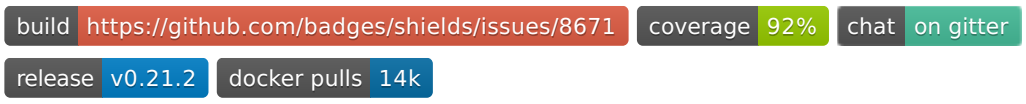
ZnapZend



ZnapZend is a ZFS centric backup tool to create snapshots and send them to backup locations. It relies on the ZFS tools snapshot, send and receive to do its work. It has the built-in ability to manage both local snapshots as well as remote copies by thinning them out as time progresses.

The ZnapZend configuration is stored as properties in the ZFS filesystem itself.

Note that while recursive configurations are well supported to set up backup and retention policies for a whole dataset subtree under the dataset to which you have applied explicit configuration, at this time pruning of such trees ("I want every dataset under var except var/tmp") is not supported. You probably do not want to enable ZnapZend against the root datasets of your pools due to that, but would have to be more fine-grained in your setup. This is consistent with (and due to) usage of recursive ZFS snapshots, where the command is targeted at one dataset and impacts it and all its children, allowing to get a consistent point-in-time set of snapshots across multiple datasets.

Compilation and Installation from source Inztructionz

If your distribution does not provide a packaged version of znapzend, or if you want to get a custom-made copy of znapzend, you will need a compiler and stuff to build some of the prerequisite perl modules into binary libraries for the target OS and architecture. For run-time you will need just perl.

The Git checkout includes a pregenerated `configure` script. For a rebuild of a checkout from scratch you may also want to `./bootstrap.sh` and then would need the autoconf/automake stack.

- On RedHat you get the necessaries with:

```
yum install perl-core
```

- On Ubuntu / Debian with:

```
apt-get install perl unzip
```

To also bootstrap on Ubuntu / Debian you may need:

```
apt-get install autoconf carton
```

- On Solaris 10 you may need the C compiler from Solaris Studio and gnu-make since the installed perl version is probably very old and you would likely have to build some dependency modules. The GNU make is needed instead of Sun make due to syntax differences. Notably you should reference it if you would boot-strap the code workspace from scratch:

```
MAKE=gmake ./bootstrap.sh
```

Note also that the perl version 5.8.4 provided with Solaris 10 is too old for the syntax and dependencies of znapzend. As one alternative, take a look at CSW packaging of perl-5.10.1 or newer and its modules, and other dependencies. To use a non-default perl, set the `PERL` environment variable to the path of your favorite perl interpreter prior to running `configure`, e.g.:

```
PERL=/opt/perl-32/bin/perl5.32.1 ./configure
```

- On OmniOS/SmartOS you will need perl and gnu-make packages.
- On macOS, if you have not already installed the Xcode command line tools, you can get them from the command line (Terminal app) with:

```
xcode-select --install  ### ...or just install the full Xcode app fro
```

With that in place you can now utter:

```
ZNAPVER=0.21.1
wget https://github.com/oetiker/znapzend/releases/download/v${ZNAPVER
tar zxvf znapzend-${ZNAPVER}.tar.gz
cd znapzend-${ZNAPVER}
### ./bootstrap.sh
./configure --prefix=/opt/znapzend-${ZNAPVER}
```

> NOTE: to get the current state of `master` branch without using git tools, you should fetch https://github.com/oetiker/znapzend/archive/master.zip

If the `configure` script finds anything noteworthy, it will tell you about it.

If any perl modules are found to be missing, they get installed locally into the znapzend installation. Your system perl installation will not be modified!

```
make
make install
```

Optionally (but recommended) put symbolic links to the installed binaries in the system PATH, e.g.:

```
ZNAPVER=0.21.1
for x in /opt/znapzend-${ZNAPVER}/bin/*; do ln -fs ../../../$x /usr/l
```

Verification Inztructionz

To make sure your resulting set of znapzend code and dependencies plays well together, you can run unit-tests with:

```
make check
```

or

```
./test.sh
```

> NOTE: the two methods run same testing scripts with different handling, so might behave differently. While that can happen in practice, that would be a bug to report and pursue fixing.

Packages

Debian control files, guide on using them and experimental debian packages can be found at https://github.com/Gregy/znapzend-debian

An RPM spec file can be found at https://github.com/asciiphil/znapzend-spec

For recent versions of Fedora and RHEL 7-9 there's also a copr repository by spike (sources at https://gitlab.com/copr_spike/znapzend):

```
dnf copr enable spike/znapzend
dnf install znapzend
```

For Gentoo there's an ebuild in the gerczei overlay.

For OpenIndiana there is an IPS package at http://pkg.openindiana.org/hipster/en/search.shtml?token=znapzend&action=Search made with the recipe at https://github.com/OpenIndiana/oi-userland/tree/oi/hipster/components/sysutils/znapzend

```
pkg install backup/znapzend
```

Configuration

Use the znapzendzetup program to define your backup settings. They will be stored directly in dataset properties, and will cover both local snapshot schedule and any number of destinations to send snapshots to (as well as potentially different retention policies on those destinations). You can enable recursive configuration, so the settings would apply to all datasets under the one you configured explicitly.

Example:

```
znapzendzetup create --recursive\
   --pre-snap-command="/bin/sh /usr/local/bin/lock_flush_db.sh" \
   --post-snap-command="/bin/sh /usr/local/bin/unlock_db.sh" \
   SRC '7d=>1h,30d=>4h,90d=>1d' tank/home \
   DST:a '7d=>1h,30d=>4h,90d=>1d,1y=>1w,10y=>1month'
root@bserv:backup/home
```

See the znapzendzetup manual for the full description of the configuration options.

For remote backup, znapzend uses ssh. Make sure to configure password-free login (authorized keys) for ssh to the backup target host with an account sufficiently privileged to manage its ZFS datasets under a chosen destination root.

For local or remote backup, znapzend can use mbuffer to level out the bursty nature of ZFS send and ZFS receive features, so it is quite beneficial even for local backups into another pool (e.g. on removable media or a NAS volume). It is also configured among the options set by znapzendzetup per dataset. Note that in order to use larger (multi-gigabyte) buffers you should point your configuration to a 64-bit binary of the mbuffer program. Sizing the buffer is a practical art, depending on the size and amount of your datasets and the I/O speeds of the storage and networking involved. As a rule of thumb, let it absorb at least a minute of I/O, so while one side of the ZFS dialog is deeply thinking, another can do its work.

Running

The znapzend daemon is responsible for doing the actual backups.

To see if your configuration is any good, run znapzend in noaction mode first.

```
znapzend --noaction --debug
```

If you don't want to wait for the scheduler to actually schedule work, you can also force immediate action by calling

```
znapzend --noaction --debug --runonce=<src_dataset>
```

then when you are happy with what you got, start it in daemon mode.

```
znapzend --daemonize
```

Best practice is to integrate znapzend into your system startup sequence, but you can also run it by hand. See the init/README.md for some inspiration.

Running by an unprivileged user

In order to allow a non-privileged user to use it, the following permissions are required on the ZFS filesystems (which you can assign with `zfs allow`):

Sending end: `destroy,hold,mount,send,snapshot,userprop`

Receiving end: `create,destroy,mount,receive,userprop`

Caveat Emptor: Receiver with some implementations of ZFS may have further constraints technologically. For example, non-root users with ZFS on Linux (as of 2022) may not write into a dataset with property `zoned=on` (including one inherited or just received -- and `zfs recv -x zoned` or similar options have no effect to not-replicate it), so this property has to be removed as soon as it appears on such destination host with the initial replication stream, e.g. leave a snippet like this running on receiving host before populating (`zfs send -R ...`) the destination for the first time:

```
while ! zfs inherit zoned backup/server1/rpool/rpool/zones/zone1/ROOT
```

You may also have to `zfs allow` by name all standard ZFS properties which your original datasets customize and you want applied to the copy (e.g. to eventually restore them), so the non-privileged user may `zfs set` them on that dataset and its descendants, e.g.:
`compression,mountpoint,canmount,setuid,atime,exec,dedup` or perhaps you optimized the original storage with the likes of:
`logbias,primarycache,secondarycache,sync` and note that other options may be problematic long-term if actually used by the receiving server, e.g.:
`refreservation,refquota,quota,reservation,encryption`

Running with restricted shell

As a further security twist on using a non-privileged user on the receiving host is to restrict its shell so just a few commands may be executed. After all, you leave its gates open with remote SSH access and a private key without a passphrase lying somewhere. Several popular shells offer a restricted option, for example BASH has a `-r` command line option and a `rbash` symlink support.

NOTE: Some SSH server versions also allow to constrain the commands which a certain key-based session may use, and/or limit from which IP addresses or DNS names such sessions may be initiated. See documentation on your server's supported `authorized_keys` file format and key words for that extra layer.

On original server, run `ssh-keygen` to generate an SSH key for the sending account (`root` or otherwise), possibly into an uniquely named file to use just for this connection. You can specify custom key file name, non-standard port, acceptable encryption algorithms and other options with SSH config:

```
# ~/.ssh/config
Host znapdest
        # "HostName" to access may even be "localhost" if the
backup storage
        # system can dial in to the systems it collects data from
(with SSH
        # port forwarding back to itself) -- e.g. running without
a dedicated
        # public IP address (consumer home network, corporate
firewall).
        #HostName localhost
        HostName znapdest.domain.org
        Port 22123
        # May list several SSH keys to try:
        IdentityFile /root/.ssh/id_ecdsa-znapdest
        IdentityFile /root/.ssh/id_rsa-znapdest
        User znapzend-server1
        IdentitiesOnly yes
```

On receiving server (example for Proxmox/Debian with ZFS on Linux):

- Create receiving user with `rbash` as the shell, and a home directory:

```
useradd -m -s `which rbash` znapzend-server1
```

- Restricted shell denies access to run programs and redirect to path names with a path separator (slash character, including `>/dev/null` quiescing). This allows to only run allowed shell commands and whatever is resolved by `PATH` (read-only after the profile file is interpreted). Typically a `bin` directory is crafted with programs you allow to run, but unlike the `chroot` jails you don't have to fiddle with dynamic libraries, etc. to make the login usable for its purpose.
    - Prepare restricted shell profile (made and owned by `root`) in the user home directory:

      ```
      # ~znapzend-server1/.rbash_profile
      # Restricted BASH settings
      # https://www.howtogeek.com/718074/how-to-use-restricted-shel
      PATH="$HOME/bin"
      export PATH
      ```

    - Neuter all other shell profiles so only the restricted one is consulted for any way the user logs in (avoid confusion):

      ```
      cd ~znapzend-server1/ && (
        rm -f .bash_history .bash_logout .bash_profile .bashrc .pro
        ln -s .rbash_profile .profile
        ln -s .rbash_profile .bashrc
        touch .hush_login )
      ```

    - (As `root`) Prepare `~/bin` for the user:

      ```
      mkdir -p ~znapzend-server1/bin
      cd ~znapzend-server1/bin
      for CMD in mbuffer zfs ; do ln -frs "`which "$CMD"`" ./ ; don
      # NOTE: If this user also receives other backups, you can
      # symlink commands needed for that e.g. "rsync" or "git"
      ```

    - Maybe go as far as to make the homedir not writeable to the user?
- Prepare SSH login:

```
mkdir -p ~znapzend-server1/.ssh
vi ~znapzend-server1/.ssh/authorized_keys
### Paste public keys from IdentityFile you used on the original serv
```

- Restrict access to SSH files (they are ignored otherwise):

```
chown -R znapzend-server1: ~znapzend-server1/.ssh
chmod 700 ~znapzend-server1/.ssh
chmod 600 ~znapzend-server1/.ssh/authorized_keys
```

- Unlock the user for ability to login (will use SSH key in practice, but unlocking in general may require a password to be set):

```
#usermod znapzend-server1 -p "`cat /dev/random | base64 | cut -b 0-20
usermod -U znapzend-server1
```

- Now is a good time to check that you can log in from the original backed-up system to the backup server (using the same account that znapzend daemon would use, to save the known SSH host keys), e.g. that keys and encryption algorithms are trusted, names are known, ports are open... If you defined a `Host znapdest` like above, just run:

```
# Interactive login?
:; ssh znapdest

# Gets PATH to run stuff?
:; ssh znapdest zfs list
```

- Dedicate a dataset (or several) you would use as destination for the znapzend daemon, and set ZFS permissions (see suggestions above), e.g.:

```
zfs create backup/server1
zfs allow -du znapzend-server1 create,destroy,mount,receive,userprop
```

NOTE: When defining a "backup plan" you would have to specify a basename for `mbuffer`, since the restricted shell would forbid running a fully specified pathname, e.g.:

```
znapzendzetup edit --mbuffer=mbuffer \
   SRC '6hours=>30minutes,1week=>6hours' rpool/export \
   DST '6hours=>30minutes,1week=>6hours,2weeks=>1day,4months=>1week,1
       znapdest:backup/server1/rpool/export
```

Running in Container

znapzend is also available as docker container image. It needs to be a privileged container depending on permissions.

```
docker run -d --name znapzend --device /dev/zfs --privileged \
    oetiker/znapzend:master
```

To configure znapzend, run in interactive mode:

```
docker exec -it znapzend /bin/sh
$ znapzendzetup create ...
# After exiting, restart znapzend container or send the HUP signal to
# reload config
```

By default, znapzend in container runs with `--logto /dev/stdout`. If you wish to add different arguments, overwrite them at the end of the command:

```
docker run --name znapzend --device /dev/zfs --privileged \
    oetiker/znapzend:master znapzend --logto /dev/stdout --runonce --
```

Be sure not to daemonize znapzend in the container, as that exits the container immediately.

Troubleshooting

By default a znapzend daemon would log its progress and any problems to local syslog as a daemon facility, so if the service misbehaves - that is the first place to look. Alternately, you can set up the service manifest to start the daemon with other logging configuration (e.g. to a file or to stderr) and perhaps with debug level enabled.

If your snapshots on the source dataset begin to pile up and not cleaned according to your expectations from the schedule you have defined, look into the logs particularly for summaries like `ERROR: suspending cleanup source dataset because X send task(s) failed` followed by each failed dataset name and a short verdict (e.g. `snapshot(s) exist on destination, but no common found on source and destination`). See above in the logs for more details, and/or disable the znapzend service temporarily (to avoid run-time conflicts) and run a manual replication:

```
znapzend --debug --runonce=<src_dataset>/failed/child --inherited
```

...to collect even more under-the-hood details about what is happening and to get ideas about fixing that. See the manual page about `--recursive` and `--inherited` modifiers to `--runonce` mode for more information.

Typical issues include:

- At least one destination is offline;
- At least one destination is full and can not be written into;
- A destination on SAN (iSCSI) or local device had transport issues and ZFS suspended all write operations until you fix and `zpool clear` it;
- Source is full (or exceeded quota) and can not be written into, so the new snapshots to send can not be made until you delete older ones;
- There are too many snapshots to clean up on source or destination, and the operation fails because the command line becomes too long. You can try running with `--features=oracleMode` to process each snapshot name as a separate command, that would be slower but more reliable in such situation;
- There are snapshots on destination, but none common with the source so incremental replication can not proceed without destroying much or all of the destination. Note this can be looking at snapshot names filtered by the pattern your backup schedule would create, and other znapzend options and/or a run of native `zfs send|zfs recv` would help if your destination has manually named snapshots that are common with your source.

> NOTE: Do not forget to re-enable the znapzend service after you have rectified the problem that prevented normal functionality.

One known problem relates to automated backups of datasets whose source can get cloned, renamed and promoted - typically boot environments (the rootfs of your OS installation and ZBE for local zones on illumos/Solaris systems behave this way to benefit from snapshots during upgrades and to allow easily switching back to older version if an update went bad). At this time (see issue #503) znapzend does not handle such datasets as branches of a larger ZFS tree and with `--autoCreation` mode in place just makes new complete datasets on the destination pool. On one hand this is wasteful for space (unless you use deduplication which comes with other costs), and on another the histories of snapshots seen in the same-named source and destination datasets can eventually no longer expose a "last-common snapshot" and this causes an error like `snapshot(s) exist on destination, but no common found on source and destination.`

In case you tinkered with ZFS attributes that store ZnapZend retention policies, or potentially if you have a severe version mismatch of ZnapZend (e.g. update from a PoC or very old version), `znapzendzetup list` is quite useful to non-intrusively discover whatever your current version can consider to be discrepancies in your active configuration.

Finally note that yet-unreleased code from the master branch may include fixes to problems you face (see recent commits and closed pull requests), but also may introduce new bugs.