dd-Transfer
Efficiently Migrate Large Amounts of Raw Data Using rysnc and dd

Content

Introduction

In the daily work of an administrator or a PC user, the task of transferring the contents of an entire hard disk poses a challenge from time to time. This can happen, for example, when creating a backup or transferring the content of a storage medium byte by byte into the cloud. Standard hard disks now have a capacity of ten terabytes and more. This data volume is not easy to transport over the available bandwidth of usual internet connections. It can be a very tedious task even for local networks. This article first briefly describes conventional ways and means of dealing with the problem. The requirements and the substeps for solving the problem that is to be provided by the program **ddtransfer.sh** are derived from the specification of the limitations of these methods. Finally, there is an outlook for the further development of the program.

Traditional options of data migration

Copying, transferring, and importing the data of an USB stick with a size of 16 GB, for instance, takes 67 minutes during normal working hours in Berlin using a alleged 100 MB connection, which is shared by more than 120 employees at the same time.

dd -> Copy -> dd

The following steps were required:

Creating the image

```
root@ddtransfer:~# time dd if=/dev/sdc bs=1G iflag=fullblock
of=sdc.img
```

Duration: 12 minutes, 25 seconds

Transferring the image

```
root@ddtransfer:~# time rsync --compress --compress-level=9
sdc.img root@158.222.102.233:/mnt/1/.
```

Duration: 51 minutes, 10 seconds

Importing the data into a storage volume

```
root@las-transfer2:/mnt/1# time dd if=sdc.img of=/dev/vdd bs=1G
iflag=fullblock
```

Duration: 3 minutes, 28 seconds

CloneZilla

I canceled a test with CloneZilla as a cross check, because I would have had to activate password authentication and this is not convenient for me. I do not want to undermine basic security measures as I have experienced quite often that such workarounds are often forgotten and end up with being hacked instances – especially when easy ad hoc passwords are used.

bbcp

With Bar Bar Copy (bbcp) the data transmission can be accelerated considerably on the one hand, but the data would have been transmitted unencrypted. On the other hand, the goal is to transfer very large files. However, if images are created beforehand, a large amount of disk space must be made available in advance (even if the files are compressed). The transfer script requires only 10 GB free space on the hard disk, although more space, e.g. 100 GB, is quite recommended.

Send via Mail

Another way which is offered to customers for the transmission of large amounts of data is the dispatch of hard disks or a server (starting from approx. 10 TB), which are then sent to the provider. For more then 10 TB of data this alternative may still be the more practicable one, but with the further technical development it is quite foreseeable that ddtransfer.sh is also a feasible way with these data volumes.

Acronis

Furthermore, I tested the creation of an online backup with Acronis, using a Strato online backup account. Pros: deduplication and compression of the backup. Cons: client must be installed separately, raw devices cannot be directly imported, but need a boot CD of the target machine and an account is required. Conclusion: The speed of this method is quite good but, as said, an account is necessary.

Requirements for a new program

With an image size of 15 GB, the procedure described above – dd to file, copying over the network, dd to a device – is completely normal and appropriate. The actual problems only appear in case of

1. a considerably larger amount of data
2. which is to be transferred over a small bandwidth

3. which additionally raises the question of data integrity

The duration of the transfer can easily take several days. If a failure occurs during this time, the transmission process must be restarted. It is obvious that the less data needs to be retransmitted, the less time is lost.

Once the transfer of a partial step has been completed, the creation of checksums at the very beginning and following the import of the files can determine whether a bit flip or other changes have occurred in the data stock. In previous procedures, this examination can only be carried out as an overall examination; if a change is detected, it is quite more annoying at the end of a very long process.

The low bandwidth of a normal internet access is the part that costs most of the processing time. However, very few programs are able to take full advantage of even that bandwidth. An effective way to make even more extensive use of bandwidth is to call parallel connections. The respective connections are by no means processed at the same speed. It is not uncommon that when 16 connections are started one after another with a few seconds interval, the tenth transmission is completed first.

The script ddtransfer.sh was written to close these gaps:

1. The data to be transferred is divided into blocks that can be copied with multiple dd-invocations in parallel.
2. A checksum is formed for each block during its creation.
3. Depending on the bandwidth and number of hops up to the target computer, many parallel connections are used for sending the blocks.
4. While further blocks are being transferred, the blocks that have already arrived on the target system are written to the target device, again several times in parallel if possible.
5. After writing each transfer to the target device, a new checksum is formed from the finished block.
6. At the end the checksums are compared. If there are differences, the affected block is transferred and calculated again.
7. By logging the individual steps precisely, a transfer that has been started can be resumed after an interruption on the block at which the interruption occurred.

As far as I know, no available tool for the transfer of large devices currently meets all these conditions. The program ddtransfer.sh is based on ‘dd’, which belongs to the so-called core utilities, so it should run on any Linux that offers a shell, also ‘rsync’ and the ‘ssh agent’ should be present. Thus it can be used with a common Live-Linux, for example to transfer Windows installations. It can also be used on small hardware such as the Raspberry Pi, which can serve as a transmission station.

The real sticking point of the whole procedure is the internet connection. Using the script in local networks does not necessarily make sense. Under the condition of a sufficiently large bandwidth, the time for the entire transmission could even be extended. However, if the amount of data is very high and the checksum calculation is important, the program should also be useful in the LAN. The difference according to the original intention of the program is brief:

1. Local: Fast data import in a few processes
2. Internet: Slow transfer in many processes
3. Remote: Fast writing of data to the respective Device with few processes

Accordingly, there are three main functions in the script:

1. **CreateImageFiles**: Create the parts of the image as individual files and create the input checksum.
2. **SendFiles**: Transfer files with Rsync with stronges compression enabled.
3. **ImageToTargetDevice**: Import of the files and subsequent creation of the output checksum from the imported data.

The **ShowProceeding** function is used for monitoring and final processing, i.e. the checksums are finally compared and individual parts are again created, transferred and imported if necessary.

The other functions

1. dd_command,
2. RemoteWorkspace,
3. Transfer,
4. RemoteStatusImageToTargetDevice and
5. ReImage

represent substeps and are invoked by the main functions.

Resume after interruption

When the target computer restarts, the script continues to run without interruption. If the source computer or the target computer fails for a longer period of time, the program can be executed again. As an option, the name of a report file must be given, which contains the necessary variable values of the previous call. Then the status of the data transfer and the blocks created is checked. Where necessary, interrupted sub processes are restarted and the entire process continues. The suffixes of the file names can also be used to determine how far processing has already progressed:

- ’run’ for still to process,
- ’transfer’ as ready for transmission,
- ’ongoing’ for being in transmission and
- ’ToDev’ for a file that is in the process of writing remotely.

The main means for the program continuation is the report file which is read and then updated. Formerly started transfers are initiated again. Continuing started Rsync calls is technically possible, but would have required rewriting some functions, which seemed to me to be too complicated under the given circumstances, because the development of the script was already too advanced.

Challenges

Disk space

There is always the possibility that the disk is filled up to 100% locally or remotely. This means that enough temporary space is required to buffer the data blocks. By continuously checking the space still available and by calculating the expected occupancy with new block files that are being processed, the full run is controlled. The compression of the files is not used because it is passed as an option to the rsync calls.

For example, the transmission of one terabyte from my office took about 19 hours on weekdays and about 16 hours on weekends. The difference in the available bandwidth was also noticeable from the early evening onward in which the temporary disk space was initially fully used locally, later, with more free bandwidth, considerably more is needed on the remote station.

Bandwidth

If the program is called with many connections, the throughput for all others using the same network slows down. But that also means, as just described, that the transmission at night or at the weekend significantly accelerates the process of data migration.

Mode of operation of ‘dd’

‘dd’ can only either read or write. For this reason, it is advantageous to process reasonably large blocks in one work step. At the same time, this program can address specific positions of a file or block device blockwise or bytewise. This makes it the only tool that is suitable for our purpose. Precise positioning is essential for splitting the data to be transmitted into clearly defined individual steps. The block and file size in ddtransfer.sh is always a multiple of the minimum block size of the file system of 512 bytes. Up to now I have only allowed one dd process per processor core (if there is only one core, there are two processes).

The way of working mentioned at the beginning of this section makes it somewhat cumbersome to determine whether ‘dd’ will still write to a file or whether the respective process has already been completed. While ‘dd’ still reads from the device – the larger the selected read block size, the longer the read process – the target file does not change and is also not considered open, so it cannot be checked by ‘lsof’. Based on the size of a file, this may have to be calculated in advance and then checked continuously. During the development of the script it has therefore proven to be useful to completely restart image block write processes after an interruption. The ‘status=progress’ option is offered in new implementations of ‘dd’, but it is not always available and I have not yet the time to check whether it can be used. Furthermore, a loss of time is compensated by the transfer time of the files when a ‘dd’ call is restarted.

‘ssh’ and ‘rsync’

Frequent ssh calls are necessary to execute the remote commands and control the processes. It turned out that the number of possible connections can also be a scarce resource. In order to avoid bottlenecks, various calls were combined to make the script more efficient. For further development, it would make sense to optimize this even more. Two new ssh connections are opened for each start of an rsync process. If one of the instances involved reboots, the rsync processes get stuck and interfere with subsequent calls. It remains to be seen which solution is most appropriate here. So far I have helped myself with restarting the VMs involved.

Invoking the command

The script should be called directly from a root shell. An ssh agent is also required to load a private ssh key. The login must also be possible remotely as ‘root’. This could be an example:

```
./ddtransfer.sh --local_device /dev/sdc --remote_device /dev/vdd
--TargetHost 46.16.76.151 --remote_transfer_dir /mnt --keep_logs
```

In this case is

- ’/dev/sdc‘ the local device to transmit
- ‘/dev/vdd‘ the storage volume of an addressed virtual machine the data will be written to

- ‘46.16.76.151‘ as IP of this VM
- ‘/mnt‘ mentions the remote directory to be used for the image files temporarily. Such a directory could be used locally as well (default is $HOME)
- ‘--keep_logs‘ lets the logs remain after completion of the run

The command

```
./ddtransfer.sh --restart report_ddt_15337519335231.log
```

restarts a formally invoked run. The file report_ddt_$(date +%s).log will be created at the first call of the transfer.

Completing the transfer for data migration with ddtransfer.sh

After transferring the image files, the report file is processed and checked to see if the start and end checksums do not match or are even missing. The absence of checksums calculated after writing each block to the target device may occur if the connection between the source and target host has been interrupted or the temporarily used disk space filled up. If the checksum is missing or divergent, it is recalculated. If it still differs, the block is completely read out, transferred and recalculated. The dd calls are logged completely so that they can be repeated easily. If the checksum is still not correct, a corresponding error message will be shown.

In the event of error messages, the log files are retained. After checking the situation, the individual steps can be restarted manually if required.

Perspective and outlook

For future development, it must be further investigated how the individual program steps relate to each other and to the other parts with regard to multiple calls. This would hopefully improve the balancing of data migration and thus speed up the overall processing.

Besides, improvements could be

- no transmission of blocks with the same checksum
- a better status display
- the dd option ‘conf=noerror,sync’
- complete clearing of hanging rsync and ssh processes in case of a new call of ddtransfer.sh without restarting source and target computers
- continue formerly started rsync jobs
- the extension of the status file, among other things to improve the resumption of the transfer and
- shorter file names in the temporary directories.