Adding ZFS to the FreeBSD dual-controller storage concept

Mikhail E. Zakharov, zmey20000@yahoo.com

Preface

Earlier we discussed the reliable FreeBSD based storage system model. Now we can see if it is possible to improve our concept with the famous ZFS technology.

But before everything else, we should name our small project. Something in honor of Beastie the daemon and the large BSD operating system family should be reasonable, so lets call our storage project the BeaST or even shorter, the BST.

The environment for our testing purposes will be similar to previous one. It is still my laptop which runs Oracle VM VirtualBox and a USB memory-stick to store shareable virtual drives and slow down their IO.

We will need to create thee virtual machines. One of them (clnt-1) will be the client for our storage system. We can easily take its configuration as-is from the previous test environment.

The last two machines (ctrl-a and ctrl-b) will serve as the storage controllers. These machines must be configured with at least 2048 MB of memory to run all our tests with ZFS without issues.

With the help of the VirtualBox Virtual Media Manager we should configure and attach to both storage controllers four **fixed-sized shareable drives** (d00, d01, d10, d11) for ZFS data volumes and four **fixed-sized shareable drives** (f00, f01, f10, f11) which we will use for ZFS cache.

We don't have any real hardware for our tests, so lets pretend that the data-drives are taken from the shared SATA shelf and imagine our cache-drives are fast Solid State Drives (SSDs) inserted into the SAS enclosure. Therefore don't forget to add appropriate controllers to the virtual-hosts configurations. Bear in mind, while the data-drives can be any size you prefer, the cache-drives must be at least 64 MB each.

Although it is not really mandatory for our purposes, you can even check the “Solid-state Drive” option for the cache-drives to assure yourself everything is done the best way possible.

See Figure 1 with the screenshot of Oracle VM VirtualBox Manager GUI showing the storage configuration section of the ctrl-1 machine.

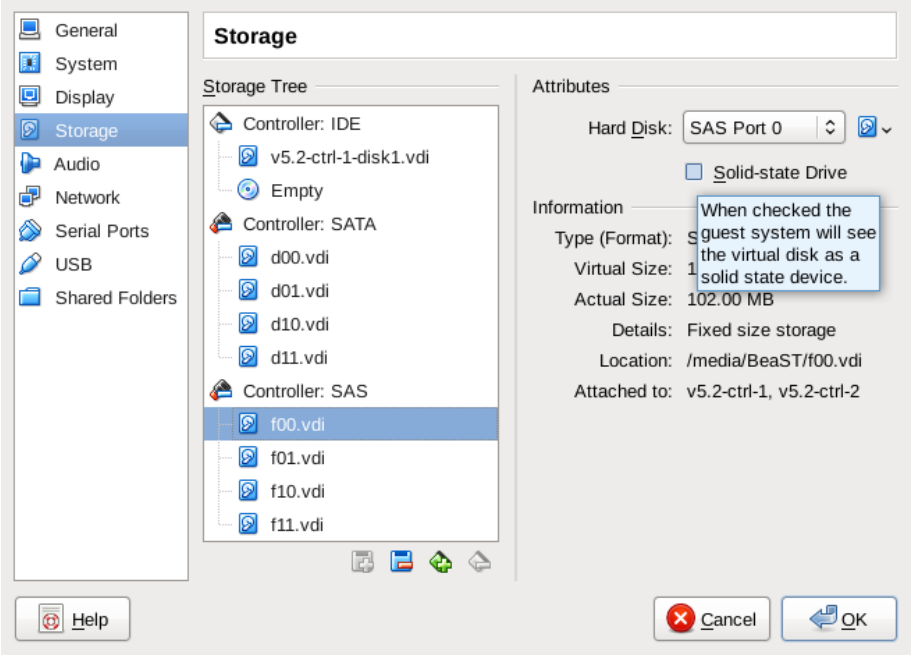

Figure 1. Controller storage layout example.

Network configuration is not changed. We will use two LAN connections: “private” for inter-controller and “public” for host-to-controllers communications.

Latest FreeBSD 10.3 Release can be installed on the dynamic-sized drives of all three virtual machines.

Configuration summary is shown in the table below:

Description	**ctrl-a**	**ctrl-b**	**clnt-1**
Inter-controller (private) network. Host-only adapter (vboxnet0)	IP: 192.168.56.10 Mask: 255.255.255.0	IP: 192.168.56.11 Mask: 255.255.255.0	–
Public network. Host-only adapter (vboxnet1)	IP: 192.168.55.10 Mask: 255.255.255.0	IP: 192.168.55.11 Mask: 255.255.255.0	IP: 192.168.55.20 Mask: 255.255.255.0
Base memory	2048 MB or more	2048 MB or more	Any appropriate value starting with 512 MB will do
Shareable, fixed-sized virtual drives for ZFS data volumes on the SATA controller.	d00, d01, d10, d11 – each drive is 100 MB size or more	d00, d01, d10, d11 – each drive is 100 MB or more	–
Shareable, fixed-sized virtual drives for ZFS cache on SAS controller	f00, f01, f10, f11 – at least 64 MB each	f00, f01, f10, f11 – at least 64 MB each	–
System virtual drives	At least 5 GB to store FreeBSD	At least 5 GB to store FreeBSD	At least 5 GB to store FreeBSD

(Dynamic-sized) on the IDE controller	10.3-Release default installation	10.3-Release default installation	10.3-Release default installation

Install FreeBSD on the ctrl-a and the ctrl-b virtual machines using default parameters and ada0 (dynamic-sized drive on the IDE controller) as the drive for the root file system. Then configure general parameters in /etc/rc.conf. This file for each controller can be easily taken from the previous environment:

ctrl-a	**ctrl-b**

hostname="ctrl-a"
 	
ifconfig_em0="inet 192.168.56.10 netmask 255.255.255.0" # Inter-controller LAN
ifconfig_em1="inet 192.168.55.10 netmask 255.255.255.0" # Public network
 	
sshd_enable="YES"
# Set dumpdev to "AUTO" to enable crash dumps, "NO" to disable 	
dumpdev="AUTO"
 	
# VirtualBox guest additions 	
vboxguest_enable="YES"
vboxservice_enable="YES"
 	
# iSCSI 	
ctld_enable="YES" # Targets
iscsid_enable="YES" # Initiators
	hostname="ctrl-b"
	
	ifconfig_em0="inet 192.168.56.11 netmask 255.255.255.0" # Inter-controller LAN
	ifconfig_em1="inet 192.168.55.11 netmask 255.255.255.0" # Public network
	
	sshd_enable="YES"
	# Set dumpdev to "AUTO" to enable crash dumps, "NO" to disable
	dumpdev="AUTO"
	
	# VirtualBox guest additions
	vboxguest_enable="YES
	vboxservice_enable="YES"
	
	# iSCSI
	ctld_enable="YES" # target
	iscsid_enable="YES" # initiator

Don't forget to set iSCSI “disconnection on fail” kernel variable in /etc/sysctl.conf on both systems to be able to failover in case of disaster to the alive controller:

kern.iscsi.fail_on_disconnection=1

After reboot check if everything runs well. Note that shared data-drives should be recognized by the kernel as ada1, ada2, ada3, ada4 and shared cache-drives would be da0, da1, da2, da3:

```
root@ctrl-a:/home/beast # dmesg | grep "da[0-9]:"
da0: <VBOX HARDDISK 1.0> Fixed Direct Access SPC-3 SCSI device
da0: 300.000MB/s transfers
da0: Command Queueing enabled
da0: 100MB (204800 512 byte sectors)
da1: <VBOX HARDDISK 1.0> Fixed Direct Access SPC-3 SCSI device
da1: 300.000MB/s transfers
da1: Command Queueing enabled
da1: 100MB (204800 512 byte sectors)
da2: <VBOX HARDDISK 1.0> Fixed Direct Access SPC-3 SCSI device
da2: 300.000MB/s transfers
da2: Command Queueing enabled
da2: 100MB (204800 512 byte sectors)
da3: <VBOX HARDDISK 1.0> Fixed Direct Access SPC-3 SCSI device
da3: 300.000MB/s transfers
da3: Command Queueing enabled
```

```
da3: 100MB (204800 512 byte sectors)
ada0: <VBOX HARDDISK 1.0> ATA-6 device
ada0: Serial Number VB9c2e46e9-3f3664e0
ada0: 33.300MB/s transfers (UDMA2, PIO 65536bytes)
ada0: 20480MB (41943040 512 byte sectors)
ada0: Previously was known as ad0
ada1: <VBOX HARDDISK 1.0> ATA-6 SATA 2.x device
ada1: Serial Number VB9508e565-d9dfd8c7
ada1: 300.000MB/s transfers (SATA 2.x, UDMA6, PIO 8192bytes)
ada1: Command Queueing enabled
ada1: 100MB (204800 512 byte sectors)
ada1: Previously was known as ad4
ada2: <VBOX HARDDISK 1.0> ATA-6 SATA 2.x device
ada2: Serial Number VB6a067a06-5a2a2e74
ada2: 300.000MB/s transfers (SATA 2.x, UDMA6, PIO 8192bytes)
ada2: Command Queueing enabled
ada2: 100MB (204800 512 byte sectors)
ada2: Previously was known as ad6
ada3: <VBOX HARDDISK 1.0> ATA-6 SATA 2.x device
ada3: Serial Number VB07f17028-9962c138
ada3: 300.000MB/s transfers (SATA 2.x, UDMA6, PIO 8192bytes)
ada3: Command Queueing enabled
ada3: 100MB (204800 512 byte sectors)
ada3: Previously was known as ad8
ada4: <VBOX HARDDISK 1.0> ATA-6 SATA 2.x device
ada4: Serial Number VBccf97b29-a1147aa3
ada4: 300.000MB/s transfers (SATA 2.x, UDMA6, PIO 8192bytes)
ada4: Command Queueing enabled
ada4: 100MB (204800 512 byte sectors)
ada4: Previously was known as ad10
```

ZFS pools basic configuration

If everything is done properly, we can configure ZFS pools on the controllers.

Lets use ada1/ada2 drives to create ctrl_a_m0 pool on the ctrl-a controller and ada3/ada4 drives to form ctrl_b_m0 pool on the ctrl-b controller. Then add volumes – v0 to each pool:

ctrl-a	**ctrl-b**
`zpool create -m none ctrl-a_m0 /dev/ada1 /dev/ada2` `zfs create -V 120M ctrl-a_m0/v0`	`zpool create -m none ctrl-b_m0 /dev/ada3 /dev/ada4` `zfs create -V 120M ctrl-b_m0/v0`

The only interesting options are:
-m none – prevents occasional pool mounting;
-V – desired volume size.

To check the result, run:

```
root@ctrl-a:/home/beast # zpool status
  pool: ctrl_a_m0
 state: ONLINE
  scan: none requested
config:
```

```
      NAME        STATE     READ WRITE CKSUM
      ctrl-a_m0   ONLINE       0     0     0
        ada1      ONLINE       0     0     0
        ada2      ONLINE       0     0     0

errors: No known data errors

root@ctrl-a:/home/beast # zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
ctrl-a_m0      124M  3.92M    19K  none
ctrl-a_m0/v0   124M   128M     8K  -
```

Read and write caches

Now lets try to move ZFS cache to the shared devices. In our case these will be da0 – da3 drives which we agree to consider the fast SSDs.

First, we will try to implement quazi-write cache with ZFS Intent Log (ZIL) forcing it to handle both types of synchronous and asynchronous transactions and write them to the shared drive da0 (the ctrl-a_m0 pool on the ctrl-a controller) and da2 (ctrl-b_m0 pool on on the ctrl-b):

ctrl-a	**ctrl-b**

`zpool add ctrl-a_m0 log /dev/da0`
 	
`# Always write and flush all file system transactions.`
`zfs set sync=always ctrl-a_m0`
	`zpool add ctrl-b_m0 log /dev/da2`
	
	`# Always write and flush all file system transactions.`
	`zfs set sync=always ctrl-b_m0`

Second, add the shared drives to the pools: da1 (on the ctrl-a) and da3 (on the ctrl-b) to use as the read-caches for both pools. Finally disable in-memory read-cache completely and move it to the shared devices.

ARC / L2ARC Configuration (read cache):

ctrl-a	**ctrl-b**

`zpool add ctrl-a_m0 cache /dev/da1`
 	
`# Turn-off ARC caching`
`zfs set primarycache=none ctrl-a_m0`
 	
`# Enable L2ARC caching to the shared device`
`zfs set secondarycache=all ctrl-a_m0`
	`zpool add ctrl-b_m0 cache /dev/da3`
	
	`# Turn-off ARC caching on ctrl-b`
	`zfs set primarycache=none ctrl-b_m0`
	
	`# Enable L2ARC caching to the shared device`
	`zfs set secondarycache=all ctrl-b_m0`

ZFS pools final configurations

On the last step of ZFS configuration we should import pools from opposite controllers. Then we have to set “failmode=continue” property.

This is quite dangerous because we just reject to stop working even if any errors are detected on the pools, but it's necessary to be able to failover to the next controller in case of disaster. So:

ctrl-a	**ctrl-b**

`zpool import -N ctrl-b_m0`
 	
`zpool set failmode=continue ctrl-a_m0`
`zpool set failmode=continue ctrl-b_m0`
	`zpool import -N ctrl-a_m0`
	
	`zpool set failmode=continue ctrl-a_m0`
	`zpool set failmode=continue ctrl-b_m0`

-N option of “zpool import” command above prevents the pool from being mounted.

Finally check zpool configuration. We should be able to see both pools on both storage controllers:

```
root@ctrl-a:/home/beast # zpool status
  pool: ctrl-a_m0
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-a_m0   ONLINE       0     0     0
        ada1      ONLINE       0     0     0
        ada2      ONLINE       0     0     0
      logs
        da0       ONLINE       0     0     0
      cache
        da1       ONLINE       0     0     0

errors: No known data errors

  pool: ctrl-b_m0
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-b_m0   ONLINE       0     0     0
        ada3      ONLINE       0     0     0
        ada4      ONLINE       0     0     0
      logs
        da2       ONLINE       0     0     0
      cache
        da3       ONLINE       0     0     0

errors: No known data errors
```

The Arbitrator

The arbitrator mechanism is similar to the one that was discussed in the previous paper. The only difference is that now we use data-volumes on ZFS pools. So just edit /etc/ctl.conf to add appropriate inter-controller communication configurations for the v0 volumes:

ctrl-a	**ctrl-b**

```
portal-group pg0 {
        discovery-auth-group
no-authentication
        listen 192.168.56.10
}

target
iqn.2016-01.local.sss.private:target0 {
        auth-group no-authentication
        portal-group pg0

        lun 0 {
                path
/dev/zvol/ctrl-a_m0/v0
        }
}
```

```
portal-group pg0 {
        discovery-auth-group
no-authentication
        listen 192.168.56.11
}

target
iqn.2016-01.local.sss.private:target0 {
        auth-group no-authentication
        portal-group pg0

        lun 0 {
                path
/dev/zvol/ctrl-b_m0/v0
        }
}
```

Then restart ctld daemon on both controllers to update iSCSI targets:

ctrl-a	**ctrl-b**
`service ctld restart`	`service ctld restart`

From both controllers connect with the LUNs on the opposite controllers:

ctrl-a	**ctrl-b**
`iscsictl -A -p 192.168.56.11 -t iqn.2016-01.local.sss.private:target0`	`iscsictl -A -p 192.168.56.10 -t iqn.2016-01.local.sss.private:target0`

And finally assembly the arbitrating construction by creating active-passive multipath devices, which have active paths pointing to the opposite controllers at their initial state:

ctrl-a	**ctrl-b**
`gmultipath create CTRL_B_BACK /dev/da4 /dev/zvol/ctrl-b_m0/v0`	`gmultipath create CTRL_A_BACK /dev/da4 /dev/zvol/ctrl-a_m0/v0`

Front-end configuration

The tandem of the arbitrator and the external shared cache forms a reliable structure resistant to a single controller failure. Therefore we can continue to configure host-connection configuration and update /etc/ctl.conf with the “public” sections to allow host access:

ctrl-a	**ctrl-b**
`portal-group pg0 { discovery-auth-group no-authentication listen 192.168.56.10 }`	`portal-group pg0 { discovery-auth-group no-authentication listen 192.168.56.11 }`

```
portal-group pg1 {
      discovery-auth-group
no-authentication
      listen 192.168.55.10
}

target
iqn.2016-01.local.sss.private:target0 {
      auth-group no-authentication
      portal-group pg0

      lun 0 {
            path /dev/zvol/ctrl-a_m0/v0
      }
}

target
iqn.2016-01.local.sss.public:target0 {
      auth-group no-authentication
      portal-group pg1

      lun 0 {
            path /dev/zvol/ctrl-a_m0/v0
      }

      lun 1 {
            path
/dev/multipath/CTRL_B_BACK
      }
}
```

```
portal-group pg1 {
        discovery-auth-group
no-authentication
        listen 192.168.55.11
}

target
iqn.2016-01.local.sss.private:target0 {
      auth-group no-authentication
      portal-group pg0

      lun 0 {
            path /dev/zvol/ctrl-b_m0/v0
      }
}

target
iqn.2016-01.local.sss.public:target0 {
        auth-group no-authentication
        portal-group pg1

      lun 0 {
            path /dev/zvol/ctrl-b_m0/v0
      }

        lun 1 {
              path
/dev/multipath/CTRL_A_BACK
        }
}
```

Then force ctld daemon to re-read and refresh iSCSI targets configuration on both controllers:

ctrl-a	**ctrl-b**
`killall -HUP ctld`	`killall -HUP ctld`

Check for essential messages in the dmesg output to see if everything goes smooth:

```
root@ctrl-a:/home/beast # dmesg | tail -10
```

```
da4 at iscsi1 bus 0 scbus8 target 0 lun 0
da4: <FREEBSD CTLDISK 0001> Fixed Direct Access SPC-4 SCSI device
da4: Serial Number MYSERIAL   0
da4: 150.000MB/s transfers
da4: Command Queueing enabled
da4: 120MB (245760 512 byte sectors)
GEOM_MULTIPATH: CTRL_B_BACK created
GEOM_MULTIPATH: da4 added to CTRL_B_BACK
GEOM_MULTIPATH: da4 is now active path in CTRL_B_BACK
GEOM_MULTIPATH: zvol/ctrl-b_m0/v0 added to CTRL_B_BACK
```

Figure 2 shows our storage system architecture layout we just created:

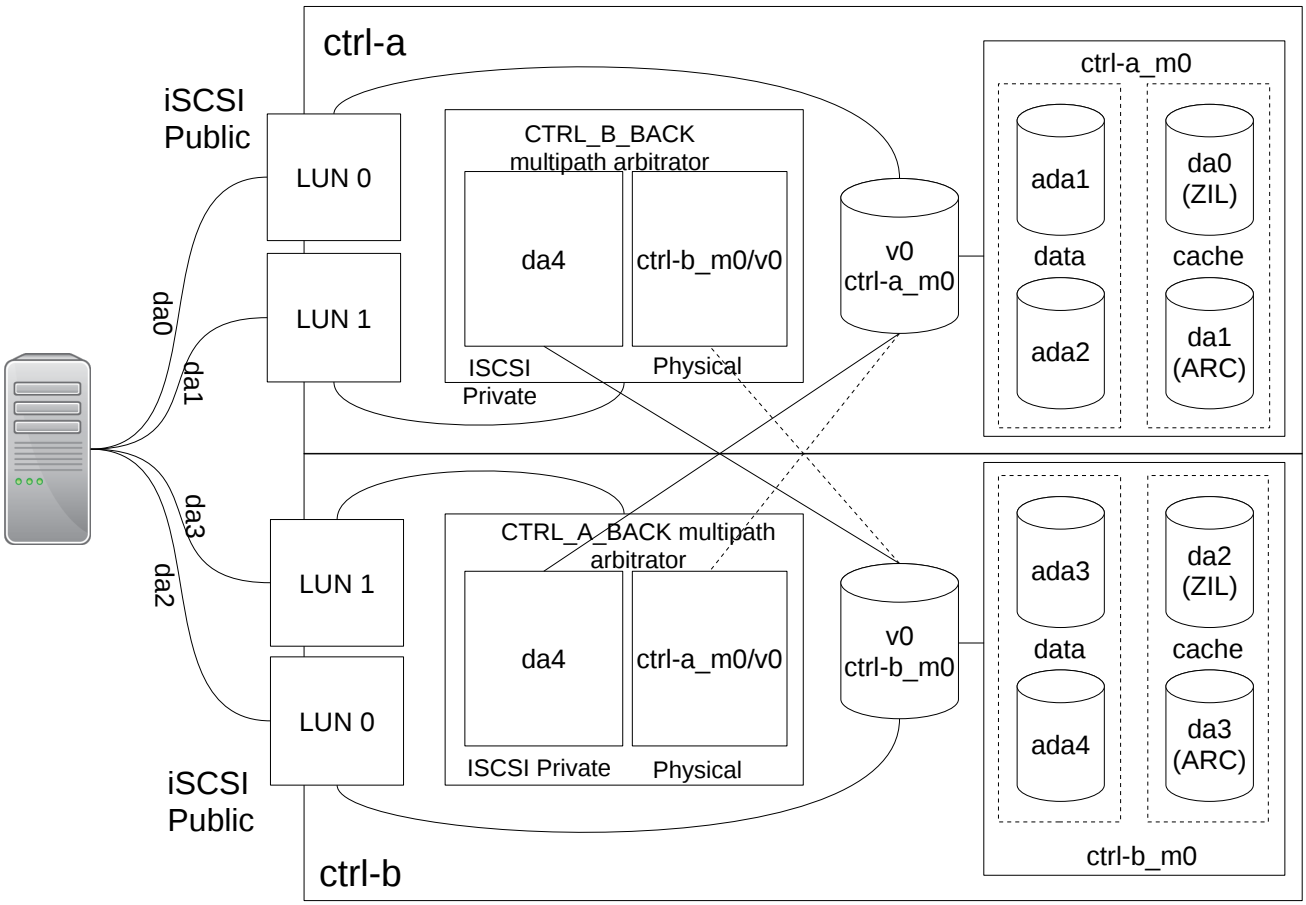

Figure 2. Dual-controller storage architecture with ZFS and external cache overview.

The client side

As the storage controllers are ready to serve requests, we can prepare the client. The clnt-1 initial configuration is identical to previous environment and can be used as-is. The essential lines of /etc/rc.conf are shown below:

```
hostname="clnt-1"
```

```
ifconfig_em0="inet 192.168.55.20 netmask 255.255.255.0"     # Public network

sshd_enable="YES"

# Set dumpdev to "AUTO" to enable crash dumps, "NO" to disable
dumpdev="AUTO"

# VirtualBox guest additions
vboxguest_enable="YES"
vboxservice_enable="YES"

# iSCSI
iscsid_enable="YES"     # Initiators
```

The /etc/sysctl.conf file is also the same:

```
kern.iscsi.fail_on_disconnection=1
```

After basic client preparation we can try to connect with public iSCSI targets of both storage controllers:

```
iscsictl -A -p 192.168.55.10 -t iqn.2016-01.local.sss.public:target0
iscsictl -A -p 192.168.55.11 -t iqn.2016-01.local.sss.public:target0
```

Check if dmesg output shows the appearance of the new da0, da1, da2, and da3 drives:

```
root@clnt-1:/home/beast # dmesg | grep "^da[0-9]:"
da0: <FREEBSD CTLDISK 0001> Fixed Direct Access SPC-4 SCSI device
da0: Serial Number MYSERIAL   1
da0: 150.000MB/s transfers
da0: Command Queueing enabled
da0: 120MB (245760 512 byte sectors: 64H 32S/T 120C)
da1: <FREEBSD CTLDISK 0001> Fixed Direct Access SPC-4 SCSI device
da1: Serial Number MYSERIAL   2
da1: 150.000MB/s transfers
da1: Command Queueing enabled
da1: 120MB (245760 512 byte sectors: 64H 32S/T 120C)
da2: <FREEBSD CTLDISK 0001> Fixed Direct Access SPC-4 SCSI device
da2: Serial Number MYSERIAL   1
da2: 150.000MB/s transfers
da2: Command Queueing enabled
da2: 120MB (245760 512 byte sectors: 64H 32S/T 120C)
da3: <FREEBSD CTLDISK 0001> Fixed Direct Access SPC-4 SCSI device
da3: Serial Number MYSERIAL   2
da3: 150.000MB/s transfers
da3: Command Queueing enabled
da3: 120MB (245760 512 byte sectors: 64H 32S/T 120C)
```

Then create appropriate multipathing devices to access both storage controllers:

```
gmultipath create CTRL_A /dev/da0 /dev/da3
gmultipath create CTRL_B /dev/da2 /dev/da1
```

Finally we can create a striped volume then add and mount a filesystem:

```
gstripe create ZDATA /dev/multipath/CTRL_A /dev/multipath/CTRL_B
newfs /dev/stripe/ZDATA
mount /dev/stripe/ZDATA /storage
```

If everything is well, we will see the appearance of the new mounted filesystem:

```
root@clnt-1:/home/beast # df -h | grep ZDATA
/dev/stripe/ZDATA    232M    8.0K    213M     0%    /storage
```

Then we can run all the tests we got familiar with in the previous version of the BeaST system. So lets start with a file copy operation:

```
root@clnt-1:/home/beast # cp ports.tgz /storage/
```

Figures 3, 4, and 5 show us the state of both controllers and the client:

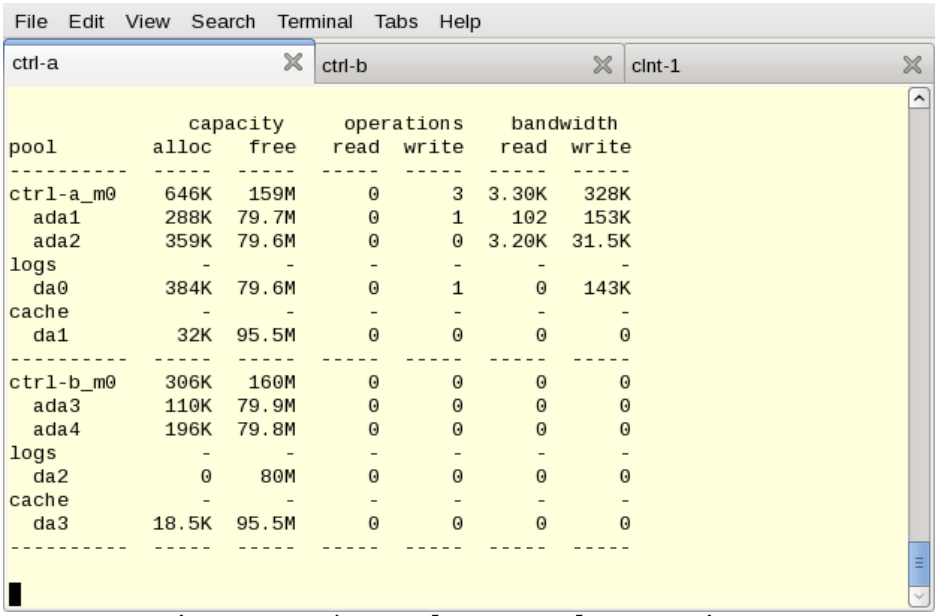

Figure 3. The ctrl-a normal operations.

You can see “zpool iostat -v 5” on Figure 3 which shows us the activity on ctrl-a_m0 pool detected by the ctrl-a. At the same time workload of the ctrl-b_m0 pool is hidden to the ctrl-a controller. Actually both pools are utilized but from different controllers.

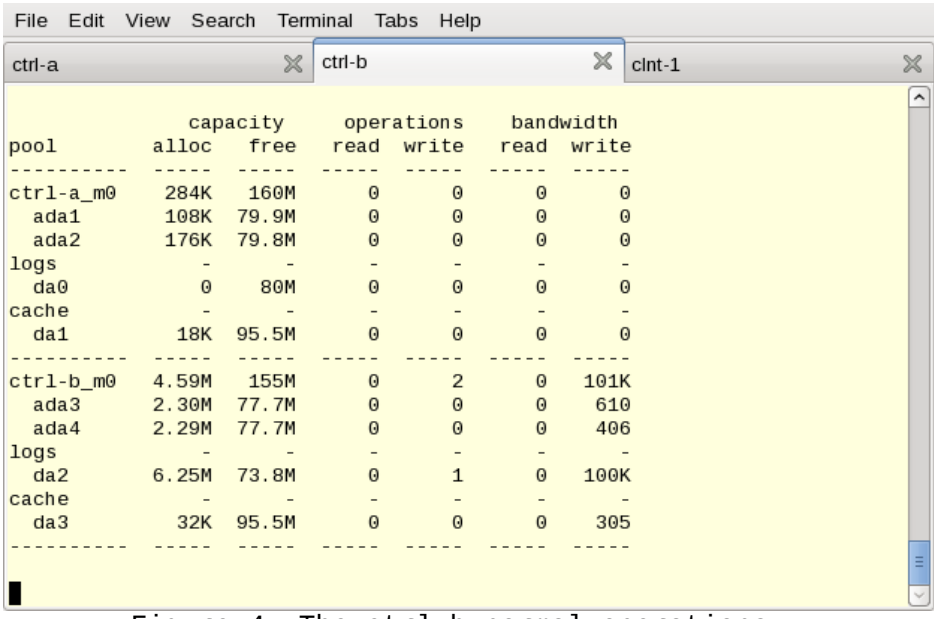

Figure 4. The ctrl-b normal operations.

The same situation on the ctrl_b controller (Figure 4): the ctrl-b_m0 pool activity is shown, but the ctrl-a_m0 workload is not detected.

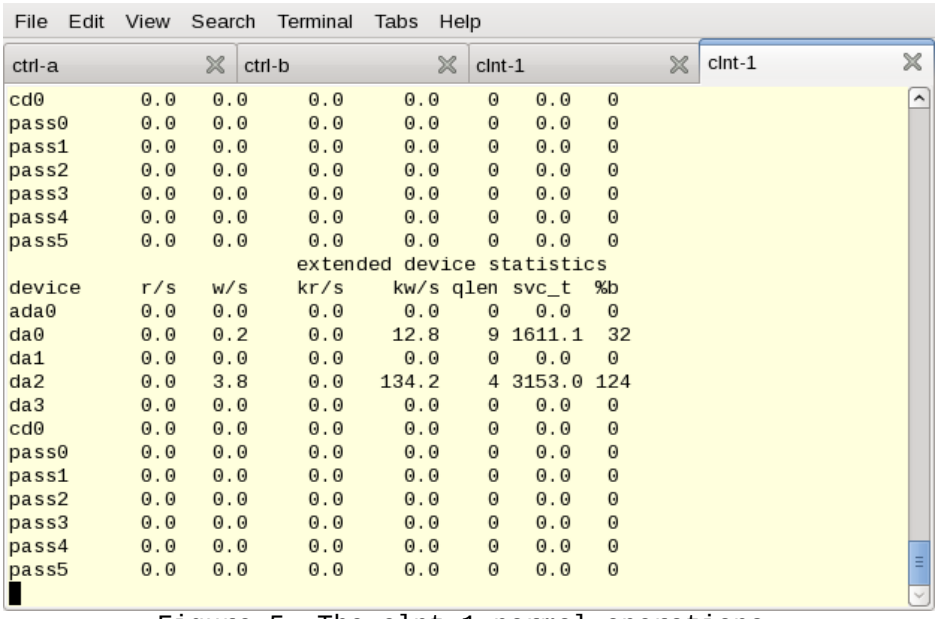

Figure 5. The clnt-1 normal operations.

From the client side (Figure 5) both primary paths are active and the data goes normally though the both owner controllers.

Now lets actually fail one of the controllers and see the result. Traditionally I will kick-off the ctrl-a.

Figure 6 now shows the survived controller:

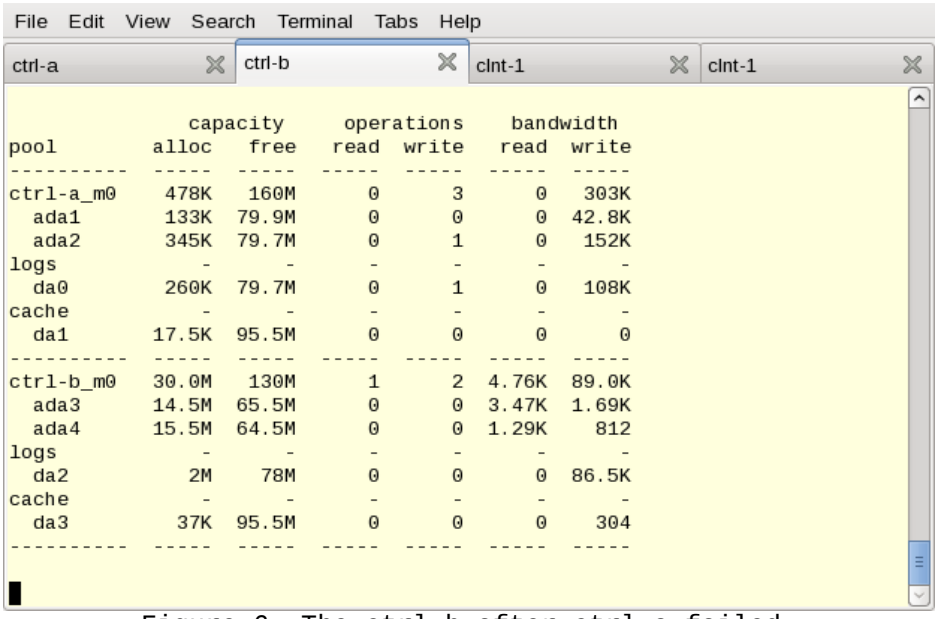

Figure 6. The ctrl-b after ctrl-a failed.

Now the ctrl-b takes all the clients workload: the arbitrator directed all data traffic through the survived controller.

And it is not really a surprise as the clnt-1 has lost all paths to the ctrl-a. On Figure 7 you can see that da0 and da1 have disappeared while da2/da3 are taking all the workload:

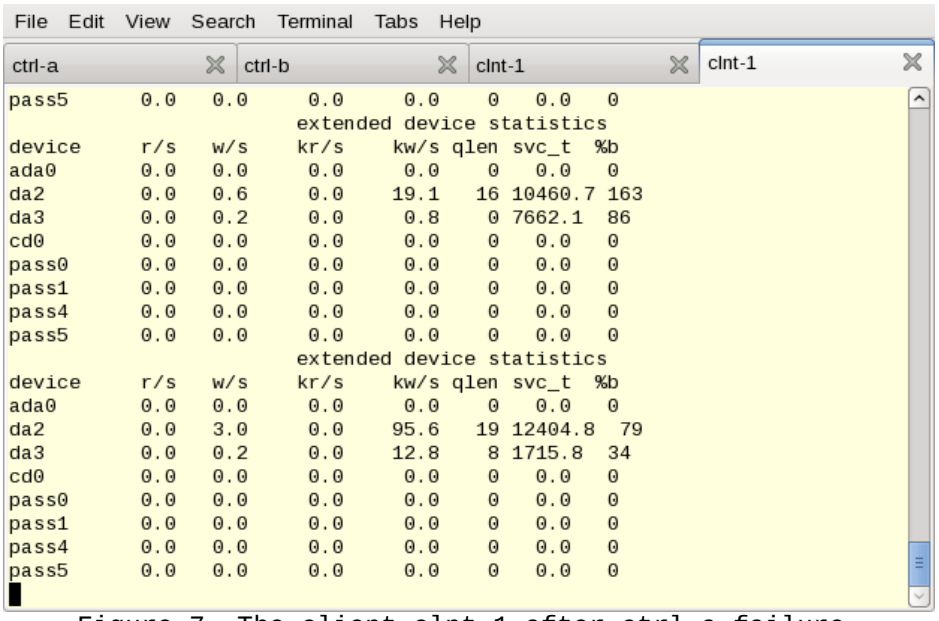

Figure 7. The client clnt-1 after ctrl-a failure.

After the file-copy operation has been finished, we can check if the data are written correctly:

```
root@clnt-1:/home/beast # md5 ports.tgz
MD5 (ports.tgz) = 82a5d6a7a3a89b7a5185a543fa6b3a56
root@clnt-1:/home/beast # md5 /storage/ports.tgz
MD5 (/storage/ports.tgz) = 82a5d6a7a3a89b7a5185a543fa6b3a56
```

We can state that the system works well in the laboratory. But now we are on a very slippery floor, as ZFS is not a cluster system and therefore is not designed to be run in the shared environments. Our storage solution is still for testing purposes only. So nobody can guarantee that everything will work in production. Beware of data loss!

The recovery procedure

From any point of view this is definitely not the brightest side of the life. Actually it is painful because the most important thing we should remember now, is that we will lose data if the last survived controller drops its access to the pools. In other words, the last controller **must** stay online until the recovery is finished: no shutdowns at all, even planned one!

Lets study the case. First of all, we must check the pool status:

```
root@ctrl-b:/home/beast # zpool status
  pool: ctrl-a_m0
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-a_m0   ONLINE       0     0     0
        ada1      ONLINE       0     0     0
        ada2      ONLINE       0     0     0
      logs
        da0       ONLINE       0     0     0
      cache
        da1       ONLINE       0     0     0

errors: No known data errors

  pool: ctrl-b_m0
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-b_m0   ONLINE       0     0     0
        ada3      ONLINE       0     0     0
        ada4      ONLINE       0     0     0
      logs
        da2       ONLINE       0     0     0
      cache
        da3       ONLINE       0     0     0

errors: No known data errors
```

But we must not trust it at all! As we remember, one controller can't see pool activities of the other controller. Additionally we disabled error detection on the pools by the “failmode=continue” option with our own hands.

So lets run the “scrub” procedure on the pool attached to the failed controller, then check the status to

see the real picture of disaster:

```
root@ctrl-b:/home/beast # zpool scrub ctrl-a_m0
root@ctrl-b:/home/beast # zpool status -v
  pool: ctrl-a_m0
 state: ONLINE
status: One or more devices has experienced an error resulting in data
      corruption.  Applications may be affected.
action: Restore the file in question if possible.  Otherwise restore the
      entire pool from backup.
   see: http://illumos.org/msg/ZFS-8000-8A
  scan: scrub repaired 1K in 0h0m with 2 errors on Tue Apr 19 16:07:39 2016
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-a_m0   ONLINE       0     0     2
        ada1      ONLINE       0     0     4
        ada2      ONLINE       0     0    10
      logs
        da0       ONLINE       0     0     0
      cache
        da1       ONLINE       0     0     0

errors: Permanent errors have been detected in the following files:

        <metadata>:<0x1b>
        <metadata>:<0x20>

  pool: ctrl-b_m0
 state: ONLINE
  scan: none requested
config:

      NAME        STATE     READ WRITE CKSUM
      ctrl-b_m0   ONLINE       0     0     0
        ada3      ONLINE       0     0     0
        ada4      ONLINE       0     0     0
      logs
        da2       ONLINE       0     0     0
      cache
        da3       ONLINE       0     0     0

errors: No known data errors
```

Finally run scrub on the ctrl-b_m0 pool and try to boot the ctrl-a controller.

Immediately after the ctrl-a is booted, login to its console and run “zpool export” to disconnect the controller from both of the pools:

```
root@ctrl-a:/home/beast # zpool export ctrl-a_m0
root@ctrl-a:/home/beast # zpool export ctrl-b_m0
```

Now we can breath normally. Calmly reboot crtl-a once more, then import pools back again:

```
root@ctrl-a:/home/beast # zpool import -N ctrl-a_m0
root@ctrl-a:/home/beast # zpool import -N ctrl-b_m0
```

Then run scrub on both controllers:

```
root@ctrl-a:/home/beast # zpool scrub ctrl-a_m0
root@ctrl-a:/home/beast # zpool scrub ctrl-b_m0
root@ctrl-b:/home/beast # zpool scrub ctrl-a_m0
root@ctrl-b:/home/beast # zpool scrub ctrl-b_m0
```

Also you may need to clear errors on the pools:

```
zpool clear ctrl-a_m0
zpool clear ctrl-b_m0
```

If we succeed with the pools recovery procedure we should carefully restore the arbitrator, all broken iSCSI connections and multipathing primary paths. In other words we have to repeat most of the steps specified in the article once more. It is a very tedious process but it should be done carefully to finish the storage system restoration.

What is next

We have added ZFS to the BeaST project and that is very good. But bearing in mind the real life and performance demands of the different workloads, it is probably not the best idea to put the main system cache to the external solid state drives. Better if we are able to implement in-memory cache and make it somehow to be mirrored between both controllers. This would be the first task for the future development.

Second, we still must test the BeaST project under the pressure of high workloads on the real hardware and test not only the performance but the stability of our storage concept.

Third, FreeBSD 10.3 Release brought to the table the high-availability options for the CAM Target Layer and we should check what the BeaST can benefit from the new features.