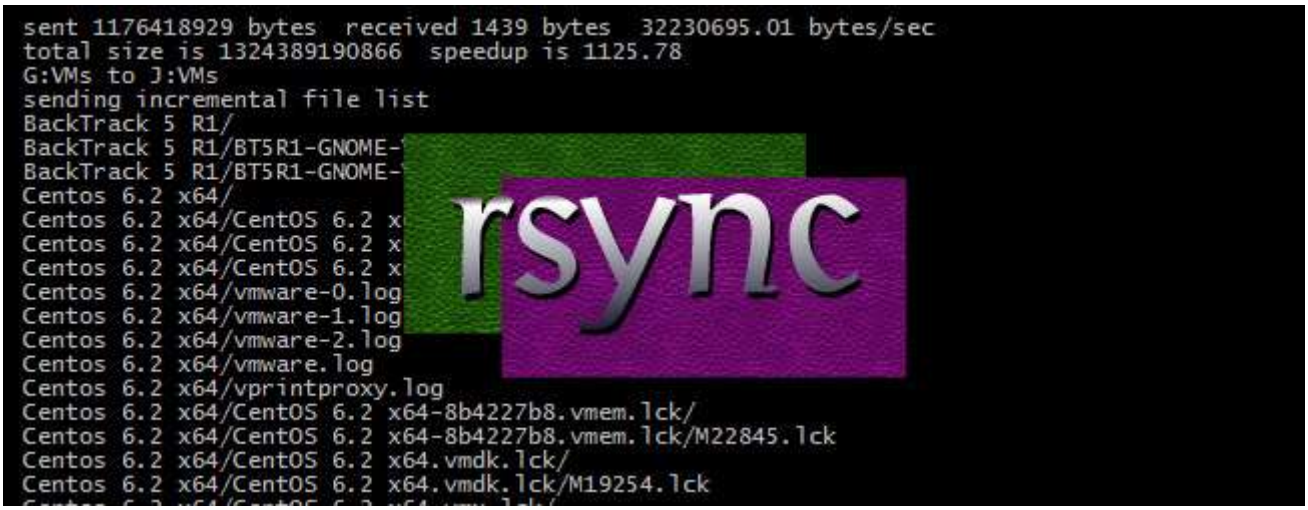
How to Use rsync to Backup Your Data on Linux

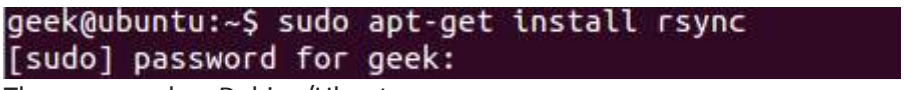
Go to: The Non-Beginner's Guide to Syncing Data with Rsync



rsync is a protocol built for Unix-like systems that provides unbelievable versatility for backing up and synchronizing data. It can be used locally to back up files to different directories or can be configured to sync across the Internet to other hosts.

It can be used on Windows systems but is only available through various ports (such as Cygwin), so in this how-to we will be talking about setting it up on Linux. First, we need to install/update the rsync client. On Red Hat distributions, the command is "yum install rsync" and on Debian it is "sudo apt-get install rsync."

The command on Red Hat/CentOS, after logging in as root (note that some recent distributions of Red Hat support the sudo method).



The command on Debian/Ubuntu.

Using rsync for local backups

In the first part of this tutorial, we will back up the files from Directory1 to Directory2. Both of these directories are on the same hard drive, but this would work exactly the same if the directories existed on two different drives. There are several different ways we can approach this, depending on what kind of backups you want to configure. For most purposes, the following line of code will suffice:

```
$ rsync -av --delete /Directory1/ /Directory2/
```

The code above will synchronize the contents of Directory1 to Directory2, and leave no differences between the two. If rsync finds that Directory2 has a file that Directory1 does not, it will delete it. If rsync finds a file that has been changed, created, or deleted in Directory1, it will reflect those same changes to Directory2.

There are a lot of different switches that you can use for rsync to personalize it to your specific needs. Here is what the aforementioned code tells rsync to do with the backups:

1. -a = recursive (recurse into directories), links (copy symlinks as symlinks), perms (preserve permissions), times (preserve modification times), group (preserve group), owner (preserve owner), preserve device files, and preserve special files.
2. -v = verbose. The reason I think verbose is important is so you can see exactly what rsync is backing up. Think about this: What if your hard drive is going bad, and starts deleting files without your knowledge, then you run your rsync script and it pushes those changes to your backups, thereby deleting all instances of a file that you did not want to get rid of?
3. –delete = This tells rsync to delete any files that are in Directory2 that aren't in Directory1. If you choose to use this option, I recommend also using the verbose options, for reasons mentioned above.

Using the script above, here's the output generated by using rsync to backup Directory1 to Directory2. Note that without the verbose switch, you wouldn't receive such detailed information.

```
[geek@localhost /]$ rsync -av --delete /Directory1/ /Directory2/
sending incremental file list
./
File1.txt
File2.jpg

sent 412 bytes  received 53 bytes  930.00 bytes/sec
total size is 249  speedup is 0.54
[geek@localhost /]$
```

The screenshot above tells us that File1.txt and File2.jpg were detected as either being new or otherwise changed from the copies existent in Directory2, and so they were backed up. Noob tip: Notice the trailing slashes at the end of the directories in my rsync command – those are necessary, be sure to remember them.

We will go over a few more handy switches towards the end of this tutorial, but just remember that to see a full listing you can type "man rsync" and view a complete list of switches to use.

That about covers it as far as local backups are concerned. As you can tell, rsync is very easy to use. It gets slightly more complex when using it to sync data with an external host over the Internet, but we will show you a simple, fast, and secure way to do that.

Using rsync for external backups

rsync can be configured in several different ways for external backups, but we will go over the most practical (also the easiest and most secure) method of tunneling rsync through SSH. Most servers and even many clients already have SSH, and it can be used for your rsync backups. We will show you the process to get one Linux machine to backup to another on a local network. The process would be the exact same if one host were out on the internet somewhere, just note that port 22 (or whatever port you have SSH configured on), would need to be forwarded on any network equipment on the server's side of things.

On the server (the computer that will be receiving the backups), make sure SSH and rsync are installed.

```
# yum -y install ssh rsync

# sudo apt-get install ssh rsync
```

Other than installing SSH and rsync on the server, all that really needs to be done is to setup the repositories on the server where you would like the files backed up, and make sure that SSH is locked down. Make sure the user you plan on using has a complex password, and it may also be a good idea to switch the port that SSH listens on (default is 22).

We will run the same command that we did for using rsync on a local computer, but include the necessary additions for tunneling rsync through SSH to a server on my local network. For user "geek" connecting to "192.168.235.137" and using the same switches as above (-av –delete) we will run the following:

```
$ rsync -av –delete -e ssh /Directory1/ geek@192.168.235.137:/Directory2/
```

If you have SSH listening on some port other than 22, you would need to specify the port number, such as in this example where I use port 12345:

```
$ rsync -av –delete -e 'ssh -p 12345' /Directory1/
geek@192.168.235.137:/Directory2/
```

```
[geek@localhost /]$ rsync -av --delete -e ssh /Directory1/ geek@192.168.235.137:/Directory2/
geek@192.168.235.137's password:
sending incremental file list
./
File1.txt
File2.jpg

sent 416 bytes  received 53 bytes  44.67 bytes/sec
total size is 249  speedup is 0.53
[geek@localhost /]$
```

As you can see from the screenshot above, the output given when backing up across the network is pretty much the same as when backing up locally, the only thing that changes is the command you use. Notice also that it prompted for a password. This is to authenticate with SSH. You can set up RSA keys to skip this process, which will also simplify automating rsync.

Automating rsync backups

Cron can be used on Linux to automate the execution of commands, such as rsync. Using Cron, we can have our Linux system run nightly backups, or however often you would like them to run.

To edit the cron table file for the user you are logged in as, run:

```
$ crontab -e
```

You will need to be familiar with vi in order to edit this file. Type “I” for insert, and then begin editing the cron table file.

Cron uses the following syntax: minute of the hour, hour of the day, day of the month, month of the year, day of the week, command.

It can be a little confusing at first, so let me give you an example. The following command will run the rsync command every night at 10 PM:

```
0 22 * * * rsync -av --delete /Directory1/ /Directory2/
```

The first “0” specifies the minute of the hour, and “22” specifies 10 PM. Since we want this command to run daily, we will leave the rest of the fields with asterisks and then paste the rsync command.

After you are done configuring Cron, press escape, and then type “:wq” (without the quotes) and press enter. This will save your changes in vi.

Cron can get a lot more in-depth than this, but to go on about it would be beyond the scope of this tutorial. Most people will just want a simple weekly or daily backup, and what we have shown you can easily accomplish that. For more info about Cron, please see the man pages.

Other useful features

Another useful thing you can do is put your backups into a zip file. You will need to specify where you would like the zip file to be placed, and then rsync that directory to your backup directory. For example:

```
$ zip /ZippedFiles/archive.zip /Directory1/ && rsync -av --delete
/ZippedFiles/ /Directory2/
```

```
[geek@localhost /]$ zip /ZippedFiles/archive.zip /Directory1/ &&
rsync -av --delete /ZippedFiles/ /Directory2/
  adding: Directory1/ (stored 0%)
sending incremental file list
./
archive.zip

sent 294 bytes  received 34 bytes  656.00 bytes/sec
total size is 344  speedup is 1.05
[geek@localhost /]$
```

The command above takes the files from Directory1, puts them in /ZippedFiles/archive.zip and then rsyncs that directory to Directory2. Initially, you may think this method would prove inefficient for large backups, considering the zip file will change every time the slightest alteration is made to a file. However, rsync only transfers the changed data, so if your zip file is 10 GB, and then you add a text file to Directory1, rsync will know that is all you added (even though it's in a zip) and transfer only the few kilobytes of changed data.

There are a couple of different ways you can encrypt your rsync backups. The easiest method is to install encryption on the hard drive itself (the one that your files are being backed up to). Another way is to encrypt your files before sending them to a remote server (or other hard drive, whatever you happen to be backing up to). We'll cover these methods in later articles.

Whatever options and features you choose, rsync proves to be one of the most efficient and versatile backup tools to date, and even a simple rsync script can save you from losing your data.

End part 1

The Non-Beginner's Guide to Syncing Data with Rsync

The rsync protocol can be pretty simple to use for ordinary backup/synchronization jobs, but some of its more advanced features may surprise you. In this article, we're going to show how even the biggest data hoarders and backup enthusiasts can wield rsync as a single solution for all of their data redundancy needs.

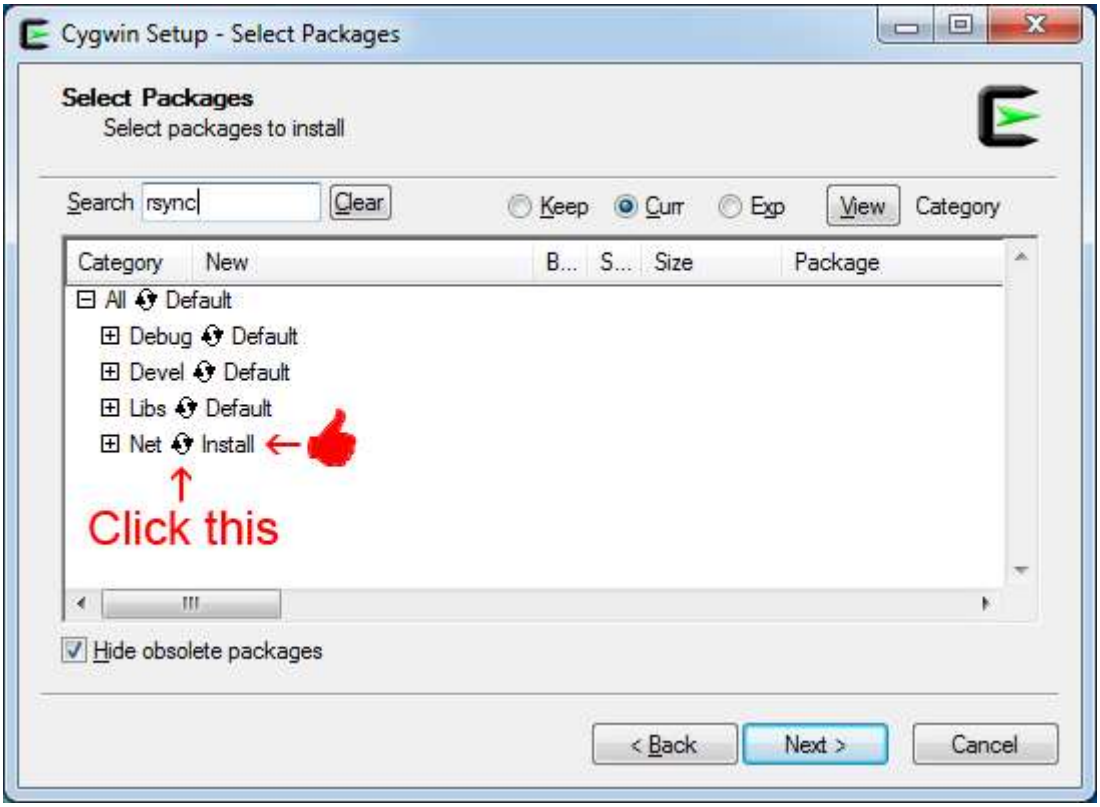
Warning: Advanced Geeks Only

If you're sitting there thinking "What the heck is rsync?" or "I only use rsync for really simple tasks," you may want to check out our previous article on how to use rsync to backup your data on Linux, which gives an introduction to rsync, guides you through installation, and showcases its more basic functions. Once you have a firm grasp of how to use rsync (honestly, it isn't that complex) and are comfortable with a Linux terminal, you're ready to move on to this advanced guide.

Running rsync on Windows

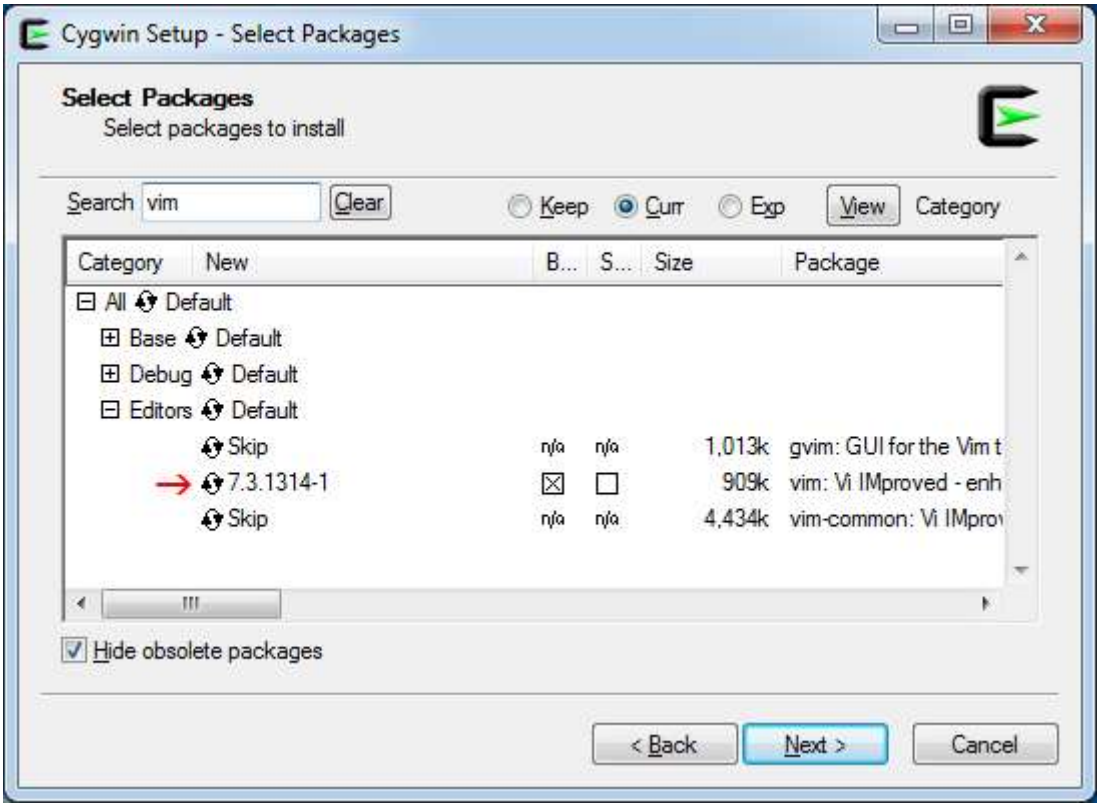
First, let's get our Windows readers on the same page as our Linux gurus. Although rsync is built to run on Unix-like systems, there's no reason that you shouldn't be able to use it just as easily on Windows. Cygwin produces a wonderful Linux API that we can use to run rsync, so head over to their website and download the 32-bit or 64-bit version, depending on your computer.

Installation is straightforward; you can keep all options at their default values until you get to the "Select Packages" screen.

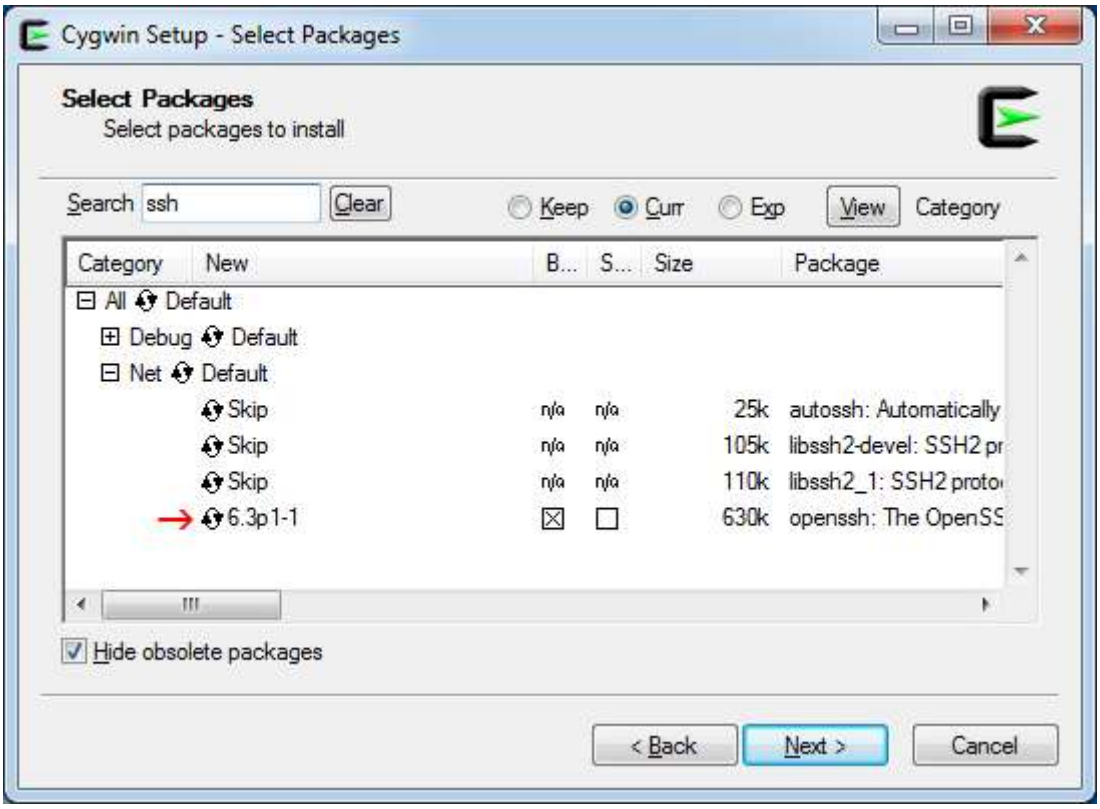


Now you need to do the same steps for Vim and SSH, but the packages are going to look a bit different when you go to select them, so here are some screenshots:

Installing Vim:



Installing SSH:



After you've selected those three packages, keep clicking next until you finish the installation. Then you can open Cygwin by clicking on the icon that the installer placed on your desktop.

rsync Commands: Simple to Advanced

Now that the Windows users are on the same page, let's take a look at a simple rsync command, and show how the use of some advanced switches can quickly make it complex.

Let's say you have a bunch of files that need backed up – who doesn't these days? You plug in your portable hard drive so you can backup your computers files, and issue the following command:

```
rsync -a /home/geek/files/ /mnt/usb/files/
```

Or, the way it would look on a Windows computer with Cygwin:

```
rsync -a /cygdrive/c/files/ /cygdrive/e/files/
```

Pretty simple, and at that point there's really no need to use rsync, since you could just drag and drop the files. However, if your other hard drive already has some of the files

and just needs the updated versions plus the files that have been created since the last sync, this command is handy because it only sends the new data over to the hard drive. With big files, and especially transferring files over the internet, that is a big deal.

Backing up your files to an external hard drive and then keeping the hard drive in the same location as your computer is a very bad idea, so let's take a look at what it would require to start sending your files over the internet to another computer (one you've rented, a family member's, etc).

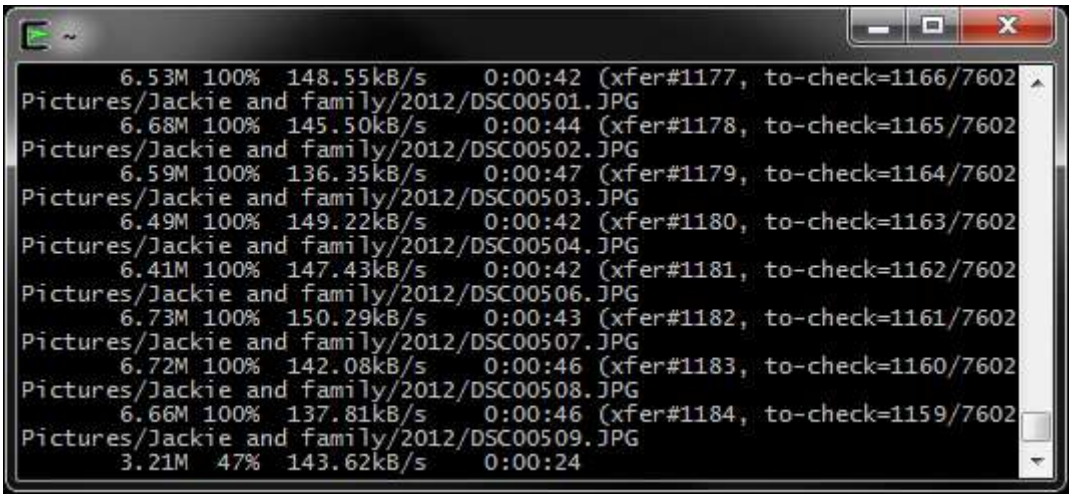
```
rsync -av --delete -e 'ssh -p 12345' /home/geek/files/
geek2@10.1.1.1:/home/geek2/files/
```

The above command would send your files to another computer with an IP address of 10.1.1.1. It would delete extraneous files from the destination that no longer exist in the source directory, output the filenames being transferred so you have an idea of what's going on, and tunnel rsync through SSH on port 12345.

The `-a -v -e --delete` switches are some of the most basic and commonly used; you should already know a good deal about them if you're reading this tutorial. Let's go over some other switches that are sometimes ignored but incredibly useful:

`--progress` – This switch allows us to see the transfer progress of each file. It's particularly useful when transferring large files over the internet, but can output a senseless amount of information when just transferring small files across a fast network.

An rsync command with the `--progress` switch as a backup is in progress:



`--partial` – This is another switch that is particularly useful when transferring large files over the internet. If rsync gets interrupted for any reason in the middle of a file transfer, the partially transferred file is kept in the destination directory and the transfer is resumed where it left off once the rsync command is executed again. When transferring large files over the internet (say, a couple of gigabytes), there's nothing worse than having a few

second internet outage, blue screen, or human error trip up your file transfer and having to start all over again.

`-P` – this switch combines `--progress` and `--partial`, so use it instead and it will make your rsync command a little neater.

`-z` or `--compress` – This switch will make rsync compress file data as it's being transferred, reducing the amount of data that has to be sent to the destination. It's actually a fairly common switch but is far from essential, only really benefiting you on transfers between slow connections, and it does nothing for the following types of files: 7z, avi, bz2, deb, g,z iso, jpeg, jpg, mov, mp3, mp4, ogg, rpm, tbz, tgz, z, zip.

`-h` or `--human-readable` – If you're using the `--progress` switch, you'll definitely want to use this one as well. That is, unless you like to convert bytes to megabytes on the fly. The `-h` switch converts all outputted numbers to human-readable format, so you can actually make sense of the amount of data being transferred.

`-n` or `--dry-run` – This switch is essential to know when you're first writing your rsync script and testing it out. It performs a trial run but doesn't actually make any changes – the would-be changes are still outputted as normal, so you can read over everything and make sure it looks okay before rolling your script into production.

`-R` or `--relative` – This switch must be used if the destination directory doesn't already exist. We will use this option later in this guide so that we can make directories on the target machine with timestamps in the folder names.

`--exclude-from` – This switch is used to link to an exclude list that contains directory paths that you don't want backed up. It just needs a plain text file with a directory or file path on each line.

`--include-from` – Similar to `--exclude-from`, but it links to a file that contains directories and file paths of data you want backed up.

`--stats` – Not really an important switch by any means, but if you are a sysadmin, it can be handy to know the detailed stats of each backup, just so you can monitor the amount of traffic being sent over your network and such.

`--log-file` – This lets you send the rsync output to a log file. We definitely recommend this for automated backups in which you aren't there to read through the output yourself. Always give log files a once over in your spare time to make sure everything is working properly. Also, it's a crucial switch for a sysadmin to use, so you're not left wondering how your backups failed while you left the intern in charge.

Let's take a look at our rsync command now that we have a few more switches added:

```
rsync -avzhP --delete --stats --log-file=/home/geek/rsynclogs/backup.log --
exclude-from '/home/geek/exclude.txt' -e 'ssh -p 12345' /home/geek/files/
geek2@10.1.1.1:/home/geek2/files/
```

The command is still pretty simple, but we still haven't created a decent backup solution. Even though our files are now in two different physical locations, this backup does nothing to protect us from one of the main causes of data loss: human error.

Snapshot Backups

If you accidentally delete a file, a virus corrupts any of your files, or something else happens whereby your files are undesirably altered, and then you run your rsync backup script, your backed up data is overwritten with the undesirable changes. When such a thing occurs (not if, but when), your backup solution did nothing to protect you from your data loss.

The creator of rsync realized this, and added the `--backup` and `--backup-dir` arguments so users could run differential backups. The very first example on rsync's website shows a script where a full backup is run every seven days, and then the changes to those files are backed up in separate directories daily. The problem with this method is that to recover your files, you have to effectively recover them seven different times. Moreover, most geeks run their backups several times a day, so you could easily have 20+ different backup directories at any given time. Not only is recovering your files now a pain, but even just looking through your backed up data can be extremely time consuming – you'd have to know the last time a file was changed in order to find its most recent backed up copy. On top of all that, it's inefficient to run only weekly (or even less often in some cases) incremental backups.

Snapshot backups to the rescue! Snapshot backups are nothing more than incremental backups, but they utilize hardlinks to retain the file structure of the original source. That may be hard to wrap your head around at first, so let's take a look at an example.

Pretend we have a backup script running that automatically backs up our data every two hours. Whenever rsync does this, it names each backup in the format of: Backup-month-day-year-time.

So, at the end a typical day, we'd have a list of folders in our destination directory like this:

```
[root@localhost backups]# ls
Backup-10-1-2013-10AM  Backup-10-1-2013-2AM  Backup-10-1-2013-6AM
Backup-10-1-2013-10PM  Backup-10-1-2013-2PM  Backup-10-1-2013-6PM
Backup-10-1-2013-12AM  Backup-10-1-2013-4AM  Backup-10-1-2013-8AM
Backup-10-1-2013-12PM  Backup-10-1-2013-4PM  Backup-10-1-2013-8PM
```

When traversing any of those directories, you'd see every file from the source directory exactly as it was at that time. Yet, there would be no duplicates across any two directories. rsync accomplishes this with the use of hardlinking through the `--link-dest=DIR` argument.

Of course, in order to have these nicely- and neatly-dated directory names, we're going to have to beef up our rsync script a bit. Let's take a look at what it would take to accomplish a backup solution like this, and then we'll explain the script in greater detail:

```
#!/bin/bash

#copy old time.txt to time2.txt

yes | cp ~/backup/time.txt ~/backup/time2.txt

#overwrite old time.txt file with new time

echo `date +"%F-%I%p"` > ~/backup/time.txt

#make the log file

echo "" > ~/backup/rsync-`date +"%F-%I%p"`.log

#rsync command

rsync -avzhPR --chmod=Du=rwx,Dgo=rx,Fu=rw,Fgo=r --delete --stats --log-file=~/backup/rsync-`date +"%F-%I%p"`.log --exclude-from '~/exclude.txt' --link-dest=/home/geek2/files/`cat ~/backup/time2.txt` -e 'ssh -p 12345' /home/geek/files/ geek2@10.1.1.1:/home/geek2/files/`date +"%F-%I%p"`/

#don't forget to scp the log file and put it with the backup

scp -P 12345 ~/backup/rsync-`cat ~/backup/time.txt`.log geek2@10.1.1.1:/home/geek2/files/`cat ~/backup/time.txt`/rsync-`cat ~/backup/time.txt`.log
```

That would be a typical snapshot rsync script. In case we lost you somewhere, let's dissect it piece by piece:

The first line of our script copies the contents of time.txt to time2.txt. The yes pipe is to confirm that we want to overwrite the file. Next, we take the current time and put it into time.txt. These files will come in handy later.

The next line makes the rsync log file, naming it rsync-date.log (where date is the actual date and time).

Now, the complex rsync command that we've been warning you about:

`-avzhPR`, `-e`, `--delete`, `--stats`, `--log-file`, `--exclude-from`, `--link-dest` — Just the switches we talked about earlier; scroll up if you need a refresher.

`--chmod=Du=rwx,Dgo=rx,Fu=rw,Fgo=r` – These are the permissions for the destination directory. Since we are making this directory in the middle of our rsync script, we need to specify the permissions so that our user can write files to it.

The use of date and cat commands

We're going to go over each use of the date and cat commands inside the rsync command, in the order that they occur. Note: we're aware that there are other ways to accomplish this functionality, especially with the use of declaring variables, but for the purpose of this guide, we've decided to use this method.

The log file is specified as:

```
~/backup/rsync-`date +"%F-%I%p"`.log
```

Alternatively, we could have specified it as:

```
~/backup/rsync-`cat ~/backup/time.txt`.log
```

Either way, the `--log-file` command should be able to find the previously created dated log file and write to it.

The link destination file is specified as:

```
--link-dest=/home/geek2/files/`cat ~/backup/time2.txt`
```

This means that the `--link-dest` command is given the directory of the previous backup. If we are running backups every two hours, and it's 4:00PM at the time we ran this script, then the `--link-dest` command looks for the directory created at 2:00PM and only transfers the data that has changed since then (if any).

To reiterate, that is why time.txt is copied to time2.txt at the beginning of the script, so the `--link-dest` command can reference that time later.

The destination directory is specified as:

```
geek2@10.1.1.1:/home/geek2/files/`date +"%F-%I%p"`
```

This command simply puts the source files into a directory that has a title of the current date and time.

Finally, we make sure that a copy of the log file is placed inside the backup.

```
scp -P 12345 ~/backup/rsync-`cat ~/backup/time.txt`.log
geek2@10.1.1.1:/home/geek2/files/`cat ~/backup/time.txt`/rsync-`cat
~/backup/time.txt`.log
```

We use secure copy on port 12345 to take the rsync log and place it in the proper directory. To select the correct log file and make sure it ends up in the right spot, the time.txt file must be referenced via the cat command. If you're wondering why we decided to cat time.txt instead of just using the date command, it's because a lot of time could have transpired while the rsync command was running, so to make sure we have the right time, we just cat the text document we created earlier.

Automation

Use Cron on Linux or Task Scheduler on Windows to automate your rsync script. One thing you have to be careful of is making sure that you end any currently running rsync processes before continuing a new one. Task Scheduler seems to close any already running instances automatically, but for Linux you'll need to be a little more creative.

Most Linux distributions can use the pkill command, so just be sure to add the following to the beginning of your rsync script:

```
pkill -9 rsync
```

Encryption

Nope, we're not done yet. We finally have a fantastic (and free!) backup solution in place, but all of our files are still susceptible to theft. Hopefully, you're backing up your files to some place hundreds of miles away. No matter how secure that faraway place is, theft and hacking can always be problems.

In our examples, we have tunneled all of our rsync traffic through SSH, so that means all of our files are encrypted while in transit to their destination. However, we need to make sure the destination is just as secure. Keep in mind that rsync only encrypts your data as it is being transferred, but the files are wide open once they reach their destination.

One of rsync's best features is that it only transfers the changes in each file. If you have all of your files encrypted and make one minor change, the entire file will have to be retransmitted as a result of the encryption completely randomizing all of the data after any change.

For this reason, it's best/easiest to use some type of disk encryption, such as BitLocker for Windows or dm-crypt for Linux. That way, your data is protected in the event of theft, but files can be transferred with rsync and your encryption won't hinder its performance.

There are other options available that work similarly to rsync or even implement some form of it, such as Duplicity, but they lack some of the features that rsync has to offer.

After you've setup your snapshot backups at an offsite location and encrypted your source and destination hard drives, give yourself a pat on the back for mastering rsync and implementing the most foolproof data backup solution possible.