructure requirements for a successful failover:

1. IP Connectivity

Great flexibility can be gained by the use of DNS for providing a bridge between a service name and an IP address that may potentially change.

There is a significant drawback though – for the entire history of the system, the service name (i.e. DNS CNAME record used to signify the application interface) must be used by all parties.

On the face of it, this sounds to be a relatively straightforward and sensible assertion (“why would it be any other way?”, someone once asked me) – however, given the wide-reaching use of network services and firewalls, the hundreds of developers, administrators and third parties that connect to the system, this is not something we can often guarantee on a legacy system.

Instead, we are restricted to ensuring that the same IP address can be used for production and standby nodes. In order to achieve this, we examined two possibilities:

- The use of Cisco LocalDirectors in order to distribute inbound connections to the correct node;
- Migration to a cross site VLAN. The subnet we were originally installed on was available only on the production site. Another recent project had required the use of a number of cross-site VLANs that we could make use of.

2. Storage Availability

Even as far back as the days of SCSI direct attached storage, we could configure arrays with multiple initiators on the bus in order to provide a resemblance of failover capability – this didn't address failures or outages influenced by locality (i.e. entire datacentre outages).

Only when storage systems began to move to fibre did we really see this sort of thing take off in a big way. Initially, we had SSA's and A5x00 using longwave GBICs providing distant connectivity.

These days, large volume corporate data storage is almost exclusively in the arena of proprietary SAN attached arrays. All the major vendors have their own products that provide remote site data replication.

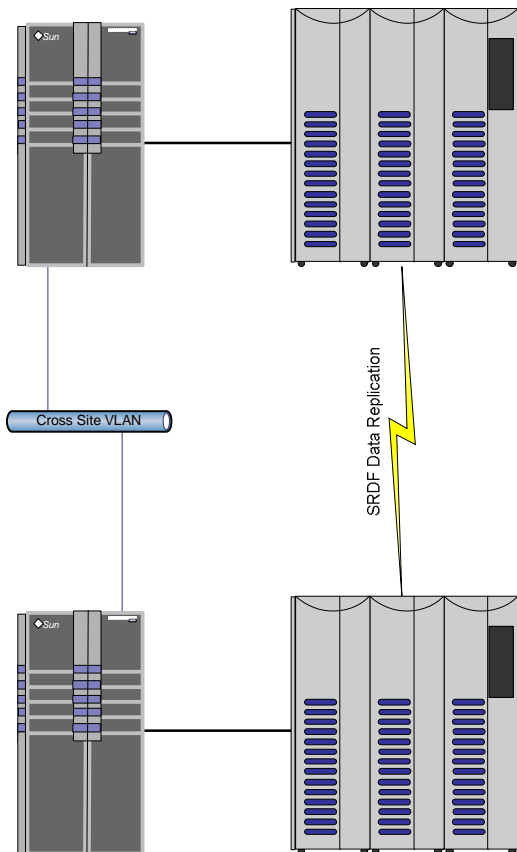
These proprietary hardware solutions generally use dedicated fibre links, and have a suitably high price tag (in terms of both capital and operational costs) associated with them to put off everybody except those with very specific requirements.

Alternative, host based solutions are available – these generally run as part of the operating system, and make use of network connectivity to transfer the data from one system to another.

Veritas Volume Replicator (VVR) is one such product – it integrates with Veritas Volume Manager to provide volume-level synchronisation. I also stumbled across an interesting concept of “Network Block Devices” on Linux (see <http://nbd.sourceforge.net>), which, as its' name suggests - a standard block device can be made available over a network link. This is a remarkably simple concept that makes remote mirroring both a trivial task and one that can be done with a normal local volume manager.

Primary Site : 64-way E10k, Solaris 2.6

EMC Symmetrix 8730



Standby Site : 48-way E10k, Solaris 2.6

EMC Symmetrix 8730

Anyway – this is an interesting distraction, however the real core of the failover procedure is the part that ensures that the standby OS configuration is as close to production as possible, and we do that by cloning the operating system, and holding it in sync while it is in standby.

In order to overcome the potential for service failure when the service is migrated to the standby host, we need to ensure that even the application is unaware which node upon which it is running.

We therefore set out to replicate the entire OS via Rsync (over SSH). This sounds easy in principle, but there are some parts of the OS that you cannot replicate (think about what might happen if you accidentally copied "/dev" from one machine onto another – would you expect it to work when you couldn't guarantee identically configured hardware?).

Cloning the system

The first step in creating the standby server is to replicate it completely.

We start the process by creating a new root filesystem. In order to do this, the domain is booted from the network, and the filesystem created, as one might expect using newfs (assuming c0t0d0 as your boot device):

```
# newfs /dev/zrdsk/c0t0d0s0
# mount /dev/dsk/c0t0d0s0 /a
```

Ufsdump/ufsrestore is then used to copy the root filesystem across. In this example, the server has no other OS filesystems – if we did have separate /var, /usr, /opt, or other filesystems, then these would have to be copied in the same manner.

On the production machine:

```
# ufsdump 0f /somewherebig/root.dump /
```

[where "somewherebig" is a filesystem suitably sized to take the entire ufsdump]. The dump was then transferred to the destination machine, and applied to the newly created filesystem. For my purposes, sharing the "/somewherebig" filesystem via NFS met my requirements, and allowed the dump to be extracted.

Storing the ufsdump (and not, for example, piping the dump directly across the network to a ufsrestore running on the other node) also provided the additional benefit of a point-in-time backup that could be re-used, should the configuration on the contingent node go disastrously wrong.

```
# mkdir /tmp/nfs
# mount -o ro production:/somewherebig /tmp/nfs
# cd /a ; ufsrestore rf /tmp/nfs/root.dump
```

At this point we need to take care before attempting to boot. There are four immediate issues that need to be addressed:

- IP addressing – if we're on the same VLAN as the production machine, then remember that we've just copied across the "/etc/hostname.*" files, and booting that image may result in a production threatening IP clash. Personally, I chose to remove all /etc/hostname.* files, and work solely through the console until the system was ready to boot independently;
- You ought to have your boot device mirrored, either by SVM or VxVM (as is the case in our example). The Solaris image that was just copied onto the standby contains the configuration for a mirrored/encapsulated rootdisk, which must be manually backed out in order to boot the environment (we can then later re-encapsulate/mirror).
- The device tree – (/dev and /devices, and /etc/path_to_inst). Unless you are in the fortunately position to have an absolutely **identically** configured contingent system, your device tree is going to have to change. To ensure a clean build, we bootstrapped a completely new tree;
- Bootblock – we've manually created a root filesystem, and will have to ensure that we also install a suitable bootblock onto the device.

Root disk unencapsulation

In order to make the system bootable, we must remove the configuration of the volume management software that was (at the time of the ufsdump) managing the root filesystem. For reference, here is the procedure for removing either VxVM or SVM from a rootdisk.

VxVM:

```
# cd /a/etc/
# cp vfstab vfstab.bak
# vi vfstab
< edit the vfstab, and ensure that all OS filesystems refer to the underlying devices >
# cp system.bak
# vi system
< remove or comment out the two entries:
rootdev:/pseudo/vxio@0:0
set xio:vol_rootdev_is_volume=1
>
# cd vx/reconfig.d/state.d
# touch install-db
# rm root-done
```

SVM:

```
# cd /a/etc/
# cp vfstab vfstab.bak
# vi vfstab
< edit the vfstab, and ensure that all OS filesystems refer to the underlying devices >
```

```
# cp system.bak
# vi system
< remove or comment out the entry:
rootdev: /pseudo/md@0:0,1,blk
>
```

Bootstrapping a new device tree

A simple “boot -r” should be able to boot the system and rebuild the device tree to the spec of the new server. However, there are a number of disadvantages to this simple approach - some of which are technical issues, other administrative:

- your controller numbers will not start at “c0”, “c1”, etc. Instead they will generally start numbering where the source machine left off. On a large server this could mean that the controllers start off in the twenties (“c21”, “c22”). Although this doesn’t affect the running of the server, it makes the system configuration less irregular, and therefore easier to support).
- Depending on storage configuration and OS revision, particular circumstances may arise (using same disk controllers in the same hardware device path), you may be in danger of exceeding LUN-per-controller limit (this was a particular concern in this example)

So, the theory is simple: we want to clear out /dev and /devices (and preferably /etc/path_to_inst). Trouble is, if we simply delete these files, the system will be unbootable.

Instead, while we are still booted into single-user mode from the network, we have to erase the old device trees and generate new ones, and there is a specific trick for the path_to_inst.

So, take a deep breath and delete the old trees...

```
# rm -rf /a/dev/* /a/devices/*
# echo '#path_to_inst_bootstrap_1' >/a/etc/path_to_inst
```

Notice that last command – we’re overwriting /etc/path_to_inst with a very particular string. Simply emptying this file, and expecting a reconfiguration won’t work (even with a “boot -a”, which **appears** to suggest that it can rebuild a missing path_to_inst) – replacing the file with this token will cause the machine to generate a new file from scratch. A good example of this in action is on a jumpstart boot image.

We can then generate the new trees:

```
# drvconfig -r /a/devices
# devlinks -r /a
```

Remember that we’re working on a legacy system here – on newer revisions of Solaris devfsadm is the preferred method of doing this.

And, all being well – you should now have a reasonably complete device tree, all that remains is to install a new bootblock and perform a reconfiguration reboot to check:

```
# installboot /usr/platform/`uname -i`/lib/fs/ufs/bootblk /dev/rdisk/c0t0d0s0
# reboot -- -rs
```

If all has gone to plan, then the domain should reboot, building a new path_to_inst and updating /dev and /devices as it goes. It ought to end up in single-user mode, whereby the remaining configuration (remirror rootdisk, set up networks) is left as an exercise to the reader.

OS syncing

After briefly considering the available options, I settled on rsync as the file synchronisation tool of choice. Rsync is a great tool for this sort of thing – it’s like an enhanced “rcp” or “scp”. It’ll handle just about anything you care to throw at it, including sparse files, device files, named pipes, on-the-fly compression and its’ particularly nifty party piece is block-level incremental transfer – that means where “rdist” will check a set of files and only transfer that subset that have

changed, rsync will go one step further, and transfer only those parts of the files that have been altered.

It’s tempting to have rsync simply synchronise the whole of the root filesystem, however, that will clearly overwrite all the good configuration that we have just done, and also have profound effects on the bootability of the machine as it overwrites “/dev” and “/devices” (a point to note – later versions of Solaris have a separate pseudo filesystem for “/dev”..).

The ideal method seemed therefore to provide both an “include” and “exclude list”, those files/directories listed in the “include” list will be synchronised recursively, except when a match is made (via rsync’s “--exclude” flag).

The include/exclude settings in the script given will synchronise the whole of the root filesystem, and exclude particularly irrelevant parts (/dev, /devices).

The OS configuration starts to get complicated under “/etc”, so I’ve deliberately excluded that and explicitly stated (via the “include” list) the files and directories underneath /etc that I want to keep synchronised. One day perhaps I’ll find the time to explore and map out the contents of “/etc” more accurately.

```
01
02 #!/usr/bin/ksh
03 # Rsync replication script
04 # Mike Scott
05 # 25/10/2005
06
07 # We should only attempt to sync if:
08 # * The destination IP address is pingable
09 # * There is a SSH connection
10 # * The vx diskgroup is locally imported (if not,
11 #   then it's possible that we're in the failed over state)
12
13 # Remote destination hostname
14 HOSTNAME="standby_bak"
15 VXDG_NAME="database"
16
17 LOGFILE="/failover/logs/rsync.${HOSTNAME}$(date +%d)'")
18
19 exec 2>&1
20 exec >>$LOGFILE
21
22 #####
23 # Supporting functions
24 timestamp() {
25     date +%d/%m/%y %H:%M:%S"
26 }
27
28 log() {
29     if [ $# -gt 0 ] ; then
30         echo $(timestamp) $*
31     else
32         while read line
33         do
34             echo $(timestamp) $line
35         done
36     fi
37 }
38
39 gameOver() {
40     log "=====
41     log " ERROR: Cannot Sync:"
42     log "      $*"
43     log "=====
44     echo logger -p user.err "/failover/rsync: $*"
45     exit 1
46 }
47
48 #####
49 # Prereq: vxdg is imported
50 log Checking Veritas Diskgroup
51 /usr/sbin/vxdg list $VXDG_NAME >/dev/null 2>&1
52
53 if [ $? -ne 0 ] ; then
54     gameOver "$VXDG_NAME Veritas Diskgroup not imported"
55 fi
56 log Veritas Diskgroup ok
57
58 log Checking failover host IP
59 ping $HOSTNAME 2 >/dev/null 2>&1
60 if [ $? -ne 0 ] ; then
61     gameOver "PING to $HOSTNAME Failed"
62 fi
63 log Failover host IP ok
64
65 log Checking failover host SSH
66 /usr/local/bin/ssh $HOSTNAME hostname >/dev/null 2>&1
67 if [ $? -ne 0 ] ; then
68     gameOver "SSH to $HOSTNAME Failed (rc=$?)"
69 fi
70 log Failover host SSH ok
71
72 log Checking Symmetrix SRDF State
73 # Note: we are assuming here that the Symmetrix Device group name
74 # is the same as the Veritas Disk Group name.
75 /usr/symcli/bin/symrdf -g $VXDG_NAME -synchronized verify >/dev/null
76 2>&1
77 if [ $? -ne 0 ] ; then
78     gameOver "SRDF is not fully synchronized for the $VXDG_NAME group"
79 fi
80 log Symmetrix SRDF State ok
81
82 RSYNC=/usr/local/bin/rsync
```

```

83  OPTIONS="--vtaz --one-file-system --rsync-path=$RSYNC --delete --
    rsh=/usr/local/bin/ssh"
84
85  # Here's the list of files to include and exclude
86  # Note that in this context a "file" means either
87  # a regular file or a directory
88  # If you are referring to a directory, then include a
89  # trailing "/"
90
91  INCLUDES="
92  /
93  /etc/init.d/
94  /etc/rc0.d/
95  /etc/rc2.d/
96  /etc/rc3.d/
97  /etc/rcS.d/
98  /etc/lp/
99  /etc/inet/
100 /etc/net/
101 /etc/opt/
102 /etc/ssh/
103 /etc/profile
104 /etc/passwd
105 /etc/shadow
106 /etc/group
107 /etc/resolv.conf
108 /etc/printers.conf
109 /etc/vfstab
110 /etc/system"
111
112 EXCLUDES="
113 /etc/
114 /kernel/drv/sd.conf
115 /dev/
116 /devices/
117 /proc/
118 /var/adm/sa/
119 /usr/emc/API/symapi/db
120 /usr/emc/API/symapi/log"
121
122 # Add exclusions to options list - I could have added them directly to
    the variable,
123 # but I felt that having an INCLUDE and EXCLUDE list as above was
    slightly clearer
124
125 for EXCLUSION in $EXCLUDES
126 do
127     OPTIONS="$OPTIONS --exclude=$EXCLUSION"
128 Done
129
130 log "Diskgroup configuration:"
131 VXVM_DG=$( /usr/sbin/vxdisk -g $VXDG_NAME -q list |wc -l|awk '{print
    $NF}' )
132 SYMM_DG=$( /usr/symcli/bin/symdg show $VXDG_NAME |awk '/Number of STD
    Devices in Group/ {print $NF}' )
133
134 log "NOTE: Number of Veritas Disks in $VXDG_NAME group : $VXVM_DG"
135 log "NOTE: Number of Symmetrix Disks in $VXDG_NAME group : $SYMM_DG"
136
137 /usr/symcli/bin/sympd list | log
138 /usr/symcli/bin/symrdf -g $VXDG_NAME query |log
139
140 log ""
141 log "Rsync Options: $OPTIONS"
142 log ""
143 log "Rsync startup"
144 ERROR=0
145
146 for FILE in $INCLUDES
147 do
148     echo Rsyncing $FILE to $HOSTNAME
149     echo -----
150     $RSYNC $OPTIONS $FILE $HOSTNAME:$FILE
151     ERROR=$(( $ERROR + $? ))
152 done |log
153
154 if [ $ERROR -gt 0 ] ; then
155     gameOver Rsync Failure
156 else
157     log "-----"
158     log " Rsync complete"
159     log "-----"
160 fi

```

Frequency

Whilst using the proposed procedure for synchronising a host, some thought must be paid to the frequency of the sync process. Not often enough, and you may be missing vital configuration in the event of a disaster – too frequent, and you may be compromising some of the functionality of a failover system.

For example, imagine what may happen if you had a filesystem corruption that was copied across to the failover host – you might not be able to **prevent** this from happening, but you can reduce the odds of it causing you an issue on both hosts by reducing the frequency of updates. Perhaps different subsets of files should be copied at differing frequencies.

For our situation, we chose to split the frequency. As we were not using network based authentication, it is important to keep the password, shadow and group files as up-to-date as possible (the machine is in 24x7 interactive use, with in excess of 16,000 entries and anything up to 2500 concurrent telnet sessions). These files are sync'd once an hour, ensuring that in the event of a failover, as many

user accounts will be completely up-to-date (with respect to account creations, deletions and password changes).

Everything else was then set to sync once daily – this caught any other configuration issues (print queues, cron changes, etc).

Failover Execution

Due to the very high profile of our specific situation, any service affecting issue would be immediately noticed by our on-site operations staff and escalated to technical support.

Depending on the nature of the issue, the problem ought to be analysed and discussed by the teams involved. The goal of this initial analysis is to ascertain the estimated time-to-recover using both a fix-on-fail or failover strategy and to determine which approach should restore service earliest with the least risk.

Summary

An "ideal world" is usually a faraway proposition for legacy systems that have evolved over a period of many years. Many teams of SysAdmins, DBAs, App Support and developers will have been involved, perhaps adding up to hundreds of personnel, each of whom have the capability to break a failover capable system by hardcoding, misconfiguration, or just plain old error.

One of the difficulties of having a failover capable system is that unless a regular test of the failover solution is organised (with customer expectations set that problems may be found, and opportunity must be given to fix), then any inability to successfully failover will only be found at the absolute worst possible opportunity.

The discussed procedure is a useful one, it treats the operating system as a filesystem with "just" a bunch of files [JBoF :-)], a commodity viewpoint whereby the initially complex problem becomes trivial.

I've also used this procedure to migrate a host – reducing a complex server migration from a multi-hour outage to (what appeared to the end-user) to be a trivial system reboot. We cloned the system a month in advance of the planned migration in order to test the hardware thoroughly prior to committing our decision. In the month leading up to the migration, we used the rsync script to hold the destination machine in sync with the source.

There is a cloud on the horizon for this type of operation though - traditionally, Solaris has been configured solely by a collection of flat files (compare with the "registry" style databases of Windows and AIX).

In Solaris 10, Sun has begun to introduce more complex configuration databases – the SMF has begun to emerge (I say "begun", as there still appears to be a transition underway of the old "rc" files). That's not to say it's not possible, just that we'll have to work a little harder to be that bit smarter in the future.

Mike Scott is the director of Hindsight IT Ltd, a small Solaris consultancy based in Central Scotland. He has been working in the North East and the central belt for longer than he cares to calculate, specialising in systems management with a keen interest in security and performance management. He can be contacted at sysadmin@hindsight.it