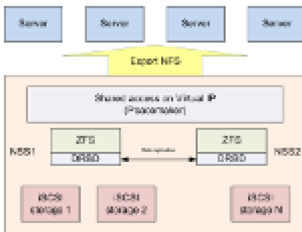
Networking

Setup of a redundant network storage system. A legacy approach.[†]

Andrea Lora,[a] Giuseppe Nantista,[a] and Augusto Pifferi.[a]



In order to provide an affordable network storage for general purpose servers, we setup a system that is able to share NFS resources avoiding single point of failure. This architecture is now used as a storage for our email server. The goal is achieved using opensource software and different vendor storages.

Keywords: HA storage, DRBD, NFS, ZFS, iSCSI.

1 INTRODUCTION

When providing a general purpose server the first goal to achieve is to avoid that a single fault on a component of the system can cause loss of data or, at least, long time of unavailability of the system. Server that offer those services must be designed to work on more physical machines, located in different IDCs, running different instances of the service, but offering same data to clients. This can be obtained by replicating every single server and assuring that in every moment data contained in the first server are the same of those contained in other.

However this approach can limit the overall scalability of the system, so the only way to get both replicated service and mirrored storage is to separate the application and the storage layer. The main advantages of this structure are two: high scalability, in fact it is possible to add other storage in order to create a clustered pool of resources; single maintenance is needed, no matter how many services use the infrastructure; servers are easier to manage, because data are kept outside.

1.1 Overview

This paper will propose a possible implementation of this scenario, using Linux as the operating system for the network storage servers, iSCSI[1] as the network storage protocol used to access different SANs from storage servers, DRBD as the replication engine, Corosync and Pacemaker as the clustering and resource management daemons used by network storage system servers, ZFS as the local file system and NFS[2–7] as the shared access protocol. This kind of cluster is often referred as *Active Passive - Shared Nothing* architecture.

The described infrastructure is shown in Fig. 1

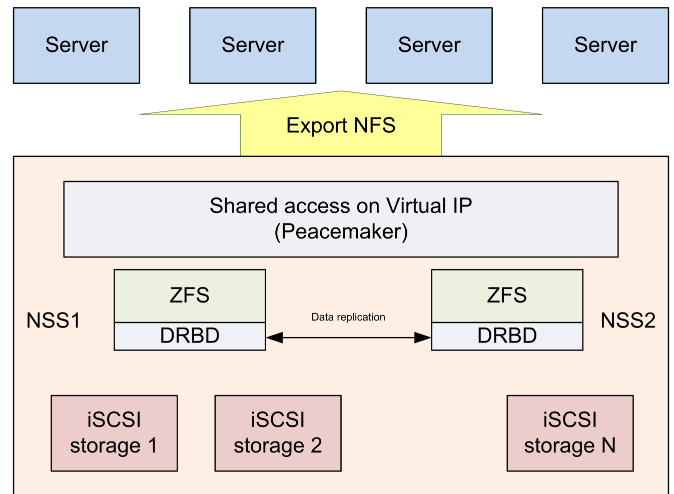


Fig. 1 Overview of the architecture

1.2 Hardware

Implementing an active/passive cluster means we have two (or more) machines configured to do the same job. Only one of the machines is actually serving the resources, while the others are put in stand-by, ready to take over the service in case the primary get some kind of failures. We'll cover details of our configuration in section 2.

1.3 Network Layer

Nodes need to talk each other to keep them synched and serving data to clients. We also need a floating IP which will be assigned to the active server. The configuration of the virtual IP is explained in the section 7.

1.4 Block storage layer

There are three main storage networking standard for linking data storage facilities, those are iSCSI, Fibre Channel and Infiniband. Both Fibre Channel and Infiniband require dedicated

[a] *Istituto di Cristallografia, C.N.R., via Salaria km 29,300 - Monterotondo, Italia*
Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 4.0 Internazionale
† Rapporto tecnico IC-RM 2016/02 protocollato in data 14/04/2016 n. 658

network hardware. The only protocol that allows the use of generic hardware such as network switches and cards is iSCSI. It also permits opensource implementation of the protocol stack.

We used three storage systems, a Syneto Unified Storage, a self assembled storage based on Illumos operating system, running Nexentastor and a HP Lefthand P4000. Then we configured pairs on LUN of the same dimension on two storages, and accessed those LUN from the network storage servers.

1.5 Replication system layer

To avoid SPOFs we need to be sure that all the data is stored in (at least) two location. This will avoid data loss in case of failure of the block storage. We chose DRBD as replication engine. We'll cover details about it in section 4.

1.6 File system layer

Since DRBD is seen by the OS as a block device we needed to choose a file system suitable for hosting data. ZFS was our choice, due to its excellent records in data storage and powerful capabilities. Implementation of ZFS on Linux requires some efforts in the configuration due the fact it's not tightly integrated in the OS. We'll cover ZFS details in section 5.

1.7 Shared access layer

After creating the file system we need some means to access it from remote machines. Industry standard to allow access of remote file system is NFS. There was no reason to go against it, in order to follow a legacy approach. We'll cover NFS details in section 5.

1.8 Orchestration

Due to the clustered nature of this stack cluster/orchestration tools were needed. Pacemaker and Corosync were used as cluster suite. Although it's possible to command corosync/-pacemaker through the *crm* command line utility, we used the Linux Cluster Management Console (LCMC) to do most of the work. The presence of a GUI allow to get at a glance overview of cluster status. Additional customization was made to activate STONITH mechanisms. We'll cover cluster details in section 7.

1.9 Monitoring

The fault tolerance capabilities of this kind of stack don't mean that we simply forget it because it works. We implemented some custom script to take analytics of the performances of cluster nodes and setup some alerts accordingly. We'll cover the details is section 8.

1.10 Backup

Cluster nodes are always seen a single entity. The fact that data is redundant doesn't mean that it's guaranteed to be safe. A simple *rm –rf* can (and will) destroy data on the active and passive node at the same time. The passive is not the backup of the active, therefore backup strategies need to be implemented to avoid data loss. We'll cover details about backup in section 8.

2 Hardware

The cluster consist in 3 Virtual Machines. Two are the active/passive nodes of the NFS server, one is a very light instance acting as a quorum node. The fat nodes are equipped with 4 virtual CPU, 8GB of RAM, a bunch of disk space (16GB). The thin node is just 1 CPU, 512GB of RAM and 4GB of disk space.

The virtual machines are spread between two ESX systems. One is a fully fledged VMware Cluster hosted on an HP Blade System, the other is an HP G5 Server. The HP Blade System has attached a HP Lefthand P4000 as storage system, while the G5 has a Syneto Unified Storage. Both the storages offer a 1TB raw LUN accessible via ISCSI from the OS of the virtual machines, and are also in charge of hosting the data of the disks of the Virtual Machines we used.

Network side we configured 3 different networks and VLANs accordingly, one for NFS share, one for Replication/Heartbeat, one for ISCSI. At present the VMware Cluster hosts one fat node and the thin (quorum) node, while the ESX standalone G5 hosts the second fat note.

Please note that this configuration isn't truly SPOF free, because in case of complete shutdown of the VMware cluster, the last node will not be able to take over the job due to the lost quorum. Simply move the quorum machine elsewhere and you'll reach full SPOF free configuration.

After the configuration of the virtual machines we installed Ubuntu 12.04 Server on each server. The Ubuntu choice was due the fact that PPA repositories were available for the zfsonlinux project, an essential part for the system, and the LTS support from Canonical.

3 Block Storage Layer

After installing the OS on the fat nodes we enabled access to the ISCSI resources installing the iscsi-initiator-utils via apt-get, then we proceed with the discover of target:

```
# iscsiadm -m discovery -t st -p ip_of_portal
```

After that we should log in to the portal

```
# iscsiadm -m node - target_name -p ip_of_portal -l
```

And configure automatic login at boot

```
# iscsiadm -m node -T target_name -p ip_of_portal --op update -n node.startup -v automatic
```

Through the use of dmesg or with fdisk –l we should now see the presence of a new block device into the OS. In our case our LUNs were 1TB large. It's important that the LUNs size is the same for both the fat nodes. Failure to do so may cause problems with the replication.

4 Replication System Layer

Due to our design requirements we needed that the LUNs were perfectly synchronized, every time, any time. The software that allow to reach this goal was the Distributed Replicated Block Device DRBD. We can understand it as network based RAID-1 (mirror). DRBD offers asynchronous (A mode), semi-synchronous (B mode) and synchronous (C mode) replication. A DRBD device is seen from the OS as a standard block device, which can be manipulated like any other: from the OS point of view a DRBD device is just another hard disk. DRBD consists in two components: one runs as a daemon, it uses a couple of tcp port to ensure bidirectional communication, the other is a kernel loadable module. DRBD can be used without orchestration tools, but that won't be useful to achieve HA because you need to manually control primary/secondary promotion.

In order to install DRBD we first install the package trough apt-get

```
# apt-get install drbd8-utils
```

After the installation of DRBD we need to create a resource for it. A resource is the block device DRBD will offer to the OS. In the resource definition there will be present the physical block device where DRBD will write. Take into account that DNS names and IP must be configured accordingly with your network topology. We used ip in the configuration, but we also edited our /etc/hosts to provide coherent information. Rely on DNS to resolve the hostname is not advisable due the possible failure of the service.

```
resource r0 {
      protocol        B;

      startup {
              degr-wfc-timeout        0;
      }

      net {
              max-epoch-size  8000;
              max-buffers     8000;
              unplug-watermark        8000;
              cram-hmac-alg   sha1;
              shared-secret   my_shared_secret;
      }

      disk {
              on-io-error     detach;
              no-disk-barrier;
              no-disk-flushes;
      }

      syncer {
              rate    25M;
               al-extents      3389;
               csums-alg       md5;
               verify-alg      md5;
               c-max-rate      100M;
```

```
                c-min-rate      10M;
        }

        on nfs-1 {
                device          /dev/drbd0;
                disk            /dev/sdb;
                flexible-meta-disk      internal;
                address         10.10.73.26:7788;
        }
        on nfs-2 {
                device          /dev/drbd0;
                disk            /dev/sdb;
                flexible-meta-disk      internal;
                address         10.10.73.27:7788;
        }
```

In this example we define a resource called r0 synched semi-synchronously (B type). It consist in two hard disk being synched on the machines (nfs-1 and nfs-2). On both nodes the disk used is /dev/sdb and the block device exposed by DRBD is /dev/drbd0. We define an internal flexible meta-disk, storing metadata of DRBD inside the replication block device. We also tuned some parameters about the syncer, in particular we set the maximum transfer rate in 25 MB/s, and we choose md5 as the checksum and verify algorithm. Others parameters are in the net section and those needs to be tuned to your particular workload.

We then initialize the meta-disk area. We need to do this only on one node.

```
[root@nfs-01 etc]# drbdadm create-md repdata
About to create a new drbd meta data block on /dev/sdb.
 . ==> This might destroy existing data! <==
Do you want to proceed? [need to type 'yes' to confirm] yes
Creating meta data... initialising activity log NOT initialized bitmap (256 KB) New drbd meta
data block sucessfully created.
```

After that we can start DRBD on both nodes. Just type

```
service drbd start
```

If we check DRBD status (*drbd-overview* command is a beautiful wrapper that show us info) we'll see that both nodes are secondary, and not synched.

```
[root@nfs-01 /root]# drbd-overview
0:r0  Connected Secondary/Secondary Inconsistent/Inconsistent A r-----
```

We shall promote one of the nodes as primary (we will use nfs-01)

```
[root@nfs-01 /root]# drbdadm -- --overwrite-data-of-peer primary r0
0:r0  Synctarget Primary/Secondary Inconsistent/Inconsistent A r-----
```

After a while a drbd-overview will show

```
[root@nfs-01 /root]# drbd-overview
0:r0  Connected Primary/Secondary UpToDate/UpToDate A r-----
```

This will informs us that the LUNs are synched. Take into account that we didn't specify a runlevel for autostart of DRBD, and that's normal. In fact the DRBD resource will be managed by the cluster suite. More informations on clustering tools is available on section VII.

5 File System Layer

We now reached our first goal: we have two different LUNs synched. Every write operation we do on nfs-01 will be replicated to the nfs-02 node. But that's just a block device, we need a file system over that to be actually useful. Our file system of choice was ZFS. We wrote a small white paper about why ZFS is the best filesystem to be used for data storage at this time, so we won't speak about its features, but we'll focus about how to integrate it in our system. Although it is possible to use ZFS in user space trough FUSE there are serious drawbacks in performances so the suggested operating method utilizes kernel loadable modules. Installation is simplified due the presence of ppa repository for Ubuntu. Installation steps are as follow:

```
[root@nfs-01 /root]# apt-get install python-software-properties
[root@nfs-01 /root]# add-apt-repository ppa:zfs-native/stable
[root@nfs-01 /root]# apt-get update
[root@nfs-01 /root]# apt-get install ubuntu-zfs
```

After executing this commands we are ready to use ZFS on Linux. We can load the module through modprobe, but if the OS detects a ZFS volume on the disks will try to autoload the module automatically. There's a small caveat on this: since DRBD will replicate the volume in Active/Passive mode the secondary machine will actually see a ZFS disk pool and will try to load the module. This will fail due the read only DRBD property on the secondary and will cause some problems in the stack. We must exclude ZFS module from the autoload list. The scripts that manage the cluster resources will be on charge of loading/unloading it. To avoid the auto load of the module we simply add the following lines on the /etc/modprobe.d/blacklist.conf

```
blacklist zfs
install zfs /bin/false
```

The first inhibits the autoload of the module, the second specify that ZFS will need to be loaded only through an *insmod* command and not through *modprobe*.

Usual rules for ZFS administration apply. Since the block device underlying ZFS is replicated, a ZFS command will propagate to the second node, so this actions must only be done on the active ZFS node.

We start creating a ZFS pool and referencing it with the correct block device.

```
zpool create tank /dev/drbd0
```

Since /dev/drbd0 is our replicated block device it is the correct one to be passed to zpool as an argument. *Tank* is a common name for a zpool, but it's your choice. Creating a pool automatically creates a ZFS filesystem with the same name, but in order to ease administration it's good practice to create nested ZFS filesystem inside the pool.

```
zfs create tank/share1
```

Here we command ZFS to create a nested filesystem inside the tank pool, and we call it share1. That's particularly useful because you can change some ZFS attributes (compression for example) on the single filesystem, rather than the pool. It also helps you managing different snapshots policy pool based.

One of the most important tune to make to ZFS is to disable the atime option of the filesystem. With atime enabled every time you access a file the filesystem updated the metadata regarding its last access timestamp. This lead to a write operation for every file read, quickly degrading the performances of the system. You can completely disable the atime property with zfs set atime=off or use a lighter timestamp called relatime. With *zfs set atime=relatime* you are telling the filesystem that you want an update of the last access timestamp, but just one every 24 hours.

Please note that we avoid to use ZFS capabilities of auto-exporting filesystems via NFS. We decided to manually export them via the usual linux tools. That's because we need more fine grained control on the events: the clustering tools are in charge of orchestrating everything. Since we blacklist the ZFS module in the modprobe the clustering tools must me take charge of loading it, we'll cover this kind of details in section 7.4.

Take into account that ZFS is not a clustered filesystem: this mean you can't have ZFS mounted on both the server at the same time serving the same data.

6 Share Access Layer

In order to access the replicated resource we just created from remote machines we choose to use the nfs server. NFS was one of the most used protocols to share files between machines in unix environment: it requires few dependencies and the protocol is old but reliable. The NFS server can be installed through apt-get install nfs-kernel-server. If you are using NFSv3 configuration is limited to the /etc/exports file, where you can add the directory you want to export and the ACL and options for them. NFSv4 need an additional component, named *idmap*, which is a daemon that map remote uid to local ones. We don't need to configure the /etc/exports at this time nor we need to enable nfs server at startup. The orchestration tools are in charge to dynamically load the kernel modules and update the exports list.

The system is designed to offer high availability and transparent migration from one node to the other. NFS clients whose operations were on the wire will have trouble reconnecting timely. We need to adjust two parameters to lower the grace time of NFS and smooth the failover. There parameters are nfsv4leasetime and nfsv4gracetime, both tunable in /proc/fs/nfsd

7 Orchestration

All the orchestration is done by corosync/pacemaker. Corosync in the software that manages cluster membership of the nodes, determining quorum and promoting/demoting nodes. Pacemaker is the resource manager that starts and stops the various services and provide fencing agents for unresponsive hosts. Corosync and pacemaker are often called together "cluster tools". Configuring them was usually done through config files, but a brilliant project called "Linux cluster manager console" allows to use a GUI to install and configure the cluster tools. You'll find the relevant configuration files in Appendix. The use of LCMC is pretty straightforward. Pacemaker actually performs the operations through agents. These agents are often bash scripts with certain exit codes. Two main kind of agents exists: LSB and OCF. LSB are considered legacy modules provided to maintain compatibility, OCF is the new format and is becoming pretty popular and more manageable.

7.1 Corosync / Pacemaker

Corosync configuration in its simplest form is limited to declaration of IP address, network card to be used to cluster communications and multicast address.

Pacemaker allow us to configure certain kind of relationship between resources such as "start A before B". This is a core feature to load the various components: we need a bottom up approach when starting resources, and vice-versa when stopping them. LCMC shows a nice view with the relationship of the resources, and allows to configure the resource agents we need for getting the job done.

Caveat: a cluster should be configured to start resources only when the appropriate fencing agents are started to avoid the risk of split brain situations which can be very difficult to recover without data loss.

7.2 Fencing agent

Fencing agents are in charge of killing a node in case it doesn't answer to keepalives: usually they use STONITH approach via IPMI or remote power switches. Since we are in a virtual environment STONITH is done via VMware api calls. In case of VMware VCenter there is a well know agent: *stonith_external/vcenter*. If you are using an unlicensed version of ESXi you can rely on ssh to poweroff the guests. You can find an example of ESXi fencing agent in the Appendix. If fencing agents fail to start for any reason, your cluster won't be able to start other resources. For debugging or devel purpose you can override through the parameter *stonith_enabled*. Please note that operating a cluster without stonith agents can lead to split brain situations and consequential data loss.

7.3 Block device agent

The block device agent in this configuration is *ocf_drbd*. It's a master/slave agent that ensures no two nodes are running the same drbd resource as primary. The only parameter this agent takes is the name of drbd resource. In our case it is r0. All the following agents run on primary node.

7.4 Filesystem agent

After the block device appears in the operating systems we need to mount the filesystem that it contains: in our case ZFS. We haven't found an ocf zfs agent, we ended writing our own. We need to implement the action start / stop / check. For start action what we need to do is manually load the ZFS module that we blacklisted before and running a zpool import. For stop action we need to zpool export the pool and remove the module from ram. For check action we can rely on zpool list. At this stage the agent takes two parameters: device to be mounted and mount point. The zpool name is hardcoded as tank.

7.5 NFS Server Agent

Nothing much to say here. A predefined ocf agent named *nfsserver* manage the task of running and stopping the NFS kernel server. As parameter it takes the script to start the nfs daemon.

7.6 Exports agent

After the NFS server is started we can describe various exports directory. An OCF agent named *exportfs* is available for such task. It takes as parameters the client ACLs, local share name, a unique *fsid* within the cluster for that share and the export options, if any.

7.7 Virtual IP agent

Clients don't connect to the real IP of the servers, because they have no clue which NFS server is primary. We define a last resource, a Virtual IP assigned to the primary server. The OCF agent is called *IPAddr2*, and the only mandatory parameter is the IP address you want to use. You can also specify a netmask if needed. This IP is what the client will use to access the nfs share.

7.8 First Start

At first start corosync form the cluster, assign one of the nodes as DC (Designated Controller) and check for pacemaker configuration. Then fencing agents are started and checked for status. If they are ok the next agent kicks in, starting drbd on both nodes, promoting one to Primary and the other as secondary. Sync starts here if needed. Only on the primary node as soon block device is confirmed to be available, filesystem is mounted, nfs kernel started, directory exported and finally IP address assigned to the primary node. If any of the steps fails to complete pacemaker perform a rollback taking down the resources on the primary node until block device layer. Then it perform a role switch on the nodes: the primary is demoted to secondary and the secondary promoted to primary. After it tries to perform all start actions on the new primary.

7.9 Failover

If failover is requested pacemaker first tries to take down all the services on the primary node. If stopping the services is successful it performs role switch on the nodes and try to start all the resources to the other node, which is primary at this point. If during the stop of the services on the old primary a certain timeout is met, or if them fails with error the cluster tools proceed to fence the unresponsive server. Often this mean a complete poweroff/poweron cycle of the server.

8 Monitoring

Even if cluster can perform automatic failover is good practice to monitor the server to get instant insight of what is happening in the system. The items we want to monitor are the system resources of the servers, the status of pacemaker resources and some statistics about performance of nfs. We installed the zabbix agent throught apt-get to get the basic stats of the server and we also added several custom checks to enable the monitoring of the pacemaker resources. We wrote a simple bash script to check the various services and added the zabbix_agent.conf the relevant keys. You can find both the script and the zabbix keys in the Appendix.

9 Backup

While operating a cluster the administrator must take into account that the secondary server is the replacement, not the backup. Secondary server are there to achieve HA, but a number of things can go wrong. Due the semi-synchronous nature of this setup both block devices are actually mirroring each other, like a RAID-1. So we need other strategies to achieve solid backup.

9.1 Snapshots

When we faced the choice of the filesystem we agreed on one thing: filesystem snapshot is just too useful to not have it. And this influenced heavily our choice in favor of ZFS. Its copy-on-write mechanism allows to take snapshots in nearly no time. More important snapshots aren't really occupying space on disks: they are just thin copies. And they are easily accessible through the hidden .zfs directory where the admin can just inspect or copy the files. What we needed was a system that automates the snapshots of the filesystem and rotates the backup in a way that respect our policy. We found in zfs-auto-backup the solution of this problem. This handful scrIPt (even available through apt-get) allows to take snapshot while defining a retention policy. We keep 4 quarter snapshot, 24 hourly snapshot, 30 daily, 4 weekly, 12 monthly and 5 yearly. If the system becomes bloated by the yearly snapshots we can easily offload them as explained in the next section.

9.2 Filesystem send

The *zfs send* command allow us to stream the whole filesystem to stdout. The *zfs recv* command takes a filesystem stream from stdin and store it in the designed pool. The capability of sending a filesystem, or just the incremental snapshots of it, is another powerful feature of ZFS and it comes extremely handy for backup strategies. We can offload old snapshots just sending them over the network to another machine that is ZFS capable. Due the fact that zfs send/recv use standard descriptors we can just pipe through an ssh connection. An example of backup is

```
zfs send tank@snap1
       | ssh host2 zfs recv tank/backup
```

ZFS properties can be different on destination. For example we activate compression on the destination filesystem. Sending data out of the cluster solves the problem of backup. We use the local snapshot for common activities, such restoring deleted files or accessing old versions of them. If the cluster has problem, or if we need do some heavy I/O load like a search in the whole snapshots for a certain string we can offload the work on the backup server which normally doesn't require to be responsive. A forked version of zfs-auto-backup feature auto send of snapshot to a remote host.

10 Conclusion

The work described in this technical report was made in order to obtain a storage system for general purpose in high availability environment, assuming multiple and contemporary access to it. The result obtained required a big effort in deploying it, resulting in very high probability of human error when managing it, furthermore we accepted some trade-off that limit the overall system performance, resulting in slow file access. However, the conceptual effort and the experience have highlighted some of the strengths and weaknesses of this system. Scalability is delegated to the iSCSI layer, growing the LUN results in the capability to immediately grow the file system, and allows, thanks to the adoption of ZFS, automatic and cost-free snapshot mechanisms and consequently enables granular recovery of individual files starting from the selected snapshot. Future works aimed at exploiting the benefits of distributed and scalable storage systems, such as CEPH, continuing to enjoy benefits of a file system like ZFS in the backup system.

Glossary

ISCSI:	Internet Small Computer Systems Interface
DRBD:	Distributed Replicated Block Device
NFS:	Network File System
SPOF:	Single Point of Failure
ZFS:	Zettabyte File System
LUN:	Logical Unit Number (SCSI devices)
LCMC:	Linux Cluster Manager Console
STONITH:	Shoot the Other Node in the Head (computer clustering)
COW:	Copy-on-write
IPMI:	Intelligent Platform Management Interface

Appendix

All the configuration files needed in this project can be found at http://code.sra.mlib.cnr.it/andlor/nfs

References

1 J. Sartran, K. Meth, C. Sapuntzakis, C. M., Z. E., Internet Small Computer Systems Interface (iSCSI), Request for Comments: 3720.

2 S. M. Inc., Network. File System Protocol Specification (NFS), Request for Comments: 1094. March 1989.

3 B. Callaghan, B. Pawlowski, P. Staubach, NFS Version 3 Protocol Specification, Request for Comments: 1813. June 1995.

4 S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, D. Noveck, NFS version 4 Protocol, Request for Comments: 3530. April 2003.

5 A. Galloway, Things about zfs that nobody told you. jul 2011.

6 A. Traeger, E. Zadok, N. Joukov, C. P. Wright, A nine year study of file system and storage benchmarking, Trans. Storage 4 (2) (2008) 5:1–5:56. doi:10.1145/1367829.1367831.

7 Solaris Internals wiki, ZFS Best Practice Guide.