Introduction to the ZFS Filesystem

Benedict Reuschling

November 2, 2017

Introduction to the ZFS Filesystem

These slides describe the ZFS filesystem. Besides the basic design ideas we will have a closer look at the features of ZFS that are interesting for users. We will explain how to create a storage pool and administer it, as well as configuring ZFS parameters. Compression, serializing pool data, self-healing in the case of errors, and deduplication will be covered.
The examples are as independent of any specific distribution as possible. When device names are used, we will use those used in FreeBSD. They can be replaced easily to fit device names used in other distributions without changing the ZFS commands too much. Special cases will be mentioned. Since ZFS has more features than this slide deck can cover in a reasonable amount of time, we will try to focus on the most important features for the end user. We will use OpenZFS[1] as it is arguably the most widely used implementation that is actively maintained.

[1] `http://open-zfs.org/wiki/Main_Page`

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Introduction

ZFS[2] was developed at Sun Microsystems (later bought by Oracle) as a completely new filesystem and volume manager (for RAID) and released as open source before the Oracle takeover. Many of the former ZFS developers quit their jobs at Oracle shortly after the acquisition, took the latest open source version of ZFS and founded the OpenZFS project.
ZFS is intended for use in storage systems and servers where huge amounts of data must be stored. However, it can also be used for a Unix desktop at home with comparably small storage requirements. ZFS' features make it appealing for a number of different use cases, big and small.
The ZFS designers focused on the following points during development:

- Data integrity - detect and correct data corruption and errors
- Storage capacity - the first 128 bit filesystem
- Simple administration - 2 commands are sufficient to manage the storage
- High Performance

[2] previous name Zettabyte File System, today just referred to as Z filesystem

Problems with Today's Filesystems

Most filesystems are still based on the assumptions from 20 years ago, while at the same time, development of more sophisticated disk drives (faster access, higher capacities) has moved forward dramatically. With the availability of SSD (Solid State Disks), filesystems from earlier Unix eras need to be adapted to fit the storage requirements of today.
Some drawbacks from current filesystems are listed here:

Silent data corruption: Errors can happen at many points of the I/O chain before data is safely stored. They mostly go unnoticed, so that corrupt data will be detected only once it is accessed again, at which time it is already too late. Problems can occur in disks, controllers, cables, drivers or firmware, or RAM without ECC[3] and therefore the proper storing of the data is prevented, without the user or operating system getting notified about it.

[3] Error-Correcting Code

Problems with Today's Filesystems

Bad administrative tools: Many data losses are caused by users or administrators. One reason is that administering data is complicated. There are labels, partitions, volumes, config files like `/etc/fstab`, that need to be properly configured to make use of the storage. The sheer number of administrative commands and parameters does not contribute to make things easier and increases the potential for errors instead.

Many limits: Volume- and file sizes, maximum number of files, files per directory, number of snapshots and other details are always limited to a specific number.

Slow processing: Locking, fixed block sized, slow writes of non-contiguous files slow down I/O performance.

Especially in the server space these factors play an important role to make data available quickly, efficiently and securely.
ZFS was developed to fix the problems mentioned above.

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Features of ZFS

ZFS was developed from scratch to fit today's requirements for performance and data integrity. During development, a special focus was put on:

- Pool-based storage - a storage pool is managed on which the whole storage logic and administration is based
- Data integrity from beginning to the end - data is verified and corrected if necessary using checksums at each point of the storage chain (from RAM down to the disks)
- Transactional operations - this model, known from databases, allows constant consistency as well as high performance
- Copy-on-write (COW) - existing data is never overwritten, but stored at a new location. Due to that, the data is always consistent and no `fsck` is needed.

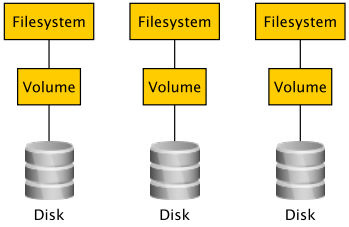
ZFS Requirements

ZFS has somewhat higher system requirements than other filesystems. Main memory is especially important for ZFS, since much of the data is pre-sorted in RAM and processed in such a way to be written to disk in an orderly fashion. On 32-bit systems it is not recommended to use ZFS for production workloads due to the limits of this architecture to a maximum of 4 GB RAM. 64-bit systems should be used at minimum and $>=$ 4 GB RAM *just for ZFS* should be reserved. The performance benefit is well worth it, since the filesystem is used to manage tertiary storage (disks) which is usually the slowest part of a computer system. Deduplication is especially memory consuming. For that, it is recommended to use $>$ 2 GB RAM *for each TB* of deduplicated storage.
Using SSDs as write or read cache allows ZFS to become even more performant (so called hybrid storage pool).
ZFS is *not* a cluster or distributed filesystem and was not developed to be one. For these use cases, ZFS is not well suited.

Traditional Storage Architectures vs. ZFS

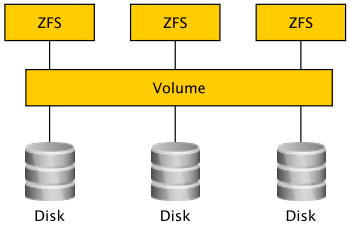
Traditional Architecture

- Partition(s)/Volume per filesystem
- Fragmented capacity and I/O bandwidth
- Manual resizing and growing the storage

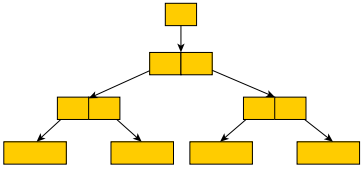


ZFS Architecture

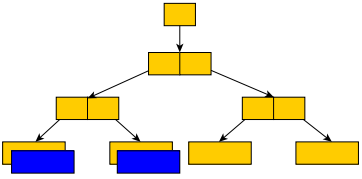
- No partitions required
- Assignment of the total storage to filesystems is flexible
- Integrated error checking
- Snapshots, clones, compression, deduplication



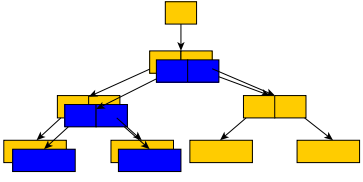
Copy on Write (COW) Transactional Filesystem



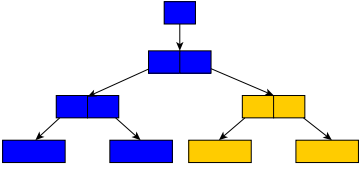
1. Consistent state



2. Write some files



3. Write metadata



4. Write the uberblock (commit)

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Simple Administration

To manage ZFS, only two commands and their options are necessary: `zpool` and `zfs`. The `zpool` command manages the underlying storage pool and connected devices. With the `zfs` command one can set all kinds of parameters for the filesystems on top of the pool. A number of subcommands allow access to all features of ZFS, some of which we will look at in more detail.

Simple Administration – Creating a Storage Pool

To create a storage pool from a single disk the following command is used:

```
# zpool create tank /dev/ada1
```

The name `tank` (can be chosen at will) is used to access the pool in the future and abstracts to devices within it. Issuing this command causes a number of further actions to happen automatically:

- Creation of a filesystem with the same name
- Mounting the filesystem under `/tank` (or whatever the name is)
- The settings are stored as part of the pool and will be reused upon reboot
- The storage is immediately available (no lengthy formatting required)

This way, possible errors in `/etc/fstab` do not happen because it is not used. In a ZFS-only system this file is completely empty. All data used for configuration is part of the pool.

Simple Administration – Displaying Pool Status

The status of the storage pool can be displayed via `zpool status`. A healthy pool looks like this:

```
# zpool status
  pool: tank
 state: ONLINE
  scan: none requested
config:

NAME         STATE     READ WRITE CKSUM
tank         ONLINE       0     0     0
  ada1       ONLINE       0     0     0

errors: No known data errors
```

All pools and their capacity can be displayed using `zpool list`.

```
# zpool list
NAME   SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
tank  1016M  89.5K  1016M     0%  1.00x  ONLINE  -
```

Simple Administration – Displaying I/O Statistics

ZFS contains a built-in tool to display I/O statistics (`iostat`) of pools. It shows the free and allocated space as well as the size of read and write operations and their I/O bandwidth.

```
$ zpool iostat
               capacity     operations     bandwidth
pool         alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
tank         62.4G  3.56T      0      3  1.19K  15.6K
```

By specifying an interval in seconds as a parameter, statistics are displayed continuously until the user interrupts it using Ctrl + C.
An even more detailed display allows the `-v` (verbose) option. I/O will then be displayed for individual devices that are part of the pool.

Simple Administration – Creating Stripes

A pool with just one disk does not provide any redundancy, capacity or even adequate performance. Due to these reasons, stripes are used because they offer higher capacity and performance, but **no redundancy**:

```
# zpool create mystripe /dev/ada1 /dev/ada2
```

```
# zpool status
  pool: mystripe
 state: ONLINE
  scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        mystripe      ONLINE       0     0     0
          ada1        ONLINE       0     0     0
          ada2        ONLINE       0     0     0

errors: No known data errors
# zpool list
NAME       SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mystripe  1.98G  92.5K  1.98G     0%  1.00x  ONLINE  -
```

Simple Administration – Creating Mirrors (RAID1)

Mirrored disks have less capacity, but ensure redundancy for disk failures and reads can happen in parallel.

```
# zpool create mymirror mirror /dev/ada1 /dev/ada2
```

Using the keyword `mirror`, ZFS is instructed to create a mirror from the specified disks.

```
# zpool status
  pool: mymirror
 state: ONLINE
  scan: none requested
config:

        NAME              STATE     READ WRITE CKSUM
        mymirror          ONLINE       0     0     0
          mirror-0        ONLINE       0     0     0
            ada1          ONLINE       0     0     0
            ada2          ONLINE       0     0     0

errors: No known data errors
# zpool list
NAME        SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mymirror   1016M  92.5K  1016M     0%  1.00x  ONLINE  -
```

Simple Administration – Increase Storage Pool Capacity 1/2

To add more devices to the pool and therefore increase overall capacity without downtime, the command `zpool add` is used. At the moment, our pool looks like this:

```
# zpool status
  pool: mypool
 state: ONLINE
  scan: none requested
config:

        NAME         STATE     READ WRITE CKSUM
        mypool       ONLINE       0     0     0
          ada1       ONLINE       0     0     0

errors: No known data errors
# zpool list
NAME     SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mypool   960M    91K   960M     0%  1.00x  ONLINE  -
```

Simple Administration – Increase Storage Pool Capacity 2/2

We add another disk to the pool, so that the data is striped (default setting) over both disks. This increases performance, but is **not** redundant. If any one disk dies, all data in the pool is lost!

```
# zpool add mypool /dev/ada2
# zpool list mypool
  NAME     SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
  mypool  1.90G   145K  1.90G     0%  1.00x  ONLINE  -
# zpool status
  pool: mypool
 state: ONLINE
  scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        mypool      ONLINE       0     0     0
          ada1      ONLINE       0     0     0
          ada2      ONLINE       0     0     0

errors: No known data errors
```

The newly gained storage space is available immediately without any further configuration.

Simple Administration – Create Mirror from Pool 1/3

A mirror can be created from a preexisting pool that has only one device. For that the command `zpool attach` must be used. In order for the new device to mirror the data of the already existing device the pool needs to be *resilvered*. This means that the pool synchronizes both devices to contain the same data at the end of the resilver operation. During this process access to the pool is slower, but still possible.

Simple Administration – Create Mirror from Pool 2/3

```
# zpool attach mypool mirror /dev/da4 /dev/da5
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mypool    960M  44.2M   916M     4%  1.00x  ONLINE  -
#
# zpool status mypool
  pool: mypool
 state: ONLINE
status: One or more devices is currently being resilvered.  The pool
        will continue to function, possibly in a degraded state.
action: Wait for the resilver to complete.
  scan: resilver in progress since Mon Nov  5 10:09:42 2012
        5.03M scanned out of 44.1M at 396K/s, 0h1m to go
        5.03M resilvered, 11.39% done
config:

NAME         STATE     READ WRITE CKSUM
mypool       ONLINE       0     0     0
  mirror-0   ONLINE       0     0     0
    da4      ONLINE       0     0     0
    da5      ONLINE       0     0     0  (resilvering)

errors: No known data errors
```

Simple Administration – Create Mirror from Pool 3/3

After the resilver process has finished the status of the pool looks like this:

```
# zpool status
  pool: mypool
 state: ONLINE
  scan: resilvered 44.2M in 0h1m with 0 errors on Mon Nov  5 10:11:34 2012
config:

NAME         STATE     READ WRITE CKSUM
mypool       ONLINE       0     0     0
  mirror-0   ONLINE       0     0     0
    da4      ONLINE       0     0     0
    da5      ONLINE       0     0     0

errors: No known data errors
```

Simple Administration – Create Filesystems/Datasets

After the pool has been created with the underlying devices, filesystems[4] can be created using `zfs create`. Instead of directories, whole datasets can be created that provide individual options.

```
# zfs create tank/home
```

The newly created dataset is available immediately with the full disk space capacity of the pool, unless other options are specified (more about that later).

```
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         146K   984M    31K  /tank
tank/home     31K   984M    31K  /tank/home
```

[4] ZFS uses the term dataset, which we're going to use as well from now on. Datasets can be used synonymously with traditional filesystems, although datasets have more features.

Simple Administration – Display and Edit Properties

Various ZFS properties can be manipulated using `zfs set`.

```
# zfs set atime=off tank
```

All currently available properties can be displayed with the `zfs get all` command. To see a specific property and the current value, the name of the property can be provided after `zfs get`:

```
# zfs get atime
NAME        PROPERTY  VALUE  SOURCE
tank        atime     off    local
tank/home   atime     off    inherited from tank
```

Just like in object-oriented programming, inheritances are hierarchical where higher objects pass on their properties to the ones below them, hence the description `inherited from tank` in the `SOURCE` column of the `tank/home` dataset. This way, a lot of typing is not necessary anymore and properties can be set for the whole pool, no matter how many datasets there are. Of course, exceptions can be made and changes can be applied to specific datasets only.

Simple Administration – Important Properties

`zfs get all` displays all currently available properties for a given dataset. Properties that have a - in their SOURCE column are read-only properties, like the creation date (`zfs get creation`) or the used space (`used`).

```
# zfs get creation,used tank
NAME   PROPERTY   VALUE                  SOURCE
tank   creation   Sun May  13 08:13 2012  -
tank   used       634M                   -
```

Other properties can be changed by the user. Many of those are set at dataset creation time by the parent dataset (inherited) or are default values (shown as `default`).

Simple Administration – Changing Mountpoints

By default, ZFS mounts all new datasets below the name of pool, which forms the root of that pool.

```
# zfs create tank/home
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         146K   984M    31K  /tank
tank/home     31K   984M    31K  /tank/home
```

When a different path should be used, then the following command can change that:

```
# zfs set mountpoint=/usr/home tank/home
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         146K   984M    31K  /tank
tank/home     31K   984M    31K  /usr/home
```

Note that the mountpoint must be specified with a leading / (as usual in Unix), but the ZFS path in the pool must not have a leading slash.

Simple Administration – helpful properties: exec 1/2

The exec property defines whether or not files can be executed on that dataset. This makes sense in some datasets where executing files would do more harm than good, like in /var/log. This increases system security when this option is set for i.e. datasets for the user homes.

```
# zfs create tank/home
# zfs set exec=off tank/home
# zfs create tank/home/susan
# zfs get exec
NAME             PROPERTY  VALUE  SOURCE
tank             exec      on     default
tank/home        exec      off    local
tank/home/susan  exec      off    inherited from tank/home
```

Simple Administration – helpful properties: exec 2/2

From the outside, the ZFS property is not visible and the execute (x) flag can be set as usual.

```
$ ls -lah /tank/home
total 5
drwxr-xr-x  3 root  wheel     3B Dec  2 23:01 .
drwxr-xr-x  6 root  wheel     6B Dec  2 22:59 ..
drwx------  2 susan wheel     2B Dec  2 23:00 susan
$ cd susan
susan$ cat myscript.sh
#!/bin/sh
date ; ls
susan$ chmod +x myscript.sh
susan$ ls -l myscript.sh
-rwx--x--x  1 root  wheel  21 Dec  2 23:12 myscript.sh
susan$ ./myscript.sh
unable to execute ./myscript.sh: Permission denied
```

Even `root` is not allowed to execute files on this dataset, as long as the `exec` property is set to `off`.

Simple Administration – helpful properties: `readonly`

Some datasets should not be written at all and only be read by the users. For this use case the property `readonly` is used. By default, this property is deactivated, meaning that datasets are mounted read-write.

```
# zfs create -p projects/current
# zfs create projects/finished
# cp -R /home/susan/projects /projects/current
# zfs get readonly projects/finished
NAME                PROPERTY  VALUE   SOURCE
projects/finished   readonly  off     local
# cp /projects/current/photosafari /projects/finished
# zfs set readonly=on projects/finished
# zfs get readonly tank/projects/finished
NAME                PROPERTY  VALUE   SOURCE
projects/finished   readonly  on      local
# cp -R /projects/current/miscphotos /projects/finished
cp: /projects/finished/miscphotos: Read-only file system
```

Backup and archive data are typical candidates for the `readonly` property to protect them from accidental (or intentional) deletion.

Simple Administration – User-defined Properties

Beside the properties set by ZFS, the users can set custom properties and save it as part of the pool configuration.
The syntax is as follows:

```
zfs set name1:name2=value pool
```

The colon is required to distinguish user-defined variables from the ones that ZFS defines. These custom-properties are helpful when, for example, the date of the last backup should be recorded or which cost center is paying for the underlying storage disks. Some automatic snapshot solutions rely on these properties to ensure proper functionality. These user-defined properties can be inherited or set on an individual dataset.

```
# zfs set hda:firstlecture=20.04.2013
# zfs get hda:firstlecture
NAME   PROPERTY           VALUE              SOURCE
tank   hda:firstlecture   20.04.2013         local
```

Destroying Datasets

Datasets, pools, snapshots and clones can be removed using `zfs destroy` or `zpool destroy`. All data on these objects is lost. When child-datasets exist ZFS warns the user and offers an option to recursively walk through the child objects.

```
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         146K   984M    31K  /tank
tank/home     31K   984M    31K  /tank/home
# zfs destroy tank
cannot destroy 'tank': filesystem has children
use '-r' to destroy the following datasets:
tank/home
```

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Quota and Reservations

As we've seen already, the full capacity of the pool is available to all ZFS datasets. Traditional filesystems used partitions to limit the amount of disk space. Since there are no partitions in ZFS, there must be another mechanism to limit the available disk space per dataset. ZFS uses **quotas** that are being used in other filesystems as well to dynamically restrict disk space usage. This way, home directories for example (more like home datasets) can have a size limit to prevent a user from completely taking up all the available disk space and take it away from other users on the system.
Another case is when a specific amount of disk space should be **reserved** for a user or a dataset. ZFS guarantees that the specified amount of space will be available no matter how much other users have already used up. Of course, reservations can not go over the total amount of disk space available in the system.

Using ZFS Quotas to Limit Disk Space 1/3

We want to set a quota for the previously created home datasets. By default, every user can make use of all the disk space that the pool provides, there is no quota applied. At the moment, disk space usage is as follows:

```
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                1.3G      31k     1.3G     0%    /hda
hda/home           1.3G      97k     1.3G     0%    /usr/home
hda/home/bcr       1.3G      84k     1.3G     0%    /usr/home/bcr
$ zfs get quota
NAME                    PROPERTY  VALUE  SOURCE
hda                     quota     none   default
hda/home                quota     none   default
hda/home/bcr            quota     none   default
```

Using ZFS Quotas to Limit Disk Space 2/3

We now set a quota of 500 MB on `hda/home/bcr` using the `zfs set quota` command.

```
# zfs set quota=500m hda/home/bcr
```

```
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                1.3G      31k     1.3G     0%    /hda
hda/home           1.3G      97k     1.3G     0%    /usr/home
hda/home/bcr       500M      84k     499M     0%    /usr/home/bcr
$ zfs get quota
NAME                    PROPERTY  VALUE  SOURCE
hda                     quota     none   default
hda/home                quota     none   default
hda/home/bcr            quota     500M   local
```

The quota does only apply to the specified dataset. By using inheritance, the quota can be set globally for all home directories. This way, datasets that are being created in the future will automatically inherit these size limits.

Using ZFS Quotas to Limit Disk Space 3/3

We now write a big file (one that is bigger than the available disk space) in `hda/home/bcr` to test whether the quota is enforced.

```
# dd if=/dev/urandom of=./bigfile bs=1g
...
dd: ./bigfile: Disc quota exceeded
1+0 records in
0+1 records out
512020480 bytes transferred in 94.152408 secs (5438209 bytes/sec)
% ls -lah bigfile
-rw-r--r--  1 bcr  bcr   499M Nov 30 15:46 bigfile
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                1.3G      31k     1.3G      0%    /hda
hda/home           1.3G      97k     1.3G      0%    /usr/home
hda/home/bcr       500M     500M       0B    100%    /usr/home/bcr
```

The quota rules are strictly enforced by ZFS. There can be no more space allocated than specified in the quota. The user either has to delete some old files, compress them or ask for a quota increase.

A quota can be deactivated using `zfs set quota=none` on the dataset.

ZFS Reservations for Disk Space Guarantees 1/3

Often, it is unpredictable how much space a specific partition is going to use. This is especially bad for non-ZFS filesystems when the allocated space is not enough and a reformatting and reinstallation is required just because the initial assumptions were wrong.
Reservations can be set in ZFS to give a specific amount of disk space to a dataset no matter how much other datasets require. The syntax for reservations is:

```
zfs set reservation=size
```

ZFS Reservations for Disk Space Guarantees 2/3

We want to ensure that the home dataset has a guaranteed 500 MB of disk space available. This is what we currently have:

```
$ df -h
Filesystem         Size    Used   Avail Capacity  Mounted on
hda                1.3G     31k    1.3G     0%    /hda
hda/home           1.3G     97k    1.3G     0%    /usr/home
hda/home/bcr       1.3G     84k    1.3G     0%    /usr/home/bcr
# zfs set reservation=500m hda/home/bcr
$ df -h
Filesystem         Size    Used   Avail Capacity  Mounted on
hda                801M     31k    801M     0%    /hda
hda/home           801M     97k    801M     0%    /usr/home
hda/home/bcr       1.3G     82k    1.3G     0%    /usr/home/bcr
$ zfs get reservation
NAME                    PROPERTY     VALUE    SOURCE
hda                     reservation  none     default
hda/home                reservation  none     default
hda/home/bcr            reservation  500M     local
```

The output of `df` is not very accurate and can be confusing at first. The total pool capacity has been reduced by 500 MB, but the dataset with the reservation still reports the original 1.3 GB space available.

ZFS Reservations for Disk Space Guarantees 3/3

```
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                801M      31k     801M      0%    /hda
hda/home           801M      97k     801M      0%    /usr/home
hda/home/bcr       1.3G      82k     1.3G      0%    /usr/home/bcr
```

When other datasets allocate more disk space the size of `hda/home/bcr` is decreased as well. However, 500 MB will still be available to that dataset.

```
# zfs create hda/home/myboss
myboss$ dd if=/dev/urandom of=./hda/home/myboss/bigfile bs=700m
dd: ./bigfile: No space left on device
2+0 records in
1+1 records out
781864960 bytes transferred in 176.822419 secs (4421752 bytes/sec)
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                  0M      31k       0M   100%    /hda
hda/home             0M      97k       0M   100%    /usr/home
hda/home/bcr       500M      82k     500M     0%    /usr/home/bcr
hda/home/myboss      0M     700M       0M   100%    /usr/home/myboss
```

Dataset reservations ensure disk space is available even in situations where pool space is short.

Combining ZFS Quotas and Reservations

ZFS quotas and reservations can be combined. This way, a reservation of disk space is ensured while at the same time ZFS takes care that only that specific amount of reserved disk space is being used, but not more.

```
# LIMIT=500m
# zfs set quota=$LIMIT hda/home/bcr
# zfs set reservation=$LIMIT hda/home/bcr
$ df -h
Filesystem         Size     Used    Avail Capacity  Mounted on
hda                791M      31k     791M     0%    /hda
hda/home           791M      97k     791M     0%    /usr/home
hda/home/bcr       500M      82k     500M     0%    /usr/home/bcr
```

This output corresponds to the output of traditional filesystem partitions with regard to space usage.

Overview

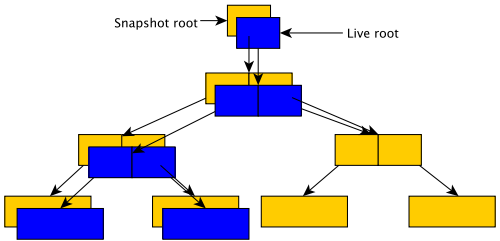
1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Snapshots

A **snapshot** is a read-only copy of a dataset or volume. They are created quickly and can be used for a wide variety of usage scenarios. Snapshots in ZFS can be taken only on whole datasets, not individual files or directories. When snapshotting a dataset, all files and directories contained within it are included in the resulting snapshot.
Snapshots are particularly useful as a quick backup before a potentially risky action. When a snapshot is taken *before* an action such as a system upgrade, software installations, tests involving the `rm` command, etc. and there is an error or mistake the snapshot can be rolled back. The dataset then returns to the point in time when the snapshot was taken (without rebooting the system) as if the action never happened.

ZFS Snapshots

Snapshots are created implicitly through the underlying copy-on-write (COW) model each time by copying parts of the filesystem tree. When a snapshot is created, ZFS preserves the old version of the filesystem state for later use. When no snapshot is created then the copy created by the COW model is discarded. That is why creating snapshots *does not cost anything*, because they are basically a by-product of the normal filesystem operation and available immediately. Since only changes are saved in the snapshot, they save a big amount of storage space.



Creating Snapshots

To create snapshots in ZFS, use the following syntax:

```
zfs snapshot dataset@name
```

`name` can be any identifier (i.e. date and time). The snapshot is created immediately. All snapshots can be shown by using the parameter `-t snapshot` with `zfs list`.

```
# zfs snapshot hda/home/bcr@backup
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr@backup        0      -  85.5K  -
```

The display shows that the snapshot is not mounted on the filesystem, which is why there is no path specified below `MOUNTPOINT`. There is also no mention of available disk space (`AVAIL` column), since snapshots can not be written (read-only).

Working with Snapshots

When comparing a snapshot with the dataset it is based on, it becomes clear how it was created.

```
# zfs list -rt all hda/home/bcr
NAME                  USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr         85.5K  1.29G  85.5K  /usr/home/bcr
hda/home/bcr@backup      0      -  85.5K  -
```

Another feature of ZFS snapshots is shown here. Snapshots save only the changes (delta) after the time they were taken and the previous (if any) snapshot and not the complete contents of the filesystem all over again to save space. This means that another snapshot of an unchanged dataset does not require any additional space!

```
# cp /etc/passwd /usr/home/bcr
# zfs snapshot hda/home/bcr@after_cp
# zfs list -rt all hda/home/bcr
NAME                    USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr            115K  1.29G    88K  /usr/home/bcr
hda/home/bcr@backup      27K      -  85.5K  -
hda/home/bcr@after_cp      0      -    88K  -
```

Working with Snapshots - Differences Between Snapshots

When the user wants to know the differences between two snapshots, then ZFS can display them using the `zfs diff` command. For our example it would look like this:

```
# zfs diff hda/home/bcr@backup
M /usr/home/bcr/
M /usr/home/bcr/.histfile
+ /usr/home/bcr/passwd
```

The following table describes the status column:

Character	Type of Change
+	File was added
−	File was deleted
M	File was changed
R	File was renamed

Working with Snapshots - Rollback

When it becomes necessary to roll back to the state that a snapshot holds, a command similar to the one known from database systems is used: `zfs rollback`.
The syntax is:

```
zfs rollback snapshot
```

When a snapshot is rolled back all files that were changed since the creation of that snapshot are discarded and the filesystem returns to the state that it had at the time when the snapshot was taken.
By default, snapshots will roll back to the most current (i.e.: the latest) one. To roll back to an older snapshot than the current one, the snapshots in between must be destroyed. For that, the option `-r` (recursive) must be used, otherwise ZFS will issue a warning.
When a snapshot that lies between the current state of the filesystem and the one that should be rolled back to should be preserved, a clone must be created. See the section on clones for more information.

Working with Snapshots - Rollback Example

In this example we roll back to the last snapshot because of an accidental delete of an important file.

```
# rm /hda/home/bcr/*
# ls
# zfs rollback hda/home/bcr@after_cp
# ls
passwd loveletter.txt
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr@backup      27K      -  85.5K  -
hda/home/bcr@after_cp      0      -    88K  -
```

The rollback operation has succeeded and all files were restored from the snapshot!

Working with Snapshots - Rollback To First Snapshot

Now we want to roll back to the very first snapshot, because we don't want to have the file `passwd` around anymore[5].

```
# ls
passwd loveletter.txt
# zfs list -t snapshot
NAME                      USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr@backup       27K      -  85.5K  -
hda/home/bcr@after_cp       0      -    88K  -
# rm /hda/home/bcr/*
# ls
# zfs rollback hda/home/bcr@backup
cannot rollback to 'hda/home/bcr@backup': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
hda/home/bcr@after_cp
# zfs rollback -r hda/home/bcr@backup
# ls
loveletter.txt
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
hda/home/bcr@backup     27K      -  85.5K  -
```

[5] and we are too lazy to just delete it. ;-)

Working with Snapshots - Restoring a File

What about situations where not the whole snapshot should be rolled back, but a few files from that state should be restored? In this case, ZFS keeps a special hidden (even a `ls -a` won't display it) directory `.zfs` on every dataset. This behavior can be changed by setting the `snapdir` property to `visible`: `zfs set snapdir=visible pool`.
We assume for this example that the snapshot from the previous slide was not rolled back.

```
# ls .zfs/snapshot
after_cp backup
# ls .zfs/snapshot/after_cp
passwd
# cp .zfs/snapshot/after_cp/passwd /hda/home/bcr
```

Single files or directories can be copied from this hidden directory. Copying files *into* that directory will fail due to the read-only nature of snapshots.

```
# cp /hda/home/bcr/loveletter.txt .zfs/snapshot/after_cp/passwd
cp: .zfs/snapshot/after_cp/loveletter.txt: Read-only file system
```

Working with Snapshots - Cloning Snapshots

Clones can be created from snapshots, which represent a *writeable* copy of a snapshot and are treated as a self-contained dataset. To create a clone, a snapshot must exist and the following syntax must be used:

```
zfs clone snapshot newdataset
```

When a clone is created, ZFS creates an implicit dependency between clone and the snapshot. Due to that, the snapshot can not be destroyed after a clone has been created that depends on it.
To remove this dependency, clones can be "promoted" to a real dataset using the command `zfs promote nameoftheclone`. The snapshot and clone have switched their dependency around and the snapshot can then be destroyed. The clone can be mounted at any position within the ZFS filesystem, it must not be located at the original path of the snapshot.

Working with Snapshots - Creating Clones from Snapshots

We assume the following filesystem is present:

```
# zfs list -rt all hda/home/bcr
NAME                          USED  AVAIL  REFER  MOUNTPOINT
geonosis/home/boba            108K  1.29G    87K  /geonosis/home/boba
geonosis/home/boba@backup      21K      -  85.5K  -
geonosis/home/boba@after_cp     0      -    87K  -
```

A clone should now be created based on the state of the filesystem after the last snapshot.

```
# zfs clone geonosis/home/boba@after_cp geonosis/home/trooper
# ls /geonosis/home/*
geonosis/home/boba
darthsidious.txt
geonosis/home/trooper
darthsidious.txt
# df -h
Filesystem                Size     Used    Avail Capacity  Mounted on
geonosis                  1.3G      31k     1.3G      0%    /geonosis
geonosis/home             1.3G      98k     1.3G      0%    /geonosis/home
geonosis/home/boba        1.3G      87k     1.3G      0%    /geonosis/home/boba
geonosis/home/trooper     1.3G      87k     1.3G      0%    /geonosis/home/trooper
```

Working with Snapshots - Promoting Clones

By cloning the created dataset it inherits all properties that the snapshot had. For demonstration purposes, we copy a file into the writeable clone.

```
# cp neworders.txt /geonosis/home/trooper
# zfs get origin geonosis/home/trooper
NAME                    PROPERTY  VALUE                          SOURCE
geonosis/home/trooper   origin    geonosis/home/boba@after_cp    -
# zfs promote geonosis/home/trooper
# zfs get origin geonosis/home/trooper
NAME                    PROPERTY  VALUE    SOURCE
geonosis/home/trooper   origin    -        -
```

The clone should now serve as the new home directory. The clone must not have the same name as any existing snapshot. Using `zfs rename` datasets can be renamed.

```
# zfs destroy -f geonosis/home/boba
# zfs rename geonosis/home/trooper geonosis/home/stormtrooper
# ls /geonosis/home/stormtrooper
darthsidious.txt neworders.txt
```

We got the file `neworders.txt` from the `geonosis/home/trooper` clone.

Overview

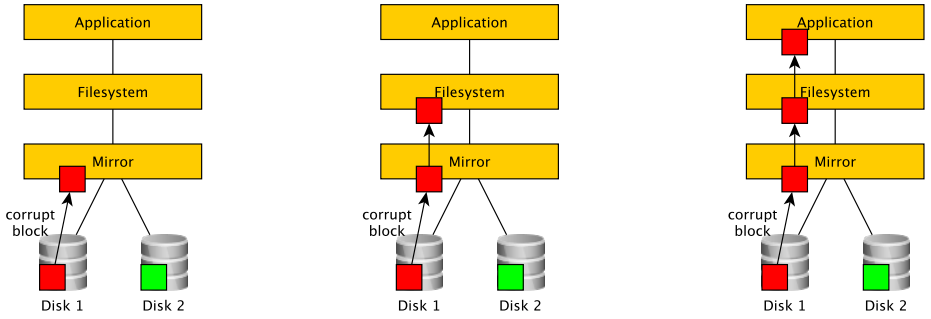
1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Selfhealing Data

As previously discussed, traditional filesystems do not necessarily perform their operations atomically. Because of that, inconsistent metadata is present after a system crash that must be resolved using `fsck` before mounting them again. This results in long offline times for the storage and errors like the loss of non-persistent data in RAM (which has not been written yet) as well as inconsistent mirror devices (so called split-brain) can not be avoided completely. ZFS on the other hand is able to detect and correct corrupted data based on checksums during normal operation. An `fsck` is not necessary, the checks for consistency via `zpool scrub` can be performed in hours where the storage is not needed that much (i.e. at night). This way, no downtimes must be scheduled and the data remains available all the time. Read-/Write operations are slowed down, but are still possible during the scrub operation.

Traditional Filesystems Do Not Detect All Errors

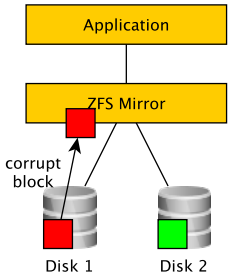
Especially in traditional redundantly held data (i.e. RAID1), the problem of silent data corruption like bit-flips is evident.



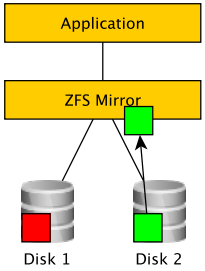
1. Read corrupt block
2. Corrupt metadata in filesystem
3. Corrupt data to application

Traditional filesystems can not detect and correct silent-data corruption! Some of these data are detected only years later when they are accessed again. Backups are especially at risk.

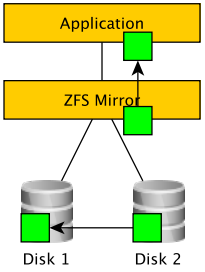
Selfhealing Data in ZFS via Redundancy



1. Reading corrupt block



2. ZFS detects wrong checksum, reads mirror with correct checksum



3. Correct data is supplied by the mirror partner device to application *and* corrupt block is corrected!

ZFS Selfhealing Demo

This demonstrates the example of the previous slide on a real ZFS pool. To do that, we create a mirror and copy some data to it. A checksum is computed as a reference for the correct pool data. The pool is exported afterwards and on one device of the pool some random data is written to simulate the data corruption. After that, the pool is imported again and the pool status is displayed. A second checksum is created to compare it with the first one. When they match, then ZFS properly detected and corrected the data automatically using the other device in the mirror.
Note that this only works when there is enough redundancy in the pool already. Either by using a redundant setup with enough devices (RAID-1, RAIDZ) or by setting the `copies` property to 2 or higher *before* data is written to the pool.

ZFS Selfhealing Demo 1/5

```
# zpool create healer mirror da0 da1
# zpool status
  pool: healer
 state: ONLINE
  scan: none requested
config:

 NAME         STATE     READ WRITE CKSUM
 healer       ONLINE       0     0     0
   mirror-0   ONLINE       0     0     0
     da0      ONLINE       0     0     0
     da1      ONLINE       0     0     0

 errors: No known data errors
# zpool list
NAME     SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
healer   960M  92.5K   960M     0%  1.00x  ONLINE  -
# cp /some/important/data /healer
# zfs list
NAME     SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
healer   960M  67.7M   892M     7%  1.00x  ONLINE  -
# sha1 /healer > checksum.txt
# cat checksum.txt
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
```

ZFS Selfhealing Demo 2/5

Warning:

This dd example can destroy data when used on the wrong device or a non-ZFS filesystem.
Use it at your own risk and make proper backups first!

```
# zpool export healer
# dd if=/dev/random of=/dev/da1 bs=1m count=200
200+0 records in
200+0 records out
209715200 bytes transferred in 62.992162 secs (3329227 bytes/sec)
# zpool import healer
```

ZFS Selfhealing Demo 3/5

```
# zpool status
  pool: healer
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
        attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
        using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
  scan: none requested
config:

 NAME        STATE     READ WRITE CKSUM
 healer      ONLINE       0     0     0
   mirror-0  ONLINE       0     0     0
     da0     ONLINE       0     0     0
     da1     ONLINE       0     0     1

 errors: No known data errors
```

We compare the checksums of the pool before and after the import.

```
# sha1 /healer >> checksum.txt
# cat checksum.txt
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
```

ZFS Selfhealing Demo 4/5

```
# zpool scrub healer
# zpool status
  pool: healer
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
        attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
        using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
  scan: scrub in progress since Mon Dec 10 12:23:30 2012
        10.4M scanned out of 67.0M at 267K/s, 0h3m to go
        9.63M repaired, 15.56% done
config:

 NAME         STATE     READ WRITE CKSUM
 healer       ONLINE       0     0     0
   mirror-0   ONLINE       0     0     0
     da0      ONLINE       0     0     0
     da1      ONLINE       0     0   627  (repairing)

errors: No known data errors
...
```

ZFS Selfhealing Demo 5/5

```
# zpool status
  pool: healer
 state: ONLINE
status: One or more devices has experienced an unrecoverable error.  An
        attempt was made to correct the error.  Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
        using 'zpool clear' or replace the device with 'zpool replace'.
   see: http://www.sun.com/msg/ZFS-8000-9P
  scan: scrub repaired 66.5M in 0h2m with 0 errors on Mon Dec 10 12:26:25 2012
config:

 NAME        STATE     READ WRITE CKSUM
 healer      ONLINE       0     0     0
   mirror-0  ONLINE       0     0     0
     da0     ONLINE       0     0     0
     da1     ONLINE       0     0 2.72K

 errors: No known data errors
# zpool clear healer
# zpool status
  pool: healer
 state: ONLINE
  scan: scrub repaired 66.5M in 0h2m with 0 errors on Mon Dec 10 12:26:25 2012
config:

 NAME        STATE     READ WRITE CKSUM
 healer      ONLINE       0     0     0
   mirror-0  ONLINE       0     0     0
     da0     ONLINE       0     0     0
     da1     ONLINE       0     0     0

 errors: No known data errors
```

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Compression

A classic way to save disk space is to compress files using tools like `zip` and others. ZFS can compress individual datasets and transparently compresses data when saved and decompresses it when loaded automatically. There is no need to go to an extra tool to do that and the filesystem can store more data in the process.
When compression is activated for a ZFS dataset then *all new data stored* is compressed. Pre-existing data on that dataset retains their original size, except when it is modified or otherwise stored on it again while compression is active.
A number of compression algorithms are available that have different results depending on the type of data to be compressed. Of course, compression requires a little more load on the CPU, but this should not be too heavy of an impact in today's multi-processor systems.

ZFS Compression - Usage Example 1/2

The FreeBSD Ports Tree contains a lot of text files (Makefiles, patches, description files, etc.), which can be compressed very well. First, we create the necessary directory structure:

```
# zfs create tank/usr
# zfs create tank/usr/ports
# zfs create -o mountpoint=/usr/ports tank/usr/ports
```

Then we set the compression to the GZIP algorithm; this could have been done when we created the datasets above using the -o option.

```
# zfs set compression=gzip tank/usr/ports
# zfs get compressratio tank/usr/ports
NAME            PROPERTY       VALUE  SOURCE
tank/usr/ports  compressratio  1.00x  -
```

Now we download a current copy of the ports tree into /usr/ports.

```
# portsnap fetch extract
```

After this operation is finished, we check the compression ratio again.

```
# zfs get compressratio tank/usr/ports
NAME            PROPERTY       VALUE  SOURCE
tank/usr/ports  compressratio  3.02x  -
```

ZFS Compression - Usage Example 2/2

To check the space savings, we create another uncompressed dataset for comparison:

```
# zfs create -o compression=off tank/usr/uncompressed
# zfs get compressratio,compression tank/usr/uncompressed
NAME                      PROPERTY       VALUE  SOURCE
tank/usr/uncompressed     compressratio  1.00x  -
tank/usr/uncompressed     compression    off    local
```

Then we copy the source tree from `/usr/ports` to the new dataset.

```
# cp -R /usr/ports/* /usr/uncompressed
# zfs get compressratio,compression tank/usr/uncompressed
NAME                      PROPERTY       VALUE  SOURCE
tank/usr/uncompressed     compressratio  1.00x  -
tank/usr/uncompressed     compression    off    local
```

We can compare space usage more easily now (unimportant fields stripped from output).

```
# zfs list -o space tank/usr/uncompressed tank/usr/ports
NAME                     AVAIL   USED    USEDDS
tank/usr/ports           2.50T   978M      978M
tank/usr/uncompressed    2.50T  2.95G     2.95G
```

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Deduplication

Data is often stored multiple times on data storage devices and filesystems. This happens because it is intentional (i.e. redundancy reasons), but also unintentionally or without the users knowledge (caching, temporary files). Some applications store data in separate hidden working directories. All of this data is taking up additional disk space for each copy stored.
Deduplication is a means to reduce the amount of data stored in this way. It can be viewed as the opposite to data redundancy.
In ZFS, deduplication is implemented at the block layer. When deduplication is activated on a ZFS pool and a duplicate block B is detected using checksums that is an exact copy of an already existing block A, a reference from B to A is stored instead. This reference does take up only a few bytes and is managed transparently to the user. This way, a lot of disk space can be saved when there is much duplicate data.

Using Deduplication

To activate deduplication on a pool, simply set it using the following command:

```
zfs set dedup=on pool
```

With this setting, a checksum of every new block is checked and if there is such a block already on the pool, then the new block is replaced by a reference to it. This type of checking is done in memory, which is why deduplication should not be activated on every pool. Blocks that were already written and are duplicates of one another are not deduplicated after the activation. Instead, the data must be copied again or changed (i.e. timestamps) to trigger the activation of the deduplication checks.
A pool on which deduplication has just been activated typically looks like this:

```
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mypool   2.84G  2.19M  2.83G     0%  1.00x  ONLINE  -
```

The `DEDUP` column shows the current rate of deduplication.

Deduplication Example

In this example data is stored in different directories on the same dataset, which resides on a pool with activated deduplication.

```
# zpool list
NAME       SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mypool    2.84G  2.19M  2.83G     0%  1.00x  ONLINE  -
# zfs get dedup mypool
NAME    PROPERTY  VALUE          SOURCE
mypool   dedup     on             local
# cd /mypool
# for d in dir1 dir2 dir3; do
for> mkdir $d && cp -R /usr/ports $d &
for> done
...
# zpool list
NAME       SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
mypool    2.84G  20.9M  2.82G     0%  3.00x  ONLINE  -
# df -h mypool
Filesystem     Size    Used   Avail Capacity  Mounted on
mypool         2.8G     54M    2.8G     2%    /mypool
```

Checking whether Deduplication is Beneficial

One can check whether it would be beneficial to activate ZFS' deduplication feature for a given set of data. For that, we use the ZFS debugger called `zdb`, which has an option to simulate the deduplication benefits.

```
# zdb -S mypool
Simulated DDT histogram:

    bucket              allocated                       referenced
    ------   ------------------------------   ------------------------------
    refcnt   blocks   LSIZE   PSIZE   DSIZE   blocks   LSIZE   PSIZE   DSIZE
    ------   ------   -----   -----   -----   ------   -----   -----   -----
         1    17.6K   32.6M   32.6M   32.6M    17.6K   32.6M   32.6M   32.6M
         2      102   73.5K   73.5K   73.5K      224    164K    164K    164K
         4        6      3K      3K      3K       30     15K     15K     15K
        16        1      1K      1K      1K       23     23K     23K     23K
     Total    17.7K   32.7M   32.7M   32.7M    17.9K   32.8M   32.8M   32.8M

     dedup = 1.00, compress = 1.00, copies = 1.00, dedup * compress / copies = 1.00
```

In this example, activating deduplication *will not be beneficial*. Factors that favor deduplication are an activated compression and the usage of the `copies` property that stores multiple copies of a block for redundancy reasons (in case of bad blocks on USB sticks, memory cards, etc.). These redundant copies are exactly what is needed for deduplication.

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

Outfitting non-ZFS Filesystems with ZFS Properties

Sometimes one cannot use ZFS as a filesystem for a specific task, but its features can still be beneficial. In this example, we will create a FAT32 filesystem to exchange data with Windows systems using a volume. A volume is exported as a block device and can be used as free, contiguous storage in this way. Afterwards, another filesystem (in our case: FAT32) can be created on this raw device. Additionally, we activate compression, a feature which FAT32 does not have, to gain more disk space.
Another use case for volumes is using them as swap devices. ZFS volumes can also be exported as iSCSI devices over the network to other computers. There, it will show up as free, *local* storage (iSCSI property).

Outfitting non-ZFS Filesystems with ZFS Properties

```
# zfs create -V 250m -o compression=on mypool/fat32
```

The volume does not show up as a separate ZFS dataset, therefore there is no `mypool/fat32` in this output:

```
# zfs list mypool
NAME     USED  AVAIL  REFER  MOUNTPOINT
mypool   258M   670M    31K  /mypool
```

Instead, it is stored in the path `/dev/zvol/mypool`.

```
# newfs_msdos -F32 /dev/zvol/mypool/fat32
# mount -t msdosfs /dev/zvol/mypool/fat32 /mnt
# df -h /mnt
Filesystem                Size    Used   Avail Capacity  Mounted on
/dev/zvol/mypool/fat32    249M     24k    249M     0%    /mnt
# mount
...
/dev/zvol/mypool/fat32 on /mnt (msdosfs, local)
```

The filesystem can now be used as a Windows shared folder without knowing that it is backed by ZFS and compressed.

Overview

1. Introduction
2. Problems with Today's Filesystems
3. Features of ZFS
   - Simple Administration
   - Quota and Reservations
   - Snapshots
   - Selfhealing Data
   - Compression
   - Deduplication
   - ZFS Volumes
   - ZFS Serialization

ZFS Serialization

ZFS contains built-in functionality to serialize the storage so that it can be sent as a byte stream to standard output. For that functionality, snapshots of the dataset are used. This way it is possible to transfer a dataset consistently between two systems running ZFS pools. The command is called `zfs send`.
The following two pools will be used to demonstrate this feature:

```
# zpool list
NAME     SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
backup   960M    77K   960M     0%  1.00x  ONLINE  -
mypool   984M  43.7M   940M     4%  1.00x  ONLINE  -
```

ZFS Serialization

The following two pools will be used to demonstrate this feature:

```
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
backup    960M    77K   960M     0%  1.00x  ONLINE  -
mypool    984M  43.7M   940M     4%  1.00x  ONLINE  -
# zfs snapshot mypool@backup1
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
mypool@backup1             0      -  43.6M  -
# zfs send mypool@backup1
Error: Stream can not be written to a terminal.
You must redirect standard output.
# zfs send mypool@backup1 > /backup/backup1
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
backup    960M  63.7M   896M     6%  1.00x  ONLINE  -
mypool    984M  43.7M   940M     4%  1.00x  ONLINE  -
```

ZFS Incremental Backups

Changes between two snapshots can be serialized and transferred using `zfs send` as well. Only the changes between the two snapshots are transferred (delta replication). This forms an incremental backup solution.

```
# zfs snapshot mypool@backup2
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
mypool@backup1         5.72M      -  43.6M  -
mypool@backup2             0      -  44.1M  -
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
backup    960M  61.7M   898M     6%  1.00x  ONLINE  -
mypool    960M  50.2M   910M     5%  1.00x  ONLINE  -
# zfs send -i mypool@backup1 mypool@backup2 > /backup/diff
# zpool list
NAME      SIZE  ALLOC   FREE    CAP  DEDUP  HEALTH  ALTROOT
backup    960M  80.8M   879M     8%  1.00x  ONLINE  -
mypool    960M  50.2M   910M     5%  1.00x  ONLINE  -
# ls -lah /backup
total 82247
drwxr-xr-x   2 root  wheel     4B Dec  3 11:46 .
drwxr-xr-x  21 root  wheel    28B Dec  3 11:32 ..
-rw-r--r--   1 root  wheel    61M Dec  3 11:36 backup1
-rw-r--r--   1 root  wheel    18M Dec  3 11:46 diff
```

ZFS Incremental Backups - Restore Data

We received the backups, but they are still stored in the binary stream format. When we want to get to the data itself, we need to use `zfs send` in conjunction with `zfs receive` (or the abbreviated `zfs recv`).

```
# zfs send mypool@backup1 | zfs receive backup/backup1
# ls -lah /backup/
total 431
drwxr-xr-x      3 root  wheel      3B Dec  3 12:06 .
drwxr-xr-x     21 root  wheel     28B Dec  3 12:05 ..
drwxr-xr-x   4219 root  wheel    4.1k Dec  3 11:34 backup1
```

In `backup1` we will now find all the data from the snapshot `mypool@backup1`.

```
# zfs list
NAME                            USED  AVAIL  REFER  MOUNTPOINT
backup                         43.7M   884M    32K  /backup
backup/backup1                 43.5M   884M  43.5M  /backup/backup1
mypool                         50.0M   878M  44.1M  /mypool
```

ZFS Backups – Encrypted Backups over the Network

Since ZFS works with standard out, the known functionalities like redirections can be used on the output. The streams can be encrypted locally or in transit via SSH to another cold standby server over an insecure network like the Internet. To make this work, a number of conditions must be met:

- passwordless SSH access using public keys to the target host
- `root` (or another privileged user) must be able to log in via SSH
- the target system must allow `root` the execution of `zfs recv` (can be configured in SSH)

A cron(8) job can be scheduled to perform the backup in regular intervals.

ZFS Backups – Encrypted Backups over the Network

For our example, we assume a recursive backup of all home directories on `host1` should be performed and sent to another host called `host2`.

```
host1# zfs snapshot -r tank/home@monday
host1# zfs send -R tank/home@monday | ssh host2 zfs recv -dvu pool
```

The option `-R` takes care of recursively descending into all child datasets. This includes snapshots, clones, and property settings.
The option `-d` in `zfs recv` will remove the original pool name on the receiving side and use only the name of the snapshot. The `-u` option determines that the receiving dataset should not be mounted. More information is displayed when `-v` is used, for example the elapsed time during the receiving operation.

Further Information

[FBSDHBZFS] The FreeBSD Handbook - 21.2 The Z File System (ZFS)
`http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/zfs.html`

[ZFSLinux] ZFS on Linux
`http://zfsonlinux.org`

[ZFSILLUMOS] Illumos Wiki page about ZFS
`http://wiki.illumos.org/display/illumos/ZFS`

[CSIMUNICH] CSI:Munich - Demonstrating ZFS on USB sticks
`http://www.youtube.com/watch?v=1zw8V8g5eT0`

[ZFSMASTERY] Allan Jude, Michael W. Lucas
FreeBSD Mastery: ZFS & FreeBSD Mastery: Advanced ZFS
`http://zfsbook.com`