

cs-sync

Manual

Realtime bidirectional and one-way folder sync with ACL propagation
and ZFS replication - for ZFS hosts

Version 3.3.0 · October 14, 2026

Contents

1. Overview	3
2. The three modes	3
2.1 sync - bidirectional	3
2.2 push - one-way, you are the source	4
2.3 pull - one-way, you are the destination	4
3. serve - the receiving end of a push	4
4. Endpoints and authentication	4
5. ZFS replication (--mode zfs, zfsreceive)	4
5.1 Proxmox: VM/CT configuration and daily freeze snapshot	5
5.2 Disaster recovery of a VM (step by step)	6
6. Installation	7
6.1 With napp-it CS (csweb-gui)	7
6.2 Standalone	8
6.3 Build from source	8
7. Tests	8
8. Known limitations	8
9. License and warranty	8

1. Overview

cs-sync is a realtime bidirectional and unidirectional folder sync service with folder ACL sync for ZFS hosts. It is part of napp-it 4ai (client-server edition, csweb-gui) and can also be used completely standalone.

Always encrypted. Every replication or sync between cs-sync hosts - over LAN, WAN or the internet - is encrypted end to end (ChaCha20-Poly1305) with a one-time shared transfer key (--key) that you set once per sync pair. Nothing is ever sent in clear text; a peer with the wrong key is rejected in the handshake. No VPN or TLS tunnel is required.

Every relationship connects two of three endpoint kinds through one unified syntax, used identically for every role:

```
local path      D:\data, /tank/data, ./data, ../data
cs-sync host    host[:port]          (default port 9010)
RustFS bucket   host[:port]/bucket   (default port 9000)
```

One string tells cs-sync whether to watch a folder, dial out to another cs-sync process, or drive an S3 bucket via rclone. Version 3.0 was a breaking rewrite: there is no upgrade path from 2.x, recreate existing services instead.

2. The three modes

Always left to right: Primary, Secondary, Backup.

Mode	Primary	Secondary	Backup
sync	local path (required)	local path or a local RustFS bucket (bidirectional)	optional, any endpoint (extra one-way leg)
push	local path (required, the source)	-	required, any endpoint (the target)
pull	required, any endpoint (the source)	-	required, local path (the destination)

```
sync: Primary(local) <====> Secondary(local) [====> Backup(any)]
push: Primary(local) =====> Backup(any)
pull: Primary(any) =====> Backup(local)
```

2.1 sync - bidirectional

```
cs-sync run --mode sync --primary /tank/share --secondary /tank/mirror
```

Realtime bidirectional sync with folder ACL propagation (NFSv4 ACLs on illumos/FreeBSD, POSIX ACLs on Linux). Both sides must be local paths on the same host, because bidirectional sync needs realtime change detection on both sides.

Secondary can also be a local RustFS/S3 bucket - the essential SMB to S3 case:

```
cs-sync run --mode sync \
  --primary /tank/smbshare \
  --secondary localhost:9000/s3share \
  --key s3admin:yourSecretKey
```

Only localhost or 127.0.0.1 is allowed. cs-sync drives rclone against the bucket (bisync --resync for the initial baseline, then per-object copy/delete) and never writes into the bucket's storage path directly.

An optional unidirectional Backup leg adds a third copy, pushed one-way from Primary to any endpoint kind:

```
cs-sync run --mode sync --primary /tank/share --secondary /tank/mirror \
  --backup 192.168.1.50:9010 --key mySharedKey
```

2.2 push - one-way, you are the source

```
cs-sync run --mode push --primary /tank/share --backup 192.168.1.50:9010 --key mySharedKey
cs-sync run --mode push --primary /tank/share --backup 192.168.1.50/bucket --key access:secret
cs-sync run --mode push --primary /tank/share --backup /mnt/otherdisk
```

Backup can be a local path (a plain one-way mirror), another cs-sync host (needs a matching serve listener) or a RustFS/S3 bucket, local or remote. Only its address and key are needed.

2.3 pull - one-way, you are the destination

```
cs-sync run --mode pull --primary 192.168.1.10/bucket --backup /tank/restore --key access:secret
cs-sync run --mode pull --primary /tank/source --backup /tank/mirror
```

Primary is the source (a RustFS bucket, local or remote, or a second local path); Backup is always local. Pulling directly from a live cs-sync host is not supported (see Known limitations).

3. serve - the receiving end of a push

Anything on the other side of a push needs its own listener:

```
cs-sync serve --dest /tank/incoming --listen 0.0.0.0:9010 \
--key mySharedKey --allow_ip 192.168.1.20
```

--key and --allow_ip are both mandatory: an empty key or a missing single-IP allowlist would let any host on the network attempt a sync. serve only ever accepts pushed data.

4. Endpoints and authentication

--key / --primary_key / --secondary_key / --backup_key. One shared default (--key) with per-leg overrides, interpreted per endpoint kind:

- cs-sync endpoint: a pre-shared transfer and encryption key. The wire protocol is natively encrypted (ChaCha20-Poly1305); the key authenticates and encrypts, and must match serve's --key.
- RustFS endpoint: access:secret (split on the first colon). cs-sync creates and updates the named rclone remote itself, non-interactively - no manual rclone config step is ever needed.

--allow_ip restricts every incoming listener a process opens (serve's receiver and run's optional --webhook-listen) to a single source IP.

Why a remote RustFS bucket cannot be the Secondary of sync: bidirectional sync needs a realtime, symmetric way to detect changes on both sides. A local bucket gets that indirectly through the Primary's filesystem watch; a remote bucket has only webhooks or periodic rescans, which are slower and asymmetric. push and pull cover the one-way case a remote bucket fits.

5. ZFS replication (--mode zfs, zfsreceive)

The Primary side (`run --mode zfs`) is the sender/master and pushes the data; the Backup side (`zfsreceive`) is the receiver/slave: it only listens and applies what it receives, nothing is pulled. `zfspull` is the former name and still works as an alias.

Event-based realtime replication of a ZFS dataset, always Primary to Backup. Instead of copying files, cs-sync detects changes (written@snap) and sends incremental zfs send | zfs rcv streams, resumable with -s. Open or running VMs are no problem: the snapshot is crash-consistent. Optionally cs-freeze4snap is called for an application-consistent daily snapshot.

```
# local: Primary and Backup on the same host
cs-sync run --mode zfs --primary tank/data --backup tank2/data --nr 1 \
--keep hours:24,days:30 --versions y
```

```
# LAN: sender on the Primary, receiver on the Backup
cs-sync zfsreceive --backup tank2/data --listen :9055 --key <key> --nr 1
cs-sync run --mode zfs --primary tank/data --backup-host 192.168.2.189:9055 --key <key> --nr 1
```

Option	Meaning
--primary / --backup	dataset names (local: both; LAN: Primary on the sender, Backup on zfsreceive)
--backup-host	address of the zfsreceive receiver (LAN mode)
--nr	sync number, part of the snapshot names sync_<nr>_<fsguid>_<slot>
--interval	check interval in seconds
--keep hours:N,days:M	retention, applied by each side on its own
--versions y/n	keep hourly/daily versions (n: only the latest two snapshots)
--rollback ask/auto	if the Backup has newer snapshots than the common base: wait for approval in the frontend (ask) or roll back (auto)
--recursive	replicate child datasets and zvols (-R)
--freeze, --freeze-bin, --freeze-args	Proxmox source: first snapshot of the day automatically via cs-freeze4snap (see 5.1), --freeze hh:mm only for other sources (ESXi); --freeze-bin only if cs-freeze4snap is not next to cs-sync, in the tools folder or in PATH; --freeze-args = step chain/policy

How it works:

- Snapshot slots: new, n0, n1 (hold 2), kh<MMDDHH>, kd<MMDD>, kf<MMDD> (daily freeze snapshot).
- The common base is found by GUID. The Primary never queries Backup snapshots; it only receives the Backup's list in the handshake.
- Crash recovery derives its state from ZFS itself; interrupted transfers are resumed with send -t <token>.
- The wire protocol reuses the encrypted handshake of the file modes (new frames 15-20, ProtoVersion unchanged).

Requirement: OpenZFS (Linux, FreeBSD, illumos, Windows). On Windows the process must run elevated, because zfs send | zfs recv pipes cannot be relaunched.

5.1 Proxmox: VM/CT configuration and daily freeze snapshot

Proxmox has two particularities.

First, the VM configuration is not inside the dataset (as with ESXi) but in `/etc/pve`. To restore a VM from its realtime-replicated dataset the VM configuration is needed. When cs-sync finds at start that the source is a Proxmox host, it reads in every cycle the config file of each VM/container whose disk datasets (`vm-<id>-disk-N`, `subvol-<id>-disk-N`) are part of the sync (not the whole `/etc/pve`) and attaches it as ZFS user property `cs:pveconf` to the disk dataset and to every sync snapshot. The properties are transferred to the backup by the cs-sync protocol (not inside the `zfs send` stream); a pure config change is sent without a new `zfs send`.

Second, an ordinary snapshot under Proxmox is not consistent in every case but has to be seen like a power failure, harmless to delicate depending on the VM. Therefore, on a Proxmox source the first snapshot of every day automatically goes through `cs-freeze4snap`, which freezes the VMs of the source dataset before the snapshot (guest agent, container freeze; the step chain per VM can be set with `--freeze-args`). `cs-freeze4snap` is looked up in the same folder as cs-sync (standalone setup), else in the napp-it tools folder or in PATH; if it is missing the freeze is skipped (warning in the log) and the snapshot stays crash-consistent. The snapshot is marked `cs:consistent=app` (frozen) or `paused` (VM only paused, crash-consistent) and kept as `kf<MMDD>` on both sides for `--keep days`. On days without data changes no new snapshot is taken. So there is at least one reliably restorable state per day.

Technical details of the config:

A replicated zvol alone is not a restorable VM: hardware, network, boot order and the disk assignment live in `/etc/pve/qemu-server/<id>.conf` (containers: `/etc/pve/lxc/<id>.conf`). When `cs-sync` finds that the Primary is a Proxmox host (`/etc/pve` exists), the sender always reads the config of every VM/CT whose disk datasets (`vm-<id>-disk-N`, `subvol-<id>-disk-N`, `base-<id>-disk-N`) are part of the sync and stores it as ZFS user property `cs:pveconf` (gzip + base64, continued in `cs:pveconf.1`, `.2` ... because one property value is limited to about 8 KB):

- on the disk dataset itself: the **current** config, on both sides. If only the config changes (RAM, NIC, new disk), only this small property is sent, no `zfs send`.
- on every sync snapshot (`n0`, `n1`, `kh..`, `kd..`, `kf..`): the config **at the time of that snapshot**, so every disk state has its matching config.

The backup side can be any OS (OmniOS, FreeBSD, Linux); the properties are read with `zfs get` and by `cs-sync pveconf`. Only the config of the VM belonging to the disk goes with that disk, never the whole `/etc/pve` (storage.cfg, users, firewall and cluster files are host settings and not part of the replication). The id travels inside the value, so the backup dataset may have any name.

Command	Meaning
<code>cs-sync pveconf list --dataset <ds></code>	all VM/CT configs below the dataset (id, type, size) and the snapshots that carry one
<code>cs-sync pveconf show --dataset <disk-ds> [--snapshot n0, kf1013 or <name>]</code>	print the config (current or of a snapshot)
<code>cs-sync pveconf restore --dataset <disk-ds> [--snapshot ..] --to <dir> [--map old=new] [--newid N] [--force]</code>	write <code><id>.conf</code> into <code><dir></code> ; <code>--map nvme=local-zfs</code> renames the storage; <code>--newid 905</code> also renames the disks <code>vm-105-disk-N</code> to <code>vm-905-disk-N</code> ; without <code>--force</code> an existing file is never overwritten

`--recursive y` with a parent dataset handles all disks below it. `cs-sync` never writes into `/etc/pve` on its own; the restore always targets a directory you name.

5.2 Disaster recovery of a VM (step by step)

This procedure was run live: backup on OmniOS (no Proxmox), VM data sent back to a Proxmox node, config restored with a new id, VM started and the guest agent answered. Names in angle brackets are placeholders.

Starting point. The primary is gone or damaged. `<B>` is the backup dataset (for example `tank/dr/vm-105-disk-0`) with the snapshots `sync_<nr>_<guid>_n0 / n1 / kh<MMDDHH> / kd<MMDD> / kf<MMDD>`. `<T>` is the Proxmox node on which the VM should run again, `<S>` its ZFS storage (see `/etc/pve/storage.cfg`), `<P>` the pool behind `<S>`.

1. Freeze the backup. Stop the receiver service of this VM on the backup host (Services > Realtime Sync) so no new snapshot is rotated during recovery. On the backup dataset never run `zfs rollback`, `zfs destroy` of `sync_` snapshots or `zfs set readonly=off`; recovery always works on a copy.

2. Find the VM. `cs-sync pveconf list --dataset <B-or-parent>` shows `id=<id>`, type (`qemu` or `lxc`) and the snapshots with a config. No output ("no cs:pveconf found"): the source was not a Proxmox host (no `/etc/pve`); then the data can still be restored (step 4), but the config must be rebuilt by hand.

3. Choose the snapshot. `zfs list -t snapshot -d 1 -o name,creation,cs:consistent <B>. kf<MMDD>` with `cs:consistent=app` is the application-consistent daily snapshot (guest agent freeze); `n0` is the newest state (crash-consistent, like after a power loss). Without a reason take `n0`; if the guest does not start cleanly, retry with `kf`.

4. Bring the data to <T> as a dataset named exactly like in the config (`vm-<id>-disk-N`, or the new id):

- backup pool is on `<T>` itself: `zfs send <B>@<snap> | zfs recv -u <P>/vm-<id>-disk-0`
- other host with ssh: `zfs send <B>@<snap> | ssh <T> zfs recv -u <P>/vm-<id>-disk-0`

- no ssh between the hosts: use cs-sync backwards. On <T>: `cs-sync zfsreceive --backup <P>/vm-<id>-disk-0 --listen :9070 --key <k> --allow_ip <backup-host> --nr 9`; on the backup host: `cs-sync run --mode zfs --primary <B> --backup-host <T>:9070 --key <k> --nr 9`. When the log shows `ok`, stop both.

Repeat for every disk of the VM (the config lists them: `scsi0`, `virtio0`, `efidisk0`, `tpmstate0`, `rootfs`). Then on <T>: `zfs set readonly=off <P>/vm-<id>-disk-N` and, after the cs-sync way, `zfs inherit cs:sync <P>/vm-<id>-disk-N`.

5. Restore the config on the host that holds the properties (the backup host): `cs-sync pveconf restore --dataset <B> --snapshot <slot> --to /root/restore --map <old-storage>=<S> [--newid <N>]` (containers: the same, the file goes to `/etc/pve/lxc`). Use `--newid` when the old id still exists on <T> or in the cluster. Copy the file to <T>: `/etc/pve/qemu-server/<id>.conf`. If `<id>.conf` already exists the command refuses (no `--force` unless the existing VM is really meant to be replaced).

6. Adapt the config. `ide2: ...:vm-<id>-cloudinit` is not replicated (separate dataset): remove it with `qm set <id> --delete ide2` or recreate it with `qm set <id> --ide2 <S>:cloudinit`. If the original VM may still run or the id was changed: new MAC (`qm set <id> --net0 virtio=<new-mac>,bridge=vmbr0`) and first `link_down=1` to avoid duplicate IPs. Disks that lived on other storages than the synced dataset are not in the backup and must be recovered separately.

7. Check before starting. `qm config <id>` must print without error; `pvesm path <S>:vm-<id>-disk-0` must give a device; `qm status <id>` says `stopped`.

8. Start and verify. `qm start <id>`, then `qm agent <id> ping` (answers when the guest agent runs) and `qm guest exec <id> -- uptime`. Check application data. A crash-consistent snapshot triggers `fsck` / journal replay on the first boot.

9. Back to normal operation. Remove `link_down`, put the VM under a new sync (new primary -> new backup, new `--nr`, initial full send). Delete the old backup dataset only after the new sync has completed.

Containers: steps 1 to 5 as above (`--to /etc/pve/lxc`, dataset `subvol-<id>-disk-0`), verification with `pct config <id>` and `pct start <id>`; the subvol datasets must have the same `mountpoint/canmount` as a healthy subvol on <T> (compare with `zfs get canmount,mountpoint`). The container path was tested for the config (properties and restore) but not for a complete restart.

Rules for automation (AI or script): never delete or roll back anything on the backup; every step has a check (output of `list`, `zfs list`, `qm config`); stop on the first error and report it; never use `--force` for an id that is in use; never start a restored copy with the network up while the original may still run.

6. Installation

Prebuilt binaries are attached to each GitHub release as `cs-sync-<os>.<arch>.tar.gz`: `mswin.amd64`, `linux.amd64`, `linux.arm64`, `illumos.amd64`, `solaris.amd64`, `freebsd.amd64`, `darwin.amd64`, `darwin.arm64`.

6.1 With napp-it CS (csweb-gui)

csweb-gui deploys and updates the binary per member under About > Download cs-tools. The Realtime Sync menu (03_System > 02_Services > 25_Realtime_Sync) expects it at a fixed path per platform:

```
data/cs_server/tools/cs-sync/<osdir>.<arch>/cs-sync[.exe]
```

<osdir> is `mswin`, `linux`, `illumos`, `solaris`, `freebsd` or `darwin`. On the Windows frontend host that is `C:\opt\csweb-gui\data\cs_server\tools\cs-sync\mswin.amd64\cs-sync.exe`; services created afterwards pick it up without a restart.

6.2 Standalone

Extract the archive for your platform, put cs-sync wherever you like and run it as shown above. rclone must be installed and on PATH for any RustFS endpoint; cs-sync shells out to it and does not vendor an S3 client.

6.3 Build from source

Requires Go 1.25 or newer.

```
git clone https://github.com/guenther-alka/cs-sync.git
cd cs-sync
go build -o cs-sync .
go build -o cs-sync.exe -ldflags "-X main.version=dev" .
GOOS=illumos GOARCH=amd64 go build -o cs-sync .
```

On PowerShell set \$env:GOOS and \$env:GOARCH as separate assignments before the build line.

7. Tests

```
go build ./...
go vet ./...
go test ./...
```

Unit tests cover the endpoint parser (including Windows drive paths versus host:port), the RustFS package (webhook events, --allow_ip filtering, xl.meta parsing) and the remote receiver (wire paths that would escape --dest are refused). There is no bundled integration suite against a real RustFS or a second cs-sync host; those were exercised manually.

8. Known limitations

- Pulling from a live cs-sync host is not supported: the wire protocol has no "send me your tree" frame. run --mode pull refuses a cs-sync Primary with a clear error. Use a serve listener and a push instead (the napp-it CS Realtime Sync menu does this automatically).
- The Backup leg of sync cannot be a local path; run a second independent push relationship.
- No ACL propagation to or from a RustFS endpoint; S3 has no filesystem ACLs.
- No recursive delete of non-empty folders on a RustFS side (documented no-op), because folders are only a naming convention over object keys.
- RustFS with a self-signed certificate: cs-sync always passes --no-check-certificate to rclone (not configurable), because rclone otherwise hangs on the TLS handshake. Invalid or expired certificates are therefore accepted too - cs-sync targets trusted LANs, not hostile networks.

9. License and warranty

BSD 2-Clause. There is no warranty. This is infrastructure software that deletes and overwrites files during normal operation. Read the delete-budget guard flags (--max-delete-count, --max-delete-percent), test with non-production data first and keep independent backups.