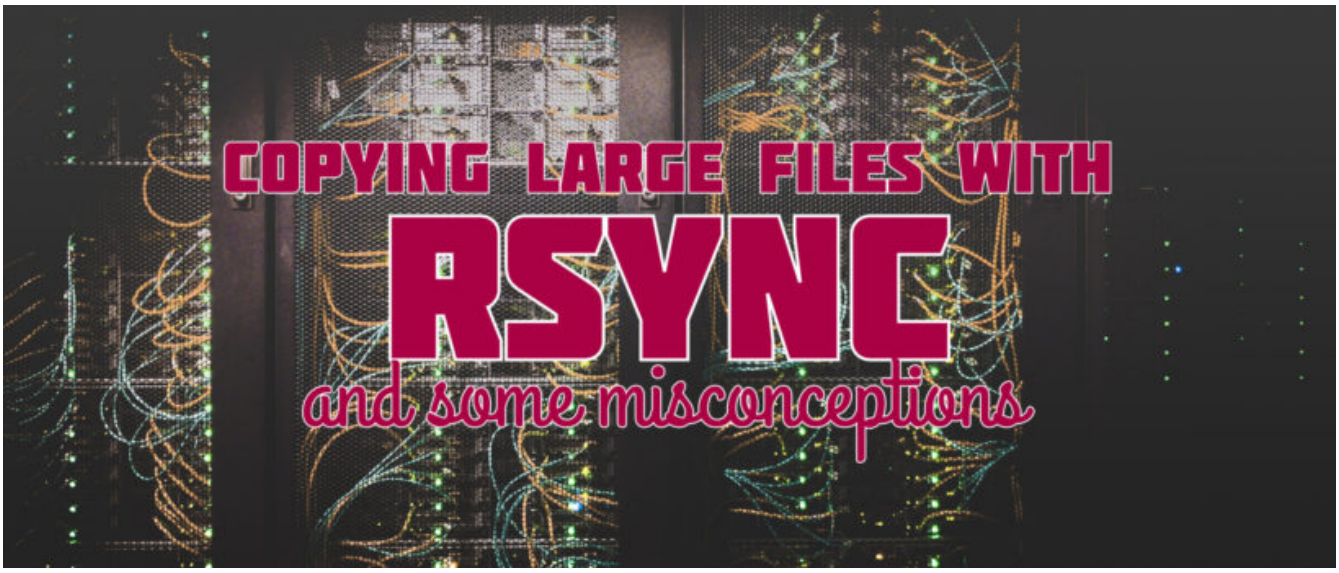
Copying large files with Rsync, and some misconceptions

By Daniel Leite de Abreu on September 16, 2019



There is a notion that a lot of people working in the IT industry often copy and paste from internet howtos. We all do it, and the copy-and-paste itself is not a problem. The problem is when we run things without understanding them.

Some years ago, a friend who used to work on my team needed to copy virtual machine templates from site A to site B. They could not understand why the file they copied was 10GB on site A but but it became 100GB on-site B.

The friend believed that *rsync* is a magic tool that should just "sync" the file as it is. However, what most of us forget is to understand what *rsync* really is, and how is it used, and the most important in my opinion is, where it come from. This article provides some further information about rsync, and an explanation of what happened in that story.

About rsync

rsync is a tool was created by Andrew Tridgell and Paul Mackerras who were motivated by the following problem:

Imagine you have two files, *file_A* and *file_B*. You wish to update *file_B* to be the same as *file_A*. The obvious method is to copy *file_A* onto *file_B*.

Now imagine that the two files are on two different servers connected by a slow communications link, for example, a dial-up IP link. If *file_A* is large, copying it onto *file_B* will be slow, and sometimes not even possible. To make it more efficient, you could compress *file_A* before sending it, but that would usually only gain a factor of 2 to 4.

Now assume that *file_A* and *file_B* are quite similar, and to speed things up, you take advantage of this similarity. A common method is to send just the differences between *file_A* and *file_B* down the link and then use such list of differences to reconstruct the file on the remote end.

The problem is that the normal methods for creating a set of differences between two files rely on being able to read both files. Thus they require that both files are available beforehand at one end of the link. If they are not both available on the same machine, these algorithms cannot be used. (Once you had copied the file over, you don't need the differences). This is the problem that *rsync* addresses.

The *rsync* algorithm efficiently computes which parts of a source file match parts of an existing destination file. Matching parts then do not need to be sent across the link; all that is needed is a reference to the part of the destination file. Only parts of the source file which are not matching need to be sent over.

The receiver can then construct a copy of the source file using the references to parts of the existing destination file and the original material.

Additionally, the data sent to the receiver can be compressed using any of a range of common compression algorithms for further speed improvements.

The rsync algorithm addresses this problem in a lovely way as we all might know.

After this introduction on *rsync*, Back to the story!

Problem 1: Thin provisioning

There were two things that would help the friend understand what was going on.

The problem with the file getting significantly bigger on the other size was caused by Thin Provisioning (TP) being enabled on the source system — a method of optimizing the efficiency of available space in Storage Area Networks (SAN) or Network Attached Storages (NAS).

The source file was only 10GB because of TP being enabled, and when transferred over using *rsync* without any additional configuration, the target destination was receiving the full 100GB of size. *rsync* could not do the magic automatically, it had to be configured.

The Flag that does this work is *-S* or *–sparse* and it tells *rsync* to handle sparse files efficiently. And it will do what it says! It will only send the sparse data so source and destination will have a 10GB file.

Problem 2: Updating files

The second problem appeared when sending over an updated file. The destination was now receiving just the 10GB, but the whole file (containing the virtual disk) was always transferred. Even when a single configuration file was changed on that virtual disk. In other words, only a small portion of the file changed.

The command used for this transfer was:

```
rsync –avS vmdk_file syncuser@host1:/destination
```

Again, understanding how *rsync* works would help with this problem as well.

The above is the biggest misconception about rsync. Many of us think *rsync* will simply send the delta updates of the files, and that it will automatically update only what needs to be updated. But this is not the default behaviour of *rsync*.

As the man page says, the default behaviour of *rsync* is to create a new copy of the file in the destination and to move it into the right place when the transfer is completed.

To change this default behaviour of *rsync*, you have to set the following flags and then rsync will send only the deltas:

```
--inplace               update destination files in-place
--partial               keep partially transferred files
--append                append data onto shorter files
--progress              show progress during transfer
```

So the full command that would do exactly what the friend wanted is:

```
rsync -av --partial --inplace --append --progress vmdk_file
syncuser@host1:/destination
```

Note that the sparse flag *-S* had to be removed, for two reasons. The first is that you can not use *–sparse* and *–inplace* together when sending a file over the wire. And second, when you once sent a file over with *–sparse*, you can't updated with *–inplace* anymore. Note that versions of rsync older than 3.1.3 will reject the combination of *–sparse* and *–inplace*.

So even when the friend ended up copying 100GB over the wire, that only had to happen once. All the following updates were only copying the difference, making the copy to be extremely efficient.

FEDORA PROJECT COMMUNITY

Daniel Leite de Abreu

A Father, a husband and another Open source enthusiast.