Backup using rsync

Many people tend to ignore how important it is to have regular backups of their data until something bad happens and their stuff is gone for good. The good news is that on Linux there is an easy way to automatically create regular backups (it should work on other systems like Windows as well with some tweaking). One can even keep some of the backups for a very long time which might come in handy if you recognize something was lost like months ago. The obvious option is to use the rsync program as basis for a remote backup system.

The rsync tool

Something very smart with rsync it that one can point it to a previous, already existing backup on the server (cf. the –link-dest option in the script below) and rsync will compare any file of the new backup to the data there. If a file is existing in the old backup already rsync will not transfer the file again, but simply hard-link to it on the server and therefore also (almost) not consume any additional storage.

Another advantage of rsync comes into play with huge files (think of videos or veracrypt containers): rsync compares a file to a previous version on the server on a block level and only transfers and updates the parts of the file that did change.

Prepare the storage

First of all to make backups you need some kind of external storage. Theoretically, you could make backups on the same machine but if that one’s lost, stolen or broken all is gone (it might be useful though to go back in time for data which was deleted or overwritten by mistake). The next best option is to have some external storage stick or drive which is manually connect to your computer to run manual backups from time to time.

The best option for normal (i.e. non-business critical) usage seems to be a small storage server (NAS) on the local network. A good and rather cheap option is for example a Raspberry Pi running OpenMediaVault connected with an external SSD.

Enable secure access (ssh)

The very first step is to enable the ssh service as such on the storage server (e.g. OpenMediaVault). How this is done exactly depends on the server but usually this should be installed and enabled by default. Otherwise just search around or there should be some documentation on the internet.

For an automated, unattended backup to another server we need to enabled ssh access based on cryptographical keys without the need to an interactive password.

The first step is to create the required keys on the device

which will be backed up (your laptop for example):

```
ssh-keygen -t rsa
```

Answer all questions with return, i.e. keeping the defaults and don’t enter a password!

Next, we need to copy the public key which was just created from the device to the remote server where the backups will be stored (take your username and the name of your server) and enter your remote password when asked for it:

```
ssh-copy-id user@hostname
```

The private, secret key will always stay on your local machine; typically it will be stored in the directory .ssh.

Now give it a try to see if everything works out fine – just enter (again take your own user name and server name):

```
ssh user@hostname
```

You should be connected to your remote shell right away without any password requests. In case it shouldn’t work try the command with ssh -v or ssh -vv and check the output for an indication what might be wrong.

To make life a bit easier you might also want to configure

some of your ssh connections so they will be easier to use. To do so, just open the config file with:

```
nano ~/.ssh/config
```

and enter your details as needed like below:

```
Host mediaserver mediaserver.alpha
    Hostname mediaserver.alpha
    User admin

Host storage
    Hostname storage.alpha
    User tom
```

There are tons or options that can be defined in this ssh config file – start out with man ssh_config to check them out.

Rsync backup (push to server)

You need to decide where to store the backup script on your device (typically a laptop nowadays) and a good idea is to create a dedicated directory in your home directory. It’s also wise to make it hidden, i.e. to start the name with a dot so it won’t be visible by standard and it won’t be backed up by the script.

```
mkdir ~/.rsync
cd ~/.rsync
nano backup.sh
```

and paste in the following while changing everything specific for your local environment and setup (e.g. path names, user name, maximum bandwidth, etc.):

```
#!/bin/bash

LOGFILE="${0%.*}".log

# To ensure only one instance of the backup-script is running we create a lock-dir first.
# The lock is removed automatically on exit (including signals).
if ! mkdir "${0%.*}".lock; then
     echo "Lock detected (either rsync still running or manually locked); backup aborted..."
    exit 1
fi
trap 'rm --recursive --force --one-file-system "${0%.*}".lock' EXIT

# In case anything is failing during execution we want to catch it here and stop the script.
# Unfortunately, the following doesn't work for commands within a 'if' query... !
trap 'echo "Error encounted while executing $BASH_COMMAND. Exiting..." >> $LOGFILE; exit 1' ERR

echo "Starting new backup..." $(date) > $LOGFILE

SOURCE="/home/tom/"
SERVER="tom@storage.alpha"
BACKUP="/srv/dev-disk-by-label-SSD500GB/backup/laptop/tom/latest"
TARGET="$SERVER:$BACKUP"

if ssh $SERVER "test -e '$BACKUP'"; then
    echo "Latest full backup still exists (not archived yet). Exiting..." >> $LOGFILE
    exit 1
fi
```

```
RSYNCOPTIONS="--archive --numeric-ids --one-file-system --exclude-from=.rsync/backup.exclude --link-dest=../hourly.0 --compress --bwlimit=400K --partial-dir=.rsync-partial --human-readable --stats"

ionice -c2 -n7 nice -n 19 rsync $RSYNCOPTIONS "$SOURCE" "$TARGET.tmp" >> $LOGFILE

ssh $SERVER "mv '$BACKUP.tmp' '$BACKUP'"         # Put the backup in place, so it's marked as completed
ssh $SERVER "touch '$BACKUP'"                    # Timestamp the backup

echo "Backup script completed..." $(date) >> $LOGFILE
```

and make it executable:

```
chmod u+x backup.sh
```

Last step for the backup is to create a small file called backup.exclude which contains what will not be backed up. An example could be:

```
# Always exclude files and directories with the following endings
*.part
*.iso
*.log
*.bak
*.old
*.tmp
*.zip
*.temp
*.core
*.lock
*.crdownload
```

```
# As an exception we include the following hidden directory
+ .rsync/

# Now exclude all hidden files and directories from the backup
.*
```

With these exclude filters everything (files and directories) starting with a dot in the name and all files ending on .part or .iso are excluded from the backup.

First of all you should try the backup manually; maybe with a rather small scope of files to backup (so it doesn’t run for hours while testing). To start the script, simply type

```
.rsync/backup.sh
```

in your home directory and check the log-file in the same directory and the backups on the server. Note: if the script is run by cron then the it will be run from your home directory, therefore the relative path to the exclude-file is relative to the home directory. If run from another directory the relative path to the exclude file must be changed accordingly.

Now run a full backup which easily can take many hours. Then manually rename that backup on the server from latest to hourly.0 and run the backup script again. This time it should complete within a few minutes at most. Check the new backup on the server and will find the hard-linked files there.

If everything looks fine one probably wants to run the script automatically like for example each hour. To enable this

simply make in entry into crontab with:

```
crontab -e
```

and add the following line (or something similar) at the end of the file:

```
0 * * * * /home/tom/.rsync/backup.sh
```

In case you want to avoid the script to be running for some time simply create the lock manually (in the script directory):

```
mkdir backup.lock
```

You might want to check the log-file in the same directoy to see if everything is working as it should. And don’t forget to remove the lock in case you created it manually and you want to run the backups again:

```
rmdir backup.lock
```

If you want to lock the backup script quite frequently it might be a good idea to define alias commands for the locks.

Archive of backups

Usually, one also wants to keep a few of the backups for a longer time. This means that some of the older backups should be kept for the future. This is accomplished with archiving the backups which can be easily automated with the following shell script.

Connect to your storage (NAS) server and become root to save the script:

```
su
nano /root/backup-archive.sh
```

Now paste in the following – adjusting the path $BASE and a few other things maybe to your local setup and needs (don’t worry, the script looks much more complicated than it actually is):

```
#!/bin/bash

# To ensure only one instance of the backup-script is running we create a lock-dir first.
# The lock is removed automatically on exit (including signals).
if ! mkdir "${0%.*}".lock; then
     echo "Lock detected (either script still running or manually locked). Exiting..."
    exit 1
fi
trap 'rm --recursive --force "${0%.*}".lock' EXIT

# In case anything is failing during execution we want to
```

```
catch it here and stop the script.
# Unfortunately, the following doesn't work for commands
within a 'if' query... !
trap 'echo "Error encounted while executing $BASH_COMMAND.
Exiting..."; exit 1' ERR

BASE="/srv/dev-disk-by-label-SSD500GB/backup/laptop/tom"

N=100   # Maximum number of backups per category

case $1 in
    hourly)
        if [ ! -d "$BASE/latest" ]; then
            echo "No new backup available to be archived (no
folder 'latest'). Exiting..."
            exit
        fi
        # If the latest backup is identical to the previous
one in 'hourly.0' then skip
        # the backup rotation. Exit status of 'diff' is 0 if
inputs (directories) are
        # identical, 1 if they are different, 2 if there's any
kind of trouble.
            if diff --recursive --brief $BASE/latest
$BASE/hourly.0; then
            echo "Not rotating, since there are no changes in
'latest' since last backup."
            echo "Removing 'latest' so a new backup will be
made."
            rm --recursive --force "$BASE/latest"
            exit
        fi
        rm --recursive --force "$BASE/hourly.8"
        for I in {100..0}; do
                if [ -d "$BASE/hourly.$I" ]; then mv
"$BASE/hourly.$I" "$BASE/hourly.$((I+1))"; fi
        done
        mv "$BASE/latest" "$BASE/hourly.0"
        ;;
    daily)
        until [ -d "$BASE/hourly.$N" ]; do      # Keep at
```

```
least hourly.0 for the hard links
             if (( $N == 1 )); then echo "No hourly backup
available for daily backup. Exiting..."; exit; fi
            let N--
        done
        rm --recursive --force "$BASE/daily.5"
        for I in {100..0}; do
                 if [ -d "$BASE/daily.$I" ]; then mv
"$BASE/daily.$I" "$BASE/daily.$((I+1))"; fi
        done
        mv "$BASE/hourly.$N" "$BASE/daily.0"
        ;;
    weekly)
        until [ -d "$BASE/daily.$N" ]; do
             if (( $N == 0 )); then echo "No daily backup
available for weekly backup. Exiting..."; exit; fi
            let N--
        done
        rm --recursive --force "$BASE/weekly.4"
        for I in {100..0}; do
                 if [ -d "$BASE/weekly.$I" ]; then mv
"$BASE/weekly.$I" "$BASE/weekly.$((I+1))"; fi
        done
        mv "$BASE/daily.$N" "$BASE/weekly.0"
        ;;
    monthly)
        until [ -d "$BASE/weekly.$N" ]; do
             if (( $N == 0 )); then echo "No weekly backup
available for monthly backup. Exiting..."; exit; fi
            let N--
        done
        rm --recursive --force "$BASE/monthly.12"
        for I in {100..0}; do
                if [ -d "$BASE/monthly.$I" ]; then mv
"$BASE/monthly.$I" "$BASE/monthly.$((I+1))"; fi
        done
        mv "$BASE/weekly.$N" "$BASE/monthly.0"
        ;;
    yearly)
        until [ -d "$BASE/monthly.$N" ]; do
             if (( $N == 0 )); then echo "No monthly backup
```

```
available for yearly backup. Exiting..."; exit; fi
            let N--
        done
        for I in {100..0}; do
                if [ -d "$BASE/yearly.$I" ]; then mv
"$BASE/yearly.$I" "$BASE/yearly.$((I+1))"; fi
        done
        mv "$BASE/monthly.$N" "$BASE/yearly.0"
        ;;
    *)
        echo "Invalid (or no) option. Exiting..."
        ;;
esac
```

It must be invoked by giving an argument (either hourly, daily, weekly, monthly, or yearly) depending on what level of backups should be archived.

Once you confirmed it’s working by running it manually a few times the best practice is to invoke it automatically and regularly by setting up crontab. The different levels of rotations should be run at different daytimes, e.g. run the yearly one at 3:10 am (once a year), the monthly one at 3:20 am (once a month) and so on. The hourly rotation should be scheduled a few minutes before the new backup on the laptop runs – e.g. run the hourly rotation at 10 mins before the top of the hour if the backup script on the source device (laptop) runs at the full hour.

Retrieve files

Retrieving documents from the backup is really easy. Simply use a command like the following (maybe first switch to a

separate local directory first):

```
mkdir retrieve
cd retrieve
nice rsync --archive --numeric-ids --progress tom@storage.alpha:/srv/dev-disk-by-label-SSD500GB/backup/laptop/tom/hourly.4/Paroles/Renaud.pdf .
```

Obviously, the above rsync command needs to be adapted to the specific setup (user name, host name, path, file to retrieve, etc.) and one could also limit the bandwidth just like in the backup script.

Final thoughts

This backup setup and the two shell scripts are quite simplistic and in no way elaborated – but the whole thing just works. It’s also worth mentioning that there are applications around that effectively implement something pretty similar; one example being the ‚rsnapshot‘ tool. Personally, I prefer to do it myself though as this gives much more flexibility control, I can learn something – and it’s just plain fun to see it working.