YARsync

Release 0.2.1

Yaroslav Nikitenko

Mar 28, 2023

DOCUMENTATION:

CHAPTER

ONE

INTRODUCTION

Yet Another Rsync is a file synchronization and backup tool. It can be used to synchronize data between different hosts or locally (for example, to a backup drive). It provides a familiar `git` command interface while working with files.

YARsync is a Free Software project covered by the GNU General Public License version 3.

1.1 Installation

`yarsync` is packaged for Debian/Ubuntu.

For Arch Linux, install the `yarsync` package from AUR. Packages for other distributions are welcome.

For an installation from PyPI, run

```
pip3 install yarsync
```

Since there is no general way to install a manual page for a Python package, one has to do it manually. For example, run as a superuser:

```
wget https://github.com/ynikitenko/yarsync/raw/master/docs/yarsync.1
gzip yarsync.1
mv yarsync.1.gz /usr/share/man/man1/
mandb
```

Make sure that the manual path for your system is correct. The command `mandb` updates the index caches of manual pages.

One can also install the most recent program version from GitHub. It incorporates latest improvements, but at the same time is less stable (new features can be changed or removed).

```
git clone https://github.com/ynikitenko/yarsync.git
pip3 install -e yarsync
```

This installs the `yarsync` executable to *~/.local/bin*, and does not require modifications of `PYTHONPATH`. After that, one can pull the repository updates without reinstallation.

To **uninstall**, run

```
pip3 uninstall yarsync
```

and remove the cloned repository.

1.2 Design and features

`yarsync` can be used to manage hierarchies of unchanging files, such as music, books, articles, photographs, etc. Its final goal is to have the same state of files across different computers. It also allows to store backup copies of data and easily copy, update or recover that. `yarsync` is

distributed
: There is no central host or repository for `yarsync`. If different replicas diverge, the program assists the user to merge the repositories manually.

efficient
: The program is run only on user demand, and does not consume system resources constantly. Already transferred files will never be transmitted again. This allows the user to rename or move files or whole directories without any costs, driving constant improvements on the repository.

non-intrusive
: `yarsync` does nothing to user data. It has no complicated packing or unpacking. All user data and program configuration are stored as usual files in the file system. If one decides to stop using `yarsync`, they can simply remove the configuration directory at any time.

simple
: `yarsync` does not implement complicated file transfer algorithms, but uses an existing, widely accepted and tested tool for that. User configuration is stored in simple text files, and repository snapshots are usual directories, which can be modified, copied or browsed from a file manager. All standard command line tools can be used in the repository, to assist its recovery or to allow any non-standard operations (for the users who understand what they do). Read the `yarsync` documentation to understand its (simple) design.

safe
: `yarsync` does its best to preserve user data. It always allows one to see what will be done before any actual modifications (*–dry-run*). It is its advantage compared to continous synchronization tools, that may be dangerous if local repository gets corrupt (e.g. encrypted by a trojan). Removed files are stored in older commits (until the user explicitly removes those).

1.3 Commands

```
checkout
clone
commit
diff
init
log
pull
push
remote
show
status
```

See `yarsync --help` for full command descriptions and options.

1.4 Requirements and limitations

`yarsync` is a Python wrapper (available for `Python>=3.6`) around `rsync` and requires a file system with **hard links**. Since these are very common tools, this means that it can easily run on any UNIX-like system. Moreover, `yarsync` is not required to be installed on the remote host: it is sufficient for `rsync` to be installed there.

In particular, `rsync` can be found:

- installed on most GNU/Linux distributions,
- installed on Mac OS,
- can be installed on Windows.

`yarsync` runs successfully on Linux. Please report to us if you have problems (or success) running it on your system.

1.5 Safety

`yarsync` has been used by the author for several years without problems and is tested. However, any data synchronization may lead to data loss, and it is recommended to have several data copies and always do a *–dry-run* (*-n*) first before the actual transfer.

1.6 Documentation

For the complete documentation, read the installed or online manual.

For more in-depth topics or alternatives, see details.

On the repository github, release notes can be found. On github pages there is the manual for yarsync 0.1.

An article in Russian that deals more with `yarsync` internals was posted on Habr.

1.7 Thanks

A good number of people have contributed to the improvement of this software. I'd like to thank Nilson Silva for packaging `yarsync` for Debian, Mikhail Zelenyy from MIPT NPM for the explanation of Python entry points, Jason Ryan and Matthew T Hoare for the inspiration to create a package for Arch, Scimmia for a comprehensive review and suggestions for my PKGBUILD, Open Data Russia chat for discussions about backup safety, Habr users and editors, and, finally, to the creators and developers of `git` and `rsync`.

1.7.1 NAME

yarsync - a file synchronization and backup tool

1.7.2 SYNOPSIS

yarsync [**-h**] [**–config-dir** *DIR*] [**–root-dir** *DIR*] [**-q** | **-v**] *command* [*args*]

1.7.3 DESCRIPTION

Yet Another Rsync stores rsync configuration and synchronizes repositories with the interface similar to git. It is *efficient* (files in the repository can be removed and renamed freely without additional transfers), *distributed* (several replicas of the repository can diverge, and in that case a manual merge is supported), *safe* (it takes care to prevent data loss and corruption) and *simple* (see this manual).

1.7.4 QUICK START

To create a new repository, enter the directory with its files and type

```
yarsync init
```

This operation is safe and will not affect existing files (including configuration files in an existing repository). Alternatively, run **init** inside an empty directory and add files afterward. To complete the initialization, make a commit:

```
yarsync commit -m "Initial commit"
```

commit creates a snapshot of the working directory, which is all files in the repository except **yarsync** configuration and data. This snapshot is very small, because it uses hard links. To check how much your directory size has changed, run **du**(1).

Commit name is the number of seconds since the Epoch (integer Unix time). This allows commits to be ordered in time, even for hosts in different zones. Though this works on most Unix systems and Windows, the epoch is platform dependent.

After creating a commit, files can be renamed, deleted or added. To see what was changed since the last commit, use **status**. To see the history of existing commits, use **log**.

Hard links are excellent at tracking file moves or renames and storing accidentally removed files. Their downside is that if a file gets corrupt, this will apply to all of its copies in local commits. The 3-2-1 backup rule requires to have at least 3 copies of data, so let us add a remote repository “my_remote”:

```
yarsync remote add my_remote remote:/path/on/my/remote
```

For local copies we still call the repositories “remote”, but their paths would be local:

```
yarsync remote add my_drive /mnt/my_drive/my_repo
```

This command only updated our configuration, but did not make any changes at the remote path (which may not exist). To make a copy of our repository, run

```
yarsync clone new-replica-name host:/mnt/my_drive/my_repo
```

clone copies all repository data (except configuration files) to a new replica with the given name and adds the new repository to remotes.

To check that we set up the repositories correctly, make a dry run with ‘**-n**’:

```
yarsync push -n new-replica-name
```

If there were no errors and no file transfers, then we have a functioning remote. We can continue working locally, adding and removing files and making commits. When we want to synchronize repositories, we **push** the changes *to* or **pull** them *from* a remote (first with a **–dry-run**). This is the recommended workflow, and if we work on different repositories in sequence and always synchronize changes, our life will be easy. Sometimes, however, we may forget to synchronize two replicas and they will end up in a diverged state; we may actually change some files or find them corrupt. Solutions to these problems involve user decisions and are described in **pull** and **push** options.

1.7.5 OPTION SUMMARY

–help, -h	show help message and exit
–config-dir=DIR	path to the configuration directory
–root-dir=DIR	path to the root of the working directory
–quiet, -q	decrease verbosity
–verbose, -v	increase verbosity
–version, -V	print version

1.7.6 COMMAND SUMMARY

checkout	restore the working directory to a commit
clone	clone a repository
commit	commit the working directory
diff	print the difference between two commits
init	initialize a repository
log	print commit logs
pull	get data from a source
push	send data to a destination
remote	manage remote repositories
show	print log messages and actual changes for commit(s)
status	print updates since last commit

1.7.7 OPTIONS

–help, **-h** : Prints help message and exits. Default if no arguments are given. After a command name, prints help for that command.

–config-dir=DIR : Provides the path to the configuration directory if it is detached. Both **–config-dir** and **–root-dir** support tilde expansion for user's home directory. See SPECIAL REPOSITORIES for usage details.

–root-dir=DIR : Provides the path to the root of the working directory for a detached repository. Requires **–config-dir**. If not set explicitly, the default working directory is the current one.

–quiet, **-q** : Decreases verbosity. Does not affect error messages (redirect them if needed).

–verbose, **-v** : Increases verbosity. May print more rsync commands and output. Conflicts with **–quiet**.

–version, **-V** : Prints the **yarsync** version and exits. If **–help** is given, it takes precedence over **–version**.

1.7.8 COMMANDS

All commands support the **–help** option. Commands that can change a repository also support the **–dry-run** option.

–dry-run, **-n** : Prints what will be transferred during a real run, but does not make any changes.

–help, **-h** : Prints help for a command or a subcommand.

1.7.9 checkout

yarsync checkout [**-h**] [**-n**] *commit*

Restores the working directory to its state during *commit*. WARNING: this will overwrite the working directory. Make sure that all important data is committed. Make a dry run first with **-n**.

If not the most recent commit was checked out, the repository HEAD (in git terminology, see **git-checkout**(1)) becomes detached, which prevents such operations as **pull** or **push**. To advance the repository to its correct state, check out the last commit or make a new one.

commit : The commit name (as printed in **log** or during **commit**).

1.7.10 clone

yarsync clone [**-h**] *name path|parent-path*

One can clone from within an existing repository **to** *parent-path* or clone **from** a repository at *path*. In both cases a new directory with the repository is created, having the same name as the original repository folder. If that directory already exists, **clone** will fail (several safety checks are being made). The local repository (origin or clone) will add another one as a remote.

Note that only data (working directory, commits, logs and synchronization information, not configuration files) will be cloned. This command will refuse to clone **from** a repository with a filter (see SPECIAL REPOSITORIES).

parent-path is useful when we want to clone several repositories into one directory. It allows us to use the same command for each of them (manually or with **mr**(1)). If one needs to have a different directory name for a repository, they can rename it manually (we don't require, but strongly encourage having same directory names for all replicas).

Positional arguments

name : Name of the new repository.

path : Path to the source repository (local or remote). Trailing slash is ignored.

parent-path : Path to the parent directory of the cloned repository (local or remote). Trailing slash is ignored.

1.7.11 commit

yarsync commit [**-h**] [**-m** *message*] [**–limit** *number*]

Commits the working directory (makes its snapshot). See QUICK START for more details on commits.

–limit=*number* : Maximum number of commits. If the current number of commits exceeds that, older ones are removed during **commit**. See SPECIAL REPOSITORIES for more details.

message : Commit message (used in logs). Can be empty.

1.7.12 diff

yarsync diff [**-h**] *commit* [*commit*]

Prints the difference between two commits (from old to the new one, the order of arguments is unimportant). If the second commit is omitted, compares *commit* to the most recent one. See **status** for the output format.

commit : Commit name.

1.7.13 init

yarsync init [**-h**] [*reponame*]

Initializes a **yarsync** repository in the current directory. Creates a configuration folder with repository files. Existing configuration and files in the working directory stay unchanged. Create a first commit for the repository to become fully operational.

reponame : Name of the repository. If not provided on the command line, it will be prompted.

1.7.14 log

yarsync log [**-h**] [**-n** *number*] [**-r**]

Prints commit logs (from newest to oldest), as well as synchronization information when it is available. To see changes in the working directory, use **status**.

Options

–max-count=*number*, **-n** : Maximum number of logs shown.

–reverse, **-r** : Reverse log order.

Example

To print information about the three most recent commits, use

```
yarsync log -n 3
```

1.7.15 pull

yarsync pull [**-h**] [**-f** | **–new** | **-b** | **–backup-dir** *DIR*] [**-n**] *source*

Gets data from a remote *source*. The difference between **pull** and **push** is mostly only the direction of transfer.

pull and **push** bring two repositories into the same state. They synchronize the working directory, that is they add to the destination new files from source, remove those missing on source and do all renames and moves of previously committed files efficiently. This is done in one run, and these changes apply also to logs, commits and synchronization. In most cases, we do not want our existing logs and commits to be removed though. By default, several checks are made to prevent data loss:

```
- local has no uncommitted changes,
- local has not a detached HEAD,
- local is not in a merging state,
- destination has no commits missing on source.
```

If any of these cases is in effect, no modifications will be made. Note that the remote may have uncommitted changes itself: always make a dry run with **-n** first!

To commit local changes to the repository, use **commit**. HEAD commit could be changed during **checkout** (see its section for the solutions). If the destination has commits missing on source, there are two options: to **–force** changes to the destination (removing these commits) or to merge changes inside the local repository with **pull –new**.

If we pull new commits from the remote, this will bring repository into a merging state. Merge will be done automatically if the last remote commit is among local ones (in that case only some older commits were transferred from there). If some recent remote commits are not present locally, however, this means that histories of the repositories diverged, and we will need to merge them manually. After we have all local and remote commits and the union of the working directories in our local repository, we can safely choose the easiest way for us to merge them. To see the changes, use **status** and **log**. For example, if we added a file in a *remote_commit* before and it was added now, we can just **commit** the changes. If we have made many local changes, renames and removals since then, we may better **checkout** our latest commit (remember that all files from the working directory are present in commits, so it is always safe) and link the new file to the working directory:

```
ln .ys/commits/<remote_commit>/path/to/file .
```

(it can be moved to its subdirectory without the risk of breaking hard links). If the remote commit was actually large, and local changes were recent but small, then we shall check out the remote commit and apply local changes by hand. After our working directory is in the desired state, we **commit** changes and the merge is finished. The result shall be pushed to the remote without problems.

pull options

–new : Do not remove local data that is missing on *source*. While this option can return deleted or moved files back to the working directory, it also adds remote logs and commits that were missing here (for example, old or unsynchronized commits). A forced **push** to the remote could remove these logs and commits, and this option allows one to first **pull** them to the local repository.

```
After **pull \--new** the local repository can enter a merging state.
```

See **pull** description for more details.

–backup, **-b** : Changed files in the working directory are renamed (appended with '~'). See **–backup-dir** for more details.

–backup-dir *DIR* : Changed local files are put into a directory *DIR* preserving their relative paths. *DIR* can be an absolute path or relative to the root of the repository. In contrast to **–backup**, **–backup-dir** does not change resulting file names.

```
This option is convenient for large file trees,
```

because it recreates the existing file structure of the repository (one doesn't have to search for new backup files in all subdirectories). For current rsync version, the command

```
    yarsync pull --backup-dir BACKUP <remote>

will copy updated files from the remote
```

and put them into the directory "BACKUP/BACKUP" (this is how rsync works). To reduce confusion, make standard **pull** first (so that during the backup there are only file updates).

```
This option is available only for **pull**,
```

because it is assumed that the user will apply local file changes after backup. For example, suppose that after a **pull –backup** one gets files *a* and *a~* in the working directory. One should first see, which version is correct. If it is the local file *a~*, then the backup can be removed:

```
    mv a~ a

By local we mean the one hard linked with local commits
```

(run *ls -i* to be sure). If the remote version is correct though, you need first to overwrite the local version not breaking the hard links. This can be done with an rsync option "–inplace":

```
    rsync --inplace a a~
    mv a~ a
    # check file contents and the links
    ls -i a .ys/commits/*/a

For a **\--backup-dir** and for longer paths these commands will be longer.
```

Finally, if you need several versions, just save one of the files under a different name in the repository.

```
After you have fixed all corrupt files, push them back to the remote.
```

pull and push options

–force, **-f** : Updates the working directory, removing commits and logs missing on source. This command brings two repositories to the nearest possible states: their working directories, commits and logs become the same. While working directories are always identical after **pull** or **push** (except for some of the **pull** options), **yarsync** generally refuses to remove existing commits or logs - unless this option is given. Use it if the destination has really unneeded commits or just remove them manually (see FILES for details on the commit directory). See also **pull –new** on how to fetch missing commits.

1.7.16 push

yarsync push [**-h**] [**-f**] [**-n**] *destination*

Sends data to a remote *destination*. See **pull** for more details and common options.

1.7.17 remote

yarsync remote [**-h**] [**-v**] [*command*]

Manages remote repositories configuration. By default, prints existing remotes. For more options, see *.ys/config.ini* in the FILES section.

-v : Verbose. Prints remote paths as well.

add

yarsync remote add [**-h**] *repository path*

Adds a new remote. *repository* is the name of the remote in local **yarsync** configuration (as it will be used later during **pull** or **push**). *path* has a standard form [user@]host:[path] for an actually remote host or it can be a local path. Since **yarsync** commands can be called from any subdirectory, local path should be absolute. Tilde for user's home directory '~' in paths is allowed.

rm

yarsync remote rm [**-h**] *repository*

Removes an existing *repository* from local configuration.

show

Prints remote repositories. Default.

1.7.18 show

yarsync show [**-h**] *commit* [*commit* ...]

Prints log messages and actual changes for commit(s). Changes are shown compared to the commit before *commit*. For the output format, see **status**. Information for several commits can be requested as well.

commit : Commit name.

1.7.19 status

yarsync status [**-h**]

Prints working directory updates since the last commit and the repository status. If there were no errors, this command always returns success (irrespective of uncommitted changes).

Output format of the updates

The output for the updates is a list of changes, including attribute changes, and is based on the format of *rsync –itemize-changes*. For example, a line

```
.d..t...... programming/
```

means that the modification time '*t*' of the directory '*d*' *programming/* in the root of the repository has changed (files were added or removed from that). All its other attributes are unchanged ('.').

The output is an 11-letter string of the format "YXcstpoguax", where 'Y' is the update type, 'X' is the file type, and the other letters represent attributes that are printed if they were changed. For a newly created file these would be '+', like

```
>f+++++++++ /path/to/file
```

The attribute letters are: **c**hecksum, **s**ize, modification **t**ime, **p**ermissions, **o**wner and **g**roup. **u** can be in fact **u**se (access) or creatio**n** time, or **b**oth. **a** stands for ACL, and **x** for extended attributes. Complete details on the output format can be found in the **rsync**(1) manual.

1.7.20 SPECIAL REPOSITORIES

A **detached** repository is one with the **yarsync** configuration directory outside the working directory. To use such repository, one must provide **yarsync** options **–config-dir** and **–root-dir** with every command (**alias**(1p) may be of help). To create a detached repository, use **init** with these options or move the existing configuration directory manually. For example, if one wants to have several versions of static Web pages, they may create a detached repository and publish the working directory without the Web server having access to the configuration. Alternatively, if one really wants to have both a continuous synchronization and **yarsync** backups, they can move its configuration outside, if that will work. Commits in such repositories can be created or checked out, but **pull** or **push** are currently not supported (one will have to synchronize them manually). A detached repository is similar to a bare repository in git, but usually has a working directory.

A repository with a **filter** can exclude (disable tracking) some files or directories from the working directory. This may be convenient, but makes synchronization less reliable, and such repository can not be used as a remote. See **rsync-filter** in the FILES section for more details.

A repository can have a **commit limit**. The maximum number of commits can be set during **commit**. **pull** and **push** do not check for missing commits on the destination when we are in a repository with commit limit. It makes a repository with commit limit more like a central repository. If we have reached the maximum number of commits, older ones are deleted during a new **commit**. Commit limit is stored in **.ys/COMMIT_LIMIT.txt**. It can be changed or removed at any time. Commit limit was introduced in `yarsync v0.2` and was designed to help against the problem of too many hard links (if it exists).

1.7.21 FILES

All **yarsync** repository configuration and data is stored in the hidden directory **.ys** under the root of the working directory. If the user no longer wants to use **yarsync** and the working directory is in the desired state, they can safely remove the **.ys** directory.

Apart from the working directory, only commits, logs and synchronization data are synchronized between the repositories. Each repository has its own configuration and name.

User configuration files

.ys/config.ini : Contains names and paths of remote repositories. This file can be edited directly or with **remote** commands according to user's preference.

```
**yarsync** supports synchronization
```

only with existing remotes. A simple configuration for a remote "my_remote" could be:

```
    [my_remote]
    path = remote:/path/on/my/remote

Several sections can be added for more remotes.
```

An example (non-effective) configuration is created during **init**. Note that comments in **config.ini** can be erased during **remote** {**add**,**rm**}.

```
Since removable media or remote hosts can change their paths
```

or IP addresses, one may use variable substitution in paths:

```
    [my_drive]
    path = $MY_DRIVE/my_repo

For the substitutions to take the effect,
```

export these variables before run:

```
    $ export MY_DRIVE=/run/media/my_drive
    $ yarsync push -n my_drive

If we made a mistake in the variable or path,
```

it will be shown in the printed command. Always use **–dry-run** first to ensure proper synchronization.

```
Another **yarsync** remote configuration option is **host**.
```

If both **path** and **host** are present, the effective path will be their concatenation "<host>:<path>". Empty **host** means local host and does not prepend the path.

```
It is possible to set default **host** for each section
```

from the section name. For that, add a default section with an option **host_from_section_name**:

```
    [DEFAULT]
    host_from_section_name

Empty lines and lines starting with \'**#**\' are ignored.
```

Section names are case-sensitive. White spaces in a section name will be considered parts of its name. Spaces around '**=**' are allowed. Full syntax specification can be found at https://docs.python.org/3/library/configparser.html.

.ys/repo_<name>.txt : Contains the repository name, which is used in logs and usually should coincide with the remote name (how local repository is called on remotes). The name can be set during **init** or edited manually.

```
Each repository replica must have a unique name.
```

For example, if one has repositories "programming/" and "music/" on a laptop "my_host", their names would probably be "my_host", and the names of their copies on an external drive could be "my_drive" (this is different from git, which uses only the author's name in logs).

```
Note that **clone** from inside a repository
```

for technical reasons creates a temporary file with the new repository name (which is also written in **CLONE_TO_<name>.txt**). If these files due to some errors remain on the system, they can be safely removed.

.ys/rsync-filter : Contains rsync filter rules, which effectively define what data belongs to the repository. The **rsync-filter** does not exist by default, but can be added for flexibility.

```
For example, the author has a repository \"~/work\",
```

but wants to keep his presentations in "tex/" in a separate repository. Instead of having a different directory "~/work_tex", he adds such rules to **rsync-filter**:

```
    # all are in git repositories
    - /repos
    # take care to sync separately
    - /tex

In this way, \"~/work/tex\" and contained git repositories will be excluded
```

from "~/work" synchronization. Lines starting with '**#**' are ignored, as well as empty lines. To complicate things, one could include a subdirectory of "tex" into "work" with an include filter '**+**'. For complete details, see FILTER RULES section of **rsync**(1).

```
While convenient for everyday use, filters make backup more difficult.
```

To synchronize a repository with them, one has to remember that it has subdirectories that need to be synchronized too. If the remote repository had its own filters, that would make synchronization even more unreliable. Therefore filters are generally discouraged: **pull** and **push** ignore remote filters (make sure you synchronize only *from* a repository with filters), while **clone** refuses to copy from a repository with **rsync-filter**.

yarsync technical directories

.ys/commits/ : Contains local commits (snapshots of the working directory). If some of the old commits are no longer needed (there are too many of them or they contain a large file), they can be removed. Make sure, however, that all remote repositories contain at least some of the present commits, otherwise future synchronization will get complicated. Alternatively, remove unneeded files or folders manually: commits can be edited, with care taken to synchronize them correctly.

.ys/logs/ : Contains text logs produced during **commit**. They are not necessary, so removing any of them will not break the repository. If one wants to fix or improve a commit message though, they may edit the corresponding log (the change will be propagated during **push**). It is recommended to store logs even for old deleted commits, which may be present on formerly used devices.

.ys/sync/ : Contains synchronization information for all known reposotories. This information is transferred between replicas during `pull`, `push` and `clone`, and it allows `yarsync` repositories to better support the 3-2-1 backup rule. The information is contained in empty files with names of the format **commit_repo.txt**. Pulling (or cloning) from a repository does not affect its files and does not update its synchronization information. **push** (and corresponding **clone**) updates synchronization for both replicas. For each repository only the most recent commit is stored. **sync** directory

was introduced in `yarsync v0.2`. See the release notes on how to convert old repositories to the new format or do it manually, if necessary.

```
If a replica has been permanently removed, its synchronization data
```

must be removed manually and propagated with **–force**.

1.7.22 EXIT STATUS

0 : Success

1 : Invalid option

7 : Configuration error

8 : Command error

9 : System error

2-6,**10-14**,**20-25**,**30**,**35** : rsync error

If the command could be run successfully, a zero code is returned. Invalid option code is returned for mistakes in command line argument syntax. Configuration error can occur when we are outside an existing repository or a **yarsync** configuration file is missing. If the repository is correct, but the command is not allowed in its current state (for example, one can not push or pull when there are uncommitted changes or add a remote with an already present name), the command error is returned. It is also possible that a general system error, such as a keyboard interrupt, is raised in the Python interpreter. See **rsync**(1) for rsync errors.

1.7.23 DIAGNOSTICS

To check that your clocks (used for properly ordering commits) at different hosts are synchronized well enough, run

```
python -c 'import time; print(time.time())'
```

To make sure that the local repository supports hard links instead of creating file copies, test it with

```
du -sh .
du -sh .ys
```

(can be run during **pull** or **clone** if they take too long). The results must be almost the same. If not, you may not use **yarsync** on this file system, have large deleted files stored in old commits or you may have subdirectories excluded with a **filter** (see SPECIAL REPOSITORIES section).

To test that a particular file “a” was hard linked to its committed versions, run

```
ls -i a .ys/commits/*/a
```

If all is correct, their inodes must be the same.

Hard links can be broken in a cloned git repository (as it could happen with **yarsync** tests before), because git does not preserve them. To fix hard links for the whole repository, run **hardlink**(1) in its root.

1.7.24 SEE ALSO

rsync(1)

The yarsync page is https://github.com/ynikitenko/yarsync.

1.7.25 BUGS

Requires a filesystem with hard links, rsync version at least 3.1.0 (released 28 September 2013) and Python >= 3.6.

Always do a **–dry-run** before actual changes. Occasionally Python errors are raised instead of correct return codes. Please report any bugs or make feature requests to https://github.com/ynikitenko/yarsync/issues.

1.7.26 COPYRIGHT

1.7.27 Advanced

Usage tips

Since `yarsync` allows using a command interface similar to `git`, one can synchronize several repositories simultaneously using myrepos.

If new data was added to several repositories simultaneously, commit the changes on one of them and synchronize that with the another. `rsync` should link the working directory with commits properly. This may fail depending on how you actually copied files (they may have changed attributes). In this case, create new commits in both repositories and manually rename them to be the same. Try to synchronize to see that all is linked properly. For example, when we move photographs from an SD card, we want to have at least two copies of them. It would be more reliable to copy data from the original source to two repositories than to push that from one of them to another (possible errors on the intermediate filesystem increase the risk). Make sure that the two repositories were synchronized beforehand.

Development

Community contributions are very important for free software projects. The best thing for the project on the starting phase is to spread information and create packages for new operating systems.

`yarsync` was tested on ext4, NFSv4 and SimFS on Arch Linux and CentOS. Tests on other systems would be useful.

Hard links

The file system must support hard links if you plan to use *commits*. Multiple hard links are supported by POSIX-compliant and partially POSIX-compliant operating systems, such as Linux, Android, macOS, and also Windows NT4 and later Windows NT operating systems [Wikipedia].

Notable file systems to **support hard links** include [hard links and comparison of file systems from Wikipedia]:

- EncFS (an Encrypted Filesystem using FUSE). Note that it doesn't support hard links when External IV Chaining is enabled (this is enabled by default in paranoia mode, and disabled by default in standard mode).
- ext2-ext4. Standard on Linux. Ext4 has a limit of 65000 hard links on a file.

- HFS+. Standard on Mac OS.
- NTFS. The only Windows file system to support hard links. It has a limit of 1024 hard links on a file.
- SquashFS, a compressed read-only file system for Linux.

Hard links are **not supported** on:

- FAT, exFAT. These are used on many flash drives.
- Joliet ("CDFS"), ISO 9660. File systems on CDs.

The majority of modern file systems support hard links. A full list of file system capabilities can be found on Wikipedia.

One can copy data to file systems without hard links, but this will reduce the functionality of `yarsync`, and one should take care not to consume too much disk space if accidentally copying files instead of hard linking.

rsync limitations

- Millions of files will be synced very slowly.
- `rsync` freezes when encountering **too many hard links**. Users report problems for repositories of 200 G or 90 GB, with many hard links. For the author's repository with 30 thousand files (160 thousand with commits) and 3 Gb of data `rsync` works fine. If you have a large repository and want to copy it with all hard links, it is recommended to create a separate partition (e.g. LVM) and copy the filesystem as a whole. You can also remove some of older backups.
- `rsync` may create separate files instead of hard linking them. It can be fixed quickly using the hardlink executable.

Alternatives

Free software that uses rsync includes:

- Back In Time. See previous snapshots using a GUI.
- Grsync, graphical interface for rsync.
- LuckyBackup. It is written in C++ and is mostly used from a graphical shell.
- rsnapshot, a filesystem snapshot utility. `rsnapshot` makes it easy to make periodic snapshots of local machines, and remote machines over ssh. Files can be restored by the users who own them, without the root user getting involved.

Other syncronization / backup / archiving software:

- casync is a combination of the rsync algorithm and content-addressable storage. It is an efficient way to deliver and update directory trees and large images over the Internet in an HTTP and CDN friendly way. Other systems that use similar algorithms include bup.
- Duplicity backs directories by producing encrypted tar-format volumes and uploading them to a remote or local file server. `duplicity` uses `librsync` and is space efficient. It supports many cloud providers. In 2021 `duplicity` supports deleted files, full unix permissions, directories, and symbolic links, fifos, and device files, but not hard links. It can be run on Linux, MacOS and Windows (under Cygwin).
- Git-annex manages distributed copies of files using git. This is a very powerful tool written in Haskell. It allows for each file to track the number of backups that contain it and their names, and it allows to plan downloading of a file to the local storage. This is its author's use case: "I have a ton of drives. I have a lot of servers. I live in a cabin on dialup and often have 1 hour on broadband in a week to get everything I need". I tried to learn `git-annex`, it was uneasy , and finally I found that it doesn't preserve timestamps (because `git` doesn't) and permissions. If that suits you, there is also a list of specialized related software. `git-annex` allows to use many cloud services as special remotes, including all rclone remotes.

- Rclone focuses on cloud and other high latency storage. It supports more than 50 different providers. As of 2021, it doesn't preserve permissions and attributes.

Continuous synchronization software:

- gut-sync offers a real-time bi-directional folder synchronization.
- Syncthing. A very powerful and developed tool, works on Linux, MacOS, Windows and Android. Mostly uses a GUI (admin panel is managed through a Web interface), but also has a command line interface.
- Unison is a file-synchronization tool for OSX, Unix, and Windows. It allows two replicas of a collection of files and directories to be stored on different hosts (or different disks on the same host), modified separately, and then brought up to date by propagating the changes in each replica to the other (pretty much like other syncronization tools work).
- Dropbox, Google Drive, Yandex Disk and many other closed-source tools fall into this cathegory.

ArchWiki includes several useful scripts for rsync and a list of its graphical front-ends. It also has a list of cloud synchronization clients and a list of synchronization and backup programs. Wikipedia offers a comparison of file synchronization software and a comparison of backup software. Git-annex has a list of git-related tools.

CHAPTER

TWO

MANUAL

2.0.1 NAME

yarsync - a file synchronization and backup tool

2.0.2 SYNOPSIS

yarsync [**-h**] [**–config-dir** *DIR*] [**–root-dir** *DIR*] [**-q** | **-v**] *command* [*args*]

2.0.3 DESCRIPTION

Yet Another Rsync stores rsync configuration and synchronizes repositories with the interface similar to git. It is *efficient* (files in the repository can be removed and renamed freely without additional transfers), *distributed* (several replicas of the repository can diverge, and in that case a manual merge is supported), *safe* (it takes care to prevent data loss and corruption) and *simple* (see this manual).

2.0.4 QUICK START

To create a new repository, enter the directory with its files and type

```
yarsync init
```

This operation is safe and will not affect existing files (including configuration files in an existing repository). Alternatively, run **init** inside an empty directory and add files afterward. To complete the initialization, make a commit:

```
yarsync commit -m "Initial commit"
```

commit creates a snapshot of the working directory, which is all files in the repository except **yarsync** configuration and data. This snapshot is very small, because it uses hard links. To check how much your directory size has changed, run **du**(1).

Commit name is the number of seconds since the Epoch (integer Unix time). This allows commits to be ordered in time, even for hosts in different zones. Though this works on most Unix systems and Windows, the epoch is platform dependent.

After creating a commit, files can be renamed, deleted or added. To see what was changed since the last commit, use **status**. To see the history of existing commits, use **log**.

Hard links are excellent at tracking file moves or renames and storing accidentally removed files. Their downside is that if a file gets corrupt, this will apply to all of its copies in local commits. The 3-2-1 backup rule requires to have at least 3 copies of data, so let us add a remote repository “my_remote”:

```
yarsync remote add my_remote remote:/path/on/my/remote
```

For local copies we still call the repositories "remote", but their paths would be local:

```
yarsync remote add my_drive /mnt/my_drive/my_repo
```

This command only updated our configuration, but did not make any changes at the remote path (which may not exist). To make a copy of our repository, run

```
yarsync clone new-replica-name host:/mnt/my_drive/my_repo
```

clone copies all repository data (except configuration files) to a new replica with the given name and adds the new repository to remotes.

To check that we set up the repositories correctly, make a dry run with '**-n**':

```
yarsync push -n new-replica-name
```

If there were no errors and no file transfers, then we have a functioning remote. We can continue working locally, adding and removing files and making commits. When we want to synchronize repositories, we **push** the changes *to* or **pull** them *from* a remote (first with a **–dry-run**). This is the recommended workflow, and if we work on different repositories in sequence and always synchronize changes, our life will be easy. Sometimes, however, we may forget to synchronize two replicas and they will end up in a diverged state; we may actually change some files or find them corrupt. Solutions to these problems involve user decisions and are described in **pull** and **push** options.

2.0.5 OPTION SUMMARY

–help, -h	show help message and exit
–config-dir=DIR	path to the configuration directory
–root-dir=DIR	path to the root of the working directory
–quiet, -q	decrease verbosity
–verbose, -v	increase verbosity
–version, -V	print version

2.0.6 COMMAND SUMMARY

checkout	restore the working directory to a commit
clone	clone a repository
commit	commit the working directory
diff	print the difference between two commits
init	initialize a repository
log	print commit logs
pull	get data from a source
push	send data to a destination
remote	manage remote repositories
show	print log messages and actual changes for commit(s)
status	print updates since last commit

2.0.7 OPTIONS

–help, **-h** : Prints help message and exits. Default if no arguments are given. After a command name, prints help for that command.

–config-dir=DIR : Provides the path to the configuration directory if it is detached. Both **–config-dir** and **–root-dir** support tilde expansion for user's home directory. See SPECIAL REPOSITORIES for usage details.

–root-dir=DIR : Provides the path to the root of the working directory for a detached repository. Requires **–config-dir**. If not set explicitly, the default working directory is the current one.

–quiet, **-q** : Decreases verbosity. Does not affect error messages (redirect them if needed).

–verbose, **-v** : Increases verbosity. May print more rsync commands and output. Conflicts with **–quiet**.

–version, **-V** : Prints the **yarsync** version and exits. If **–help** is given, it takes precedence over **–version**.

2.0.8 COMMANDS

All commands support the **–help** option. Commands that can change a repository also support the **–dry-run** option.

–dry-run, **-n** : Prints what will be transferred during a real run, but does not make any changes.

–help, **-h** : Prints help for a command or a subcommand.

2.0.9 checkout

yarsync checkout [**-h**] [**-n**] *commit*

Restores the working directory to its state during *commit*. WARNING: this will overwrite the working directory. Make sure that all important data is committed. Make a dry run first with **-n**.

If not the most recent commit was checked out, the repository HEAD (in git terminology, see **git-checkout**(1)) becomes detached, which prevents such operations as **pull** or **push**. To advance the repository to its correct state, check out the last commit or make a new one.

commit : The commit name (as printed in **log** or during **commit**).

2.0.10 clone

yarsync clone [**-h**] *name path|parent-path*

One can clone from within an existing repository **to** *parent-path* or clone **from** a repository at *path*. In both cases a new directory with the repository is created, having the same name as the original repository folder. If that directory already exists, **clone** will fail (several safety checks are being made). The local repository (origin or clone) will add another one as a remote.

Note that only data (working directory, commits, logs and synchronization information, not configuration files) will be cloned. This command will refuse to clone **from** a repository with a filter (see SPECIAL REPOSITORIES).

parent-path is useful when we want to clone several repositories into one directory. It allows us to use the same command for each of them (manually or with **mr**(1)). If one needs to have a different directory name for a repository, they can rename it manually (we don't require, but strongly encourage having same directory names for all replicas).

Positional arguments

name : Name of the new repository.

path : Path to the source repository (local or remote). Trailing slash is ignored.

parent-path : Path to the parent directory of the cloned repository (local or remote). Trailing slash is ignored.

2.0.11 commit

yarsync commit [**-h**] [**-m** *message*] [**–limit** *number*]

Commits the working directory (makes its snapshot). See QUICK START for more details on commits.

–limit=*number* : Maximum number of commits. If the current number of commits exceeds that, older ones are removed during **commit**. See SPECIAL REPOSITORIES for more details.

message : Commit message (used in logs). Can be empty.

2.0.12 diff

yarsync diff [**-h**] *commit* [*commit*]

Prints the difference between two commits (from old to the new one, the order of arguments is unimportant). If the second commit is omitted, compares *commit* to the most recent one. See **status** for the output format.

commit : Commit name.

2.0.13 init

yarsync init [**-h**] [*reponame*]

Initializes a **yarsync** repository in the current directory. Creates a configuration folder with repository files. Existing configuration and files in the working directory stay unchanged. Create a first commit for the repository to become fully operational.

reponame : Name of the repository. If not provided on the command line, it will be prompted.

2.0.14 log

yarsync log [**-h**] [**-n** *number*] [**-r**]

Prints commit logs (from newest to oldest), as well as synchronization information when it is available. To see changes in the working directory, use **status**.

Options

–max-count=*number*, **-n** : Maximum number of logs shown.

–reverse, **-r** : Reverse log order.

Example

To print information about the three most recent commits, use

```
yarsync log -n 3
```

2.0.15 pull

yarsync pull [**-h**] [**-f** | **–new** | **-b** | **–backup-dir** *DIR*] [**-n**] *source*

Gets data from a remote *source*. The difference between **pull** and **push** is mostly only the direction of transfer.

pull and **push** bring two repositories into the same state. They synchronize the working directory, that is they add to the destination new files from source, remove those missing on source and do all renames and moves of previously committed files efficiently. This is done in one run, and these changes apply also to logs, commits and synchronization. In most cases, we do not want our existing logs and commits to be removed though. By default, several checks are made to prevent data loss:

```
- local has no uncommitted changes,
- local has not a detached HEAD,
- local is not in a merging state,
- destination has no commits missing on source.
```

If any of these cases is in effect, no modifications will be made. Note that the remote may have uncommitted changes itself: always make a dry run with **-n** first!

To commit local changes to the repository, use **commit**. HEAD commit could be changed during **checkout** (see its section for the solutions). If the destination has commits missing on source, there are two options: to **–force** changes to the destination (removing these commits) or to merge changes inside the local repository with **pull –new**.

If we pull new commits from the remote, this will bring repository into a merging state. Merge will be done automatically if the last remote commit is among local ones (in that case only some older commits were transferred from there). If some recent remote commits are not present locally, however, this means that histories of the repositories diverged, and we will need to merge them manually. After we have all local and remote commits and the union of the working directories in our local repository, we can safely choose the easiest way for us to merge them. To see the changes, use **status** and **log**. For example, if we added a file in a *remote_commit* before and it was added now, we can just **commit** the changes. If we have made many local changes, renames and removals since then, we may better **checkout** our latest commit (remember that all files from the working directory are present in commits, so it is always safe) and link the new file to the working directory:

```
ln .ys/commits/<remote_commit>/path/to/file .
```

(it can be moved to its subdirectory without the risk of breaking hard links). If the remote commit was actually large, and local changes were recent but small, then we shall check out the remote commit and apply local changes by hand. After our working directory is in the desired state, we **commit** changes and the merge is finished. The result shall be pushed to the remote without problems.

pull options

–new : Do not remove local data that is missing on *source*. While this option can return deleted or moved files back to the working directory, it also adds remote logs and commits that were missing here (for example, old or unsynchronized commits). A forced **push** to the remote could remove these logs and commits, and this option allows one to first **pull** them to the local repository.

```
After **pull \--new** the local repository can enter a merging state.
```

See **pull** description for more details.

–backup, **-b** : Changed files in the working directory are renamed (appended with '~'). See **–backup-dir** for more details.

–backup-dir *DIR* : Changed local files are put into a directory *DIR* preserving their relative paths. *DIR* can be an absolute path or relative to the root of the repository. In contrast to **–backup**, **–backup-dir** does not change resulting file names.

```
This option is convenient for large file trees,
```

because it recreates the existing file structure of the repository (one doesn't have to search for new backup files in all subdirectories). For current rsync version, the command

```
    yarsync pull --backup-dir BACKUP <remote>

will copy updated files from the remote
```

and put them into the directory "BACKUP/BACKUP" (this is how rsync works). To reduce confusion, make standard **pull** first (so that during the backup there are only file updates).

```
This option is available only for **pull**,
```

because it is assumed that the user will apply local file changes after backup. For example, suppose that after a **pull –backup** one gets files *a* and *a~* in the working directory. One should first see, which version is correct. If it is the local file *a~*, then the backup can be removed:

```
    mv a~ a

By local we mean the one hard linked with local commits
```

(run *ls -i* to be sure). If the remote version is correct though, you need first to overwrite the local version not breaking the hard links. This can be done with an rsync option "–inplace":

```
    rsync --inplace a a~
    mv a~ a
    # check file contents and the links
    ls -i a .ys/commits/*/a

For a **\--backup-dir** and for longer paths these commands will be longer.
```

Finally, if you need several versions, just save one of the files under a different name in the repository.

```
After you have fixed all corrupt files, push them back to the remote.
```

pull and push options

–force, **-f** : Updates the working directory, removing commits and logs missing on source. This command brings two repositories to the nearest possible states: their working directories, commits and logs become the same. While working directories are always identical after **pull** or **push** (except for some of the **pull** options), **yarsync** generally refuses to remove existing commits or logs - unless this option is given. Use it if the destination has really unneeded commits or just remove them manually (see FILES for details on the commit directory). See also **pull –new** on how to fetch missing commits.

2.0.16 push

yarsync push [**-h**] [**-f**] [**-n**] *destination*

Sends data to a remote *destination*. See **pull** for more details and common options.

2.0.17 remote

yarsync remote [**-h**] [**-v**] [*command*]

Manages remote repositories configuration. By default, prints existing remotes. For more options, see *.ys/config.ini* in the FILES section.

-v : Verbose. Prints remote paths as well.

add

yarsync remote add [**-h**] *repository path*

Adds a new remote. *repository* is the name of the remote in local **yarsync** configuration (as it will be used later during **pull** or **push**). *path* has a standard form [user@]host:[path] for an actually remote host or it can be a local path. Since **yarsync** commands can be called from any subdirectory, local path should be absolute. Tilde for user's home directory '~' in paths is allowed.

rm

yarsync remote rm [**-h**] *repository*

Removes an existing *repository* from local configuration.

show

Prints remote repositories. Default.

2.0.18 show

yarsync show [**-h**] *commit* [*commit* ...]

Prints log messages and actual changes for commit(s). Changes are shown compared to the commit before *commit*. For the output format, see **status**. Information for several commits can be requested as well.

commit : Commit name.

2.0.19 status

yarsync status [**-h**]

Prints working directory updates since the last commit and the repository status. If there were no errors, this command always returns success (irrespective of uncommitted changes).

Output format of the updates

The output for the updates is a list of changes, including attribute changes, and is based on the format of *rsync –itemize-changes*. For example, a line

```
.d..t...... programming/
```

means that the modification time '*t*' of the directory '*d*' *programming/* in the root of the repository has changed (files were added or removed from that). All its other attributes are unchanged ('.').

The output is an 11-letter string of the format "YXcstpoguax", where 'Y' is the update type, 'X' is the file type, and the other letters represent attributes that are printed if they were changed. For a newly created file these would be '+', like

```
>f+++++++++ /path/to/file
```

The attribute letters are: **c**hecksum, **s**ize, modification **t**ime, **p**ermissions, **o**wner and **g**roup. **u** can be in fact **u**se (access) or creatio**n** time, or **b**oth. **a** stands for ACL, and **x** for extended attributes. Complete details on the output format can be found in the **rsync**(1) manual.

2.0.20 SPECIAL REPOSITORIES

A **detached** repository is one with the **yarsync** configuration directory outside the working directory. To use such repository, one must provide **yarsync** options **–config-dir** and **–root-dir** with every command (**alias**(1p) may be of help). To create a detached repository, use **init** with these options or move the existing configuration directory manually. For example, if one wants to have several versions of static Web pages, they may create a detached repository and publish the working directory without the Web server having access to the configuration. Alternatively, if one really wants to have both a continuous synchronization and **yarsync** backups, they can move its configuration outside, if that will work. Commits in such repositories can be created or checked out, but **pull** or **push** are currently not supported (one will have to synchronize them manually). A detached repository is similar to a bare repository in git, but usually has a working directory.

A repository with a **filter** can exclude (disable tracking) some files or directories from the working directory. This may be convenient, but makes synchronization less reliable, and such repository can not be used as a remote. See **rsync-filter** in the FILES section for more details.

A repository can have a **commit limit**. The maximum number of commits can be set during **commit**. **pull** and **push** do not check for missing commits on the destination when we are in a repository with commit limit. It makes a repository with commit limit more like a central repository. If we have reached the maximum number of commits, older ones are

deleted during a new **commit**. Commit limit is stored in **.ys/COMMIT_LIMIT.txt**. It can be changed or removed at any time. Commit limit was introduced in `yarsync v0.2` and was designed to help against the problem of too many hard links (if it exists).

2.0.21 FILES

All **yarsync** repository configuration and data is stored in the hidden directory **.ys** under the root of the working directory. If the user no longer wants to use **yarsync** and the working directory is in the desired state, they can safely remove the **.ys** directory.

Apart from the working directory, only commits, logs and synchronization data are synchronized between the repositories. Each repository has its own configuration and name.

User configuration files

.ys/config.ini : Contains names and paths of remote repositories. This file can be edited directly or with **remote** commands according to user's preference.

```
**yarsync** supports synchronization
```

only with existing remotes. A simple configuration for a remote "my_remote" could be:

```
    [my_remote]
    path = remote:/path/on/my/remote

Several sections can be added for more remotes.
```

An example (non-effective) configuration is created during **init**. Note that comments in **config.ini** can be erased during **remote** {**add**,**rm**}.

```
Since removable media or remote hosts can change their paths
```

or IP addresses, one may use variable substitution in paths:

```
    [my_drive]
    path = $MY_DRIVE/my_repo

For the substitutions to take the effect,
```

export these variables before run:

```
    $ export MY_DRIVE=/run/media/my_drive
    $ yarsync push -n my_drive

If we made a mistake in the variable or path,
```

it will be shown in the printed command. Always use **–dry-run** first to ensure proper synchronization.

```
Another **yarsync** remote configuration option is **host**.
```

If both **path** and **host** are present, the effective path will be their concatenation "<host>:<path>". Empty **host** means local host and does not prepend the path.

```
It is possible to set default **host** for each section
```

from the section name. For that, add a default section with an option **host_from_section_name**:

```
    [DEFAULT]
    host_from_section_name

Empty lines and lines starting with \'**#**\' are ignored.
```

Section names are case-sensitive. White spaces in a section name will be considered parts of its name. Spaces around '=' are allowed. Full syntax specification can be found at https://docs.python.org/3/library/configparser.html.

.ys/repo_<name>.txt : Contains the repository name, which is used in logs and usually should coincide with the remote name (how local repository is called on remotes). The name can be set during **init** or edited manually.

```
Each repository replica must have a unique name.
```

For example, if one has repositories "programming/" and "music/" on a laptop "my_host", their names would probably be "my_host", and the names of their copies on an external drive could be "my_drive" (this is different from git, which uses only the author's name in logs).

```
Note that **clone** from inside a repository
```

for technical reasons creates a temporary file with the new repository name (which is also written in **CLONE_TO_<name>.txt**). If these files due to some errors remain on the system, they can be safely removed.

.ys/rsync-filter : Contains rsync filter rules, which effectively define what data belongs to the repository. The **rsync-filter** does not exist by default, but can be added for flexibility.

```
For example, the author has a repository \"~/work\",
```

but wants to keep his presentations in "tex/" in a separate repository. Instead of having a different directory "~/work_tex", he adds such rules to **rsync-filter**:

```
    # all are in git repositories
    - /repos
    # take care to sync separately
    - /tex

In this way, \"~/work/tex\" and contained git repositories will be excluded
```

from "~/work" synchronization. Lines starting with '#' are ignored, as well as empty lines. To complicate things, one could include a subdirectory of "tex" into "work" with an include filter '+'. For complete details, see FILTER RULES section of **rsync**(1).

```
While convenient for everyday use, filters make backup more difficult.
```

To synchronize a repository with them, one has to remember that it has subdirectories that need to be synchronized too. If the remote repository had its own filters, that would make synchronization even more unreliable. Therefore filters are generally discouraged: **pull** and **push** ignore remote filters (make sure you synchronize only *from* a repository with filters), while **clone** refuses to copy from a repository with **rsync-filter**.

yarsync technical directories

.ys/commits/ : Contains local commits (snapshots of the working directory). If some of the old commits are no longer needed (there are too many of them or they contain a large file), they can be removed. Make sure, however, that all remote repositories contain at least some of the present commits, otherwise future synchronization will get complicated. Alternatively, remove unneeded files or folders manually: commits can be edited, with care taken to synchronize them correctly.

.ys/logs/ : Contains text logs produced during **commit**. They are not necessary, so removing any of them will not break the repository. If one wants to fix or improve a commit message though, they may edit the corresponding log (the change will be propagated during **push**). It is recommended to store logs even for old deleted commits, which may be present on formerly used devices.

.ys/sync/ : Contains synchronization information for all known reposotories. This information is transferred between replicas during `pull`, `push` and `clone`, and it allows `yarsync` repositories to better support the 3-2-1 backup rule. The information is contained in empty files with names of the format **commit_repo.txt**. Pulling (or cloning) from a repository does not affect its files and does not update its synchronization information. **push** (and corresponding **clone**) updates synchronization for both replicas. For each repository only the most recent commit is stored. **sync** directory was introduced in `yarsync v0.2`. See the release notes on how to convert old repositories to the new format or do it manually, if necessary.

```
If a replica has been permanently removed, its synchronization data
```

must be removed manually and propagated with **–force**.

2.0.22 EXIT STATUS

0 : Success

1 : Invalid option

7 : Configuration error

8 : Command error

9 : System error

2-6,**10-14**,**20-25**,**30**,**35** : rsync error

If the command could be run successfully, a zero code is returned. Invalid option code is returned for mistakes in command line argument syntax. Configuration error can occur when we are outside an existing repository or a **yarsync** configuration file is missing. If the repository is correct, but the command is not allowed in its current state (for example, one can not push or pull when there are uncommitted changes or add a remote with an already present name), the command error is returned. It is also possible that a general system error, such as a keyboard interrupt, is raised in the Python interpreter. See **rsync**(1) for rsync errors.

2.0.23 DIAGNOSTICS

To check that your clocks (used for properly ordering commits) at different hosts are synchronized well enough, run

```
python -c 'import time; print(time.time())'
```

To make sure that the local repository supports hard links instead of creating file copies, test it with

```
du -sh .
du -sh .ys
```

(can be run during **pull** or **clone** if they take too long). The results must be almost the same. If not, you may not use **yarsync** on this file system, have large deleted files stored in old commits or you may have subdirectories excluded with a **filter** (see SPECIAL REPOSITORIES section).

To test that a particular file "a" was hard linked to its committed versions, run

```
ls -i a .ys/commits/*/a
```

If all is correct, their inodes must be the same.

Hard links can be broken in a cloned git repository (as it could happen with **yarsync** tests before), because git does not preserve them. To fix hard links for the whole repository, run **hardlink**(1) in its root.

2.0.24 SEE ALSO

rsync(1)

The yarsync page is https://github.com/ynikitenko/yarsync.

2.0.25 BUGS

Requires a filesystem with hard links, rsync version at least 3.1.0 (released 28 September 2013) and Python >= 3.6.

Always do a **–dry-run** before actual changes. Occasionally Python errors are raised instead of correct return codes. Please report any bugs or make feature requests to https://github.com/ynikitenko/yarsync/issues.

2.0.26 COPYRIGHT

CHAPTER

THREE

ADVANCED

3.1 Usage tips

Since `yarsync` allows using a command interface similar to `git`, one can synchronize several repositories simultaneously using myrepos.

If new data was added to several repositories simultaneously, commit the changes on one of them and synchronize that with the another. `rsync` should link the working directory with commits properly. This may fail depending on how you actually copied files (they may have changed attributes). In this case, create new commits in both repositories and manually rename them to be the same. Try to synchronize to see that all is linked properly. For example, when we move photographs from an SD card, we want to have at least two copies of them. It would be more reliable to copy data from the original source to two repositories than to push that from one of them to another (possible errors on the intermediate filesystem increase the risk). Make sure that the two repositories were synchronized beforehand.

3.2 Development

Community contributions are very important for free software projects. The best thing for the project on the starting phase is to spread information and create packages for new operating systems.

`yarsync` was tested on ext4, NFSv4 and SimFS on Arch Linux and CentOS. Tests on other systems would be useful.

3.3 Hard links

The file system must support hard links if you plan to use *commits*. Multiple hard links are supported by POSIX-compliant and partially POSIX-compliant operating systems, such as Linux, Android, macOS, and also Windows NT4 and later Windows NT operating systems [Wikipedia].

Notable file systems to **support hard links** include [hard links and comparison of file systems from Wikipedia]:

- EncFS (an Encrypted Filesystem using FUSE). Note that it doesn't support hard links when External IV Chaining is enabled (this is enabled by default in paranoia mode, and disabled by default in standard mode).
- ext2-ext4. Standard on Linux. Ext4 has a limit of 65000 hard links on a file.
- HFS+. Standard on Mac OS.
- NTFS. The only Windows file system to support hard links. It has a limit of 1024 hard links on a file.
- SquashFS, a compressed read-only file system for Linux.

Hard links are **not supported** on:

- FAT, exFAT. These are used on many flash drives.

- Joliet ("CDFS"), ISO 9660. File systems on CDs.

The majority of modern file systems support hard links. A full list of file system capabilities can be found on Wikipedia.

One can copy data to file systems without hard links, but this will reduce the functionality of `yarsync`, and one should take care not to consume too much disk space if accidentally copying files instead of hard linking.

3.4 rsync limitations

- Millions of files will be synced very slowly.
- `rsync` freezes when encountering **too many hard links**. Users report problems for repositories of 200 G or 90 GB, with many hard links. For the author's repository with 30 thousand files (160 thousand with commits) and 3 Gb of data `rsync` works fine. If you have a large repository and want to copy it with all hard links, it is recommended to create a separate partition (e.g. LVM) and copy the filesystem as a whole. You can also remove some of older backups.
- `rsync` may create separate files instead of hard linking them. It can be fixed quickly using the hardlink executable.

3.5 Alternatives

Free software that uses rsync includes:

- Back In Time. See previous snapshots using a GUI.
- Grsync, graphical interface for rsync.
- LuckyBackup. It is written in C++ and is mostly used from a graphical shell.
- rsnapshot, a filesystem snapshot utility. `rsnapshot` makes it easy to make periodic snapshots of local machines, and remote machines over ssh. Files can be restored by the users who own them, without the root user getting involved.

Other syncronization / backup / archiving software:

- casync is a combination of the rsync algorithm and content-addressable storage. It is an efficient way to deliver and update directory trees and large images over the Internet in an HTTP and CDN friendly way. Other systems that use similar algorithms include bup.
- Duplicity backs directories by producing encrypted tar-format volumes and uploading them to a remote or local file server. `duplicity` uses `librsync` and is space efficient. It supports many cloud providers. In 2021 `duplicity` supports deleted files, full unix permissions, directories, and symbolic links, fifos, and device files, but not hard links. It can be run on Linux, MacOS and Windows (under Cygwin).
- Git-annex manages distributed copies of files using git. This is a very powerful tool written in Haskell. It allows for each file to track the number of backups that contain it and their names, and it allows to plan downloading of a file to the local storage. This is its author's use case: "I have a ton of drives. I have a lot of servers. I live in a cabin on dialup and often have 1 hour on broadband in a week to get everything I need". I tried to learn `git-annex`, it was uneasy , and finally I found that it doesn't preserve timestamps (because `git` doesn't) and permissions. If that suits you, there is also a list of specialized related software. `git-annex` allows to use many cloud services as special remotes, including all rclone remotes.
- Rclone focuses on cloud and other high latency storage. It supports more than 50 different providers. As of 2021, it doesn't preserve permissions and attributes.

Continuous synchronization software:

- gut-sync offers a real-time bi-directional folder synchronization.

- Syncthing. A very powerful and developed tool, works on Linux, MacOS, Windows and Android. Mostly uses a GUI (admin panel is managed through a Web interface), but also has a command line interface.
- Unison is a file-synchronization tool for OSX, Unix, and Windows. It allows two replicas of a collection of files and directories to be stored on different hosts (or different disks on the same host), modified separately, and then brought up to date by propagating the changes in each replica to the other (pretty much like other syncronization tools work).
- Dropbox, Google Drive, Yandex Disk and many other closed-source tools fall into this cathegory.

ArchWiki includes several useful scripts for rsync and a list of its graphical front-ends. It also has a list of cloud synchronization clients and a list of synchronization and backup programs. Wikipedia offers a comparison of file synchronization software and a comparison of backup software. Git-annex has a list of git-related tools.