Backups: simple

If you want more reliable backups, you need an offsite backup process that happens no matter how absent-minded you are. **Juliet Kemp** explains how.

On-site backups – an external disk, for example, or even tapes – are all well and good in their own way. Indeed, I'd go so far as to say that they're mandatory if you care at all about your data. If you have lost data, you'll know I'm right here. If you haven't, please take my word for it rather than finding out the hard way!

But what about when some light-fingered type lifts your external disk along with your laptop, or a solar flar creates a highly localised electro magnetic pulse that wipes every storage medium in the office (unlikely, but if you're paranoid enough, you'll have planned for it anyway)? That's when you need an off-site backup. We're told all the time to back up regularly but realistically, the the only way to ensure that backups happen as often as they should is to make the process automatic.

We'll look here at two options for off-site backups. The first, using *Rsync*, is useful if you have shell access to another machine located elsewhere, and the second uses Google's Gmail service for storage. Both are network-based, which means that they have some size limitations, which will depend in part on the speed of your connection. At the end of the tutorial I'll discuss possibilities for larger quantities of data.

Our expert

Juliet Kemp has been playing with Linux systems for around eight years now, after discovering that it was an excellent way to avoid revising for her finals. She's been a full-time Linux sysadmin for several years, and is worryingly obsessive about backups.

Rsync

If you have access to an off-site shell account (you may get one with your hosting package, or be able to pay for one), you can use that old Unix standby, *rsync*.

The basic *rsync* command is:

```
rsync /dir/source /dir/destination
```

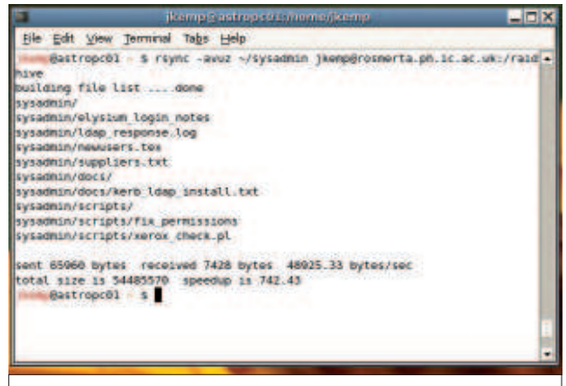
This looks at **/dir/source**, and compares it with **/dir/destination**. If any of the files in the source directory are new, or have been updated, *rsync* will copy them across to the destination directory. This means that the first time you run it on a particular directory, it will copy everything, and may take quite a while. The second and subsequent times it will copy only the files that have changed, so it'll be much faster. This saves both time and (with off-site transfer) network capacity.

The above command as written will only synchronise from one local directory to another local directory – basically acting as a gloried version of *cp*. What is important for our purposes is that **/dir/destination** can in fact be on another machine: referred to as **machinename.example.com:/dir/destination**. You can also reverse this, and copy from a source directory on a remote machine to a local destination directory, which is useful when it comes to restoring data.

Rsync can run with a variety of options. The ones you probably want are **-avuz**. We'll run down those one by one: **-v** sets the verbose option, so you can follow what's happening; **-a** is the archive option, which recurses down the directory structure and also keeps ownership, permissions, edit times, and so on intact; **-u** sets the update option, which keeps intact any files that are newer on the destination directory than on the source; and **-z** compresses during transfer, speeding things up a little.

Symlinks are not followed by default, but just saved as symlinks. If you need them to be followed (and the files that they lead to to be copied in full), you can set that as an option. Check out **man rsync** for that and the other available options.

So, if your shell account is on **offsite.example.com**, and you have a backup directory set up at **/home/user/backup**, you can back up the directory **/test** like this:



Using rsync's verbose option (-v) means you'll get loads of useful feedback on how the backups are proceeding.

data insurance

```
rsync -avuz /test user@offsite.example.com:/home/user/backup
```

You will need *rsyncd* to be running on the remote machine. If you have a commercial shell account this will probably already be the case; if not, talk to the sysadmin. If the remote machine belongs to you, install the relevant package for your distribution. On Debian, you'll also need to edit **/etc/default/rsync** to set **RSYNC_ENABLE** to true, then run **/etc/init.d/rsync start**.

You'll be challenged for your SSH password – we'll look at how to avoid this in a moment – then *rsync* will tell you that it's building the file list. This means that it's working out what files to copy across. In this case, it'll copy all of them, as this is a first backup. In due course, it will start copying the files – and since we used the **-v** switch, it'll tell you about each file. Once *rsync* has finished, your backed-up files will be in **/home/user/backup/test** on **offsite.example.com**.

At this point it's worth thinking about the way in which *rsync* deals with directories. It acts differently depending on whether or not there is a trailing slash at the end of the source directory. As a simple example:

```
rsync /test/one /backup
```

will transfer all the files in **/test/one** to **/backup/one**. In other words, it copies the whole directory (and its contents) by name. The following code:

```
rsync /test/one/ /backup
```

(note trailing slash on the source directory) will copy all the files in **/test/one** to **/backup**. It copies only the contents, not the directory itself. Most of the time, you'll want to use the first version, because it's tidier.

Once your first run has finished, try changing a single file and running the same command again. You'll see that this time, only the file that you changed is copied over.

Password-free SSH

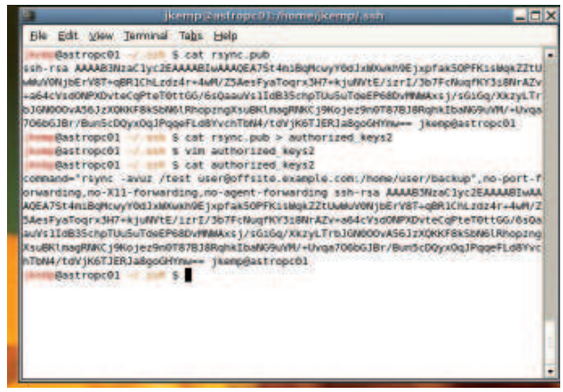
OK, so *rsync* is copying files happily on to your off-site machine. But at present it still requires an SSH password – not good for automation! What you need to do to get around this, as you might have guessed, is to set up a password-free SSH key.

First of all, you need to create your private key. On your home machine, fire up a terminal window and type

```
ssh-keygen -t rsa -f ~/.ssh/rsync
```

to generate an RSA key (you can also generate other types of key – check the man page for details) and save it in **/home/ser/.ssh/rsync**. This will prompt you for a passphrase – just hit Return to get an empty passphrase. Once it's done, you should have files **/home/user/.ssh/rsync** and **/home/user/.ssh/rsync.pub**.

What you would normally do here with a regular passworded key is add the contents of **/home/user/.ssh/rsync.pub** to the file **/home/user/.ssh/authorized_keys2** on **offsite.example.com**, and off you would go. However, in this case, that's dangerous, because this key has no passphrase. So anyone who got access to your home machine could get access to **offsite.example.com**, and do anything they wanted to your files there or to anything else you have access to – not very good from a security perspective!



› **You'll need to edit Authorized_keys2 so that your privete key (which has no password) can run only your backup command.**

What you can do to reduce the security risk is to edit the **authorized_keys2** file on **offsite.example.com** in order to restrict that key so that it is permitted only to run your backup command. Copy **/home/user/.ssh/rsync.pub** to **tmpfile** and edit **tmpfile** so that its single line (It'll be a big block of what looks like random characters, but is in fact your private key) has this at the start of it:

```
command="rsync -avuz -e "ssh -i /home/user/.ssh/rsync" /test
user@offsite.example.com:/home/user/backup",no-port-
forwarding,no-X11-forwarding,no-agent-forwarding
```

Now add the contents of **tmpfile** to **/home/user/.ssh/authorized_keys2** on **offsite.example.com** (cut and paste, or use *cat*, as you prefer). Make sure it's all one line.

The *rsync* command needs to know where to find your private key, so you'll need to run:

```
rsync -avuz -e "ssh -i /home/user/.ssh/rsync" /test user@offsite.
example.com:/home/user/backup
```

This specifies the SSH key to use. This line must match the command you put in the **authorized_keys2** file above.

When you run it, you should see *rsync* connecting without asking for a passphrase, building its file list, and then copying across any files you've changed since the last run. Success!

Automating with cron

It is a sad but undeniable truth that if you have to remember to do something, then backups just won't happen often enough. So the last stage is to make this happen automatically. Type **crontab -e** and add this line to your crontab file:

```
5 0 * * * rsync -auz /test user@offsite.example.com:/home/user/
backup
```

Note that the **-v** option isn't there any more – you don't want lots of *Cron* output. Obviously, if you want to *rsync* a directory other than **/test**, you should change that. Remember to change the **authorized_keys2** file on **offsite.example.com** as well! It must match the command you're running. The numbers at the start of the crontab entry mean that this will run at five minutes past midnight (0), every day (check the crontab man page for more details). And you're done – your backup will now run automatically every night. »

Quick tip

If you need information about any Unix command, **man command name** is a good place to start.

Gmail

» Gmail already offers 5GB of storage for free, and has plans to steadily increase this (and if you need more, you can pay for an account with even more space), so it can be a good off-site backup solution for some files. You can set up a Perl script to email a given set of files to your Gmail account, then run it automatically every night with *Cron*.

Detailed description of Perl coding is outside the scope of this tutorial, but the script sould be fairly clear. You'll need to have the Perl modules **Net::SMTP**, **File::Find**, **Mime::Lite**, **Archive::Tar**, and **IO::Zlib** installed for this script. In Debian, the first two of these are a default part of the regular Perl install; the others are available as **libmime-lite-perl**, **libarchive-tar-perl** and **libio-zlib-perl**.

If you can't get these via your distribution, you can install them with CPAN, using the line:

```
perl -MCPAN -e "install Net::SMTP"
```

(substitute the other module names for Net::SMTP as appropriate).

OK, now for that script:

```
#!/usr/bin/perl -w
use strict;
use Archive::Tar;
use File::Find;
use MIME::Lite;
use Net::SMTP;

sub sendmail();
sub wanted();

my $email       = 'my.
name@gmail.com';
my $smtpserver  = "smtp.example.com";
my @archive_list;
my $backup_dir  = "/home/jkemp/personal/";
my $tarfile     = "/home/jkemp/gmailtar.tgz";

find ( \&wanted, $backupdir );

Archive::Tar->create_archive($tarfile, "1", @archive_list);
sendmail();
sub sendmail() {
   my $msg = MIME::Lite->new(To   => $email,
                     Subject => "Backup email",
                     Type  => "multipart/mixed");

   $msg->attach(Type  => "application/gzip",
            Path  => $tarfile,
            Filename  => "backup.tgz");
   $msg->send('smtp', $smtpserver);
}
```

Remember that you'll need to change the permissions so that it's user-executable before you can run it (**chmod u+x gmail.pl**).

Quick tip

When using subroutines, you must either define them before they are first used, or declare them first, and define them at the end – we've done the latter here.

> **"Computers are much better at remembering things than humans are."**

Perl stew

OK, let's take a quick look at that script. The **-w** flag on the first line means that Perl will warn you if you're doing anything dubious – things that might indicate a typo or thinko. The next line, **use strict**, is another error-catching measure that makes you declare all your variables before using them. I'd recommend always using both of these in your Perl scripts. It's slightly more hassle but saves a lot of debugging time.

The **find** command (from **File::Find**) recurses over the directory you specify (here it's **/home/jkemp/personal/**). Use a small directory – as we'll see later, this method is size-limited. The **find** command calls the **wanted** subroutine, which is declared at the start of the script and defined later. The subroutine checks that the name it's been passed actually corresponds to a file, and if so, adds the full name of that file to the array of files to be archived.

The line starting **Archive::Tar** does all the heavy lifting. It creates a tar archive with name as given by the variable **$tarfile**, compressed (that's where the **1** comes in – 'true' means compress, 'false' means don't compress), from the files in **@backupdir**.

Then the script sends the email. (I've put this as a subroutine so that if you're testing, it's easy to take the email-sending line out while you're checking that your archive does everything it wants to.) The email code is fairly self-explanatory. It creates a new MIME message, attaches the *gzip*ped backup file, and sends the message. You'll need to know the name of your SMTP server – ask your ISP if in doubt, or check the config on your email program.

To test that this is working properly, first comment out the **sendmail()** line by adding a hash (**#**) character at the start, then resave the program. Then run it (**/path/to/gmail.pl**) and check the location of your **gmailtar.tgz** file. If it's there, move it into a new clean test directory, and unzip it (**tar -zxf gmailtar.tgz**). Are all the files there? Great. Now uncomment that *Sendmail* line, save, and run the program again. Check your Gmail account – you should have a new mail with *gzip*ped attachment.

This script creates a tarball and leaves it on your computer. To delete this, add the line

```
unlink $tarfile;
```

after the **sendmail()** line. Alternatively, it will just be overwritten the next time the script runs. You probably don't want to have the delete line in during testing, because it can be handy to keep the file around so you can see what's going on. To make this run daily, as with the first method, set it up in your crontab.

The third way

The major downside to this is that there is a limit on how much data you can send this way. Very large emails are a nuisance and can cause network slowdown, and some servers will reject them, so this method is really only useful for small numbers of smallish files. Text files tend to be small, so by all means email the current draft of your new bestselling novel to yourself every night. It's not a useful option as a backup for your photos or music files, though. Obviously this would also work with any other webmail account;

CPAN

CPAN is an enormous collection of free Perl modules – reusable Perl toolboxes, in effect. There's a pain-free installation method, as described in the tutorial above. If you're writing Perl at all regularly, it is well worth checking out. Because anyone can upload to CPAN, quality does vary, but there's some good stuff in there, and the docs can be very helpful. At its best it can save an awful lot of wheel-reinventing.

To identify the most valuable CPAN modules, it's helpful to check the 'kwalitee' scores on the CPAN Testing Service page (**http://cpants.perl.org/kwalitee.html**), look at the recommended CPAN modules on the Perl Foundation website (**www.perlfoundation.org/perl5/index.cgi?recommended_cpan_modules**), and check out CPAN Ratings (**http://cpanratings.perl.org**).

Anacron

If your computer isn't always on – for example, if you use a laptop – you can use *Anacron* instead of *Cron*.

With *Cron*, you schedule a job at a specific date and time. If at that particular time the machine is down, the job just won't run (until the next time it's scheduled). With *Anacron*, you schedule a job to run at specific intervals – for example, daily, weekly, or monthly. *Anacron* will try to keep as closely to this schedule as system uptime permits. So if a job is supposed to run daily, and when the computer is switched on the *Anacron* daemon finds that it hasn't run in the last 24 hours, the job will be run there and then.

The downside of *Anacron* is that it can only run at intervals of one or more days – unlike *Cron*, which can run at intervals as small as one minute. For our purposes this isn't a problem, as we want the backup to run only daily. The other issue is that *Anacron* must be configured by the root user, whereas *Cron* can be used by any user. However, if this is your own machine you presumably have root access!

Edit the file **/etc/anacrontab**, as root, to add the line:

```
1 5 backup rsync -auz -e "ssh -i /home/user/.ssh/rsync" /test
user@offsite.example.com:/home/user/backup
```

This line will run the given rsync command every day (first parameter), with a delay of 5 minutes (second parameter), and identify the job logs as "backup" (third parameter).

we've just used Gmail because it's the one that offers the most free space. Both the automatic network-based options discussed here are fundamentally a fairly limited solution to the problem of off-site backup. The main problem is one of data size – sending data across the network takes time, and the more data you have, the more time it takes. (It also costs money, on most home broadband plans!) Neither of these methods are really suitable as off-site backups for, say, your music collection or photo collection. They're more for files such as text files and other important smaller files.

One alternative option would be to use an external disk to minimise the initial overhead. You could dump your music and photos on to an external disk at home, then take it to your off-site machine (maybe in your office, if that's on overnight, though you may want to check this with your employers!) and hook it up there. After this, any updates would be smaller, and could be done via *rsync*.

Alternatively you could have two disks, one at home and one at work, and swap them weekly. Hook up a disk at home, and set up *rsync* using *Cron* to run locally every night, and make a backup on to the external disk. On Mondays, take this disk to your off-site location – ie your work – and leave it in your desk. Take your other disk home that evening and hook it up. Repeat the swap every week. Your off-site backup may be up to a week old, but that's still better than nothing. This has, of course, the non-automatic element that you need to remember to do the swap!

Take it further

As discussed above, you're still limited in terms of the past data that you can keep. If you really need extensive backup history or to store larger files, you probably need to look into a tailor-made backup solution such as *Bacula* (**www.bacula.org**), and possibly into paying for hosting.

Hopefully, one of the methods we've looked at here has taken your fancy for your off-site backup needs. Just remember: the more often you can run backups, the better. And computers are much better at remembering things than humans are, just as they're way better at copying large quantities of data about the place. So, let the computer do the things it's good at, and you can get on with the things that humans are good at. Such as writing the Great British Novel, the Next Killer Web Application, or the ground-breaking critique of the blogosphere that will propel you to online stardom, safe in the knowledge that the fruits of your labours are secure. LXF

Subversion

The advantage of the Gmail method is that it means that you keep multiple versions of your files. Each email is separate and will be kept until you delete it (or run out of space). With *rsync*, files are overwritten, although not deleted. To delete files on the destination that have been deleted from the source you need to use the **--delete** flag. The problem with *rsync* overwriting files is that if you make a mistake and don't notice it until after your regular scheduled backup, it's too late. Yesterday's version is gone; the only version you have is today's.

One way to avoid this is to use a version control system such as *Subversion*. Version control systems mean that you have a record of all the changes made to your files. So if you screw up, you can just revert to an earlier version. Use *rsync* to back up your repository (your version control database), and you're good to go.

If you really don't want to do that, there are other options. One possibility would be to set up a *Cron* job on your off-site machine that runs just before the scheduled *rsync*, and copies your existing backup directory elsewhere. For example, this line in your crontab would do the job:

```
5 23 * * * cp -rf /home/user/backup /home/user/backup2
```

This runs at 11:05 pm daily. It would only keep one extra day's worth of data, but that might be enough for you. You write a more elaborate script to keep more days' worth of backups, but there's a penalty to that in terms of disk space. Every extra day of backups increases the disk space required by the same amount again – version control has a much smaller footprint.

Quick tip

Using a version control system is arguably a good idea anyway – you never know when you might want to undo a change!