O'REILLY
ONLamp.com
LAMP: THE OPEN SOURCE WEB PLATFORM

Published on **ONLamp.com** (http://www.onlamp.com/)
http://www.onlamp.com/pub/a/onlamp/2004/04/08/disaster_recovery.html
See this if you're having trouble printing code examples

Planning for Disaster Recovery on LAMP Systems

by Robert Jones
04/08/2004

I make my living building custom databases with web interfaces for biotechnology companies. These are MySQL and Perl CGI applications, running under Linux, and every one of them is different. Disaster recovery planning for these applications has consisted of routine tape backups of all the software and data, a bunch of `ReadMe` files, and having me around to put the pieces back together if something breaks—and things do break... power supplies, disk drives, RAID controllers, you name it. Recovery means we fix or replace the hardware, reinstall Linux, restore the apps from tape, and then stitch everything back together. Some recoveries have been easy. Others have involved pacing back and forth and swearing for hours on end while figuring out how the heck I had all this working in the first place. Not pretty, but that's just what you do, right?

That's fine in some situations but in larger companies, especially those with a formal "Corporate IT" group, this approach just doesn't cut it. There is a clash of cultures here that many of us have to face as our startup companies reach a certain size. All of a sudden we find ourselves spending way too much time in meetings, drawing up formal specifications and policies for this, that, and the other. Disaster recovery planning is one of the first of these efforts that most of us have had to deal with. Don't get me wrong, disaster recovery is a critical issue. It's just that it can be a very painful process for those of us who come from an informal development background.

I went through this last year when the CIO at one of my clients brought in outside consultants to formulate their disaster recovery plan. Their first step was to ask what the name of my executable was and where the installation script was located... OK... bit of a problem there. I have 54 Perl CGI scripts in one application alone, nine applications, and no installation scripts for any of them. Eventually, I understood what they really wanted to know, as opposed to what they asked in the questionnaire that they gave me to fill out. I viewed the process as an opportunity, rather than a hassle, and reviewed how I had my applications set up from their perspective. I had to make some changes to the software but, more importantly, I came up with an approach that I now use with all my projects to design in disaster recovery right from the start. I know I'm not the only one dealing with this issue so here are some ideas that you might want to build into your apps.

The (Configuration) Problem

The problem with our sort of database applications is that they weave themselves into the Linux system configuration. We add definition blocks to the configuration files for Apache, MySQL, Samba, et al. We create system-wide environment variables in user shells and insert symbolic links into the filesystem. Every time we rebuild a system we have to make the configuration changes anew. The potential for error is large, even assuming we remember all the steps.

On top of that, we need to deal with software dependencies. Perl modules are a godsend, but most of the sophisticated modules use of other module sextensively. When we use these, we inherit a hierarchy of dependencies that can make installation and recovery even more challenging and error prone. We don't want to give up the things that make our mode of development so productive, but we do need to understand and manage these dependencies. Our goals in designing for disaster recovery should be to keep things simple, to understand where dependencies exist, and to limit those where appropriate.

Separate the Application from the System

I place all my application software and data on its own partition, preferably on a separate disk. You want to be able to restore the application onto any suitable Linux system without regard for how that system is configured. Don't use `/var`, `/opt`, or `/usr/local`. You don't want your software anywhere near anything that the system or any other package might install.

My preference is to create a partition called `/proj` on its own disk with each application in its own directory under there.

With this layout I can take any suitable hardware and perform a standard install of whatever version of Linux is current. This step is really simple; anyone can do it and it lets you verify general system operation before we start with any of my applications. That is exactly what the disaster recovery people want to hear.

Only then do I create my partition and restore my software and data from tape. It is totally separate from any of the system software. This way the request for whomever looks after your backups is simply "Give me the latest copy of `/proj`", as opposed to "Give me this from `/usr/lib`, this from `/usr/local/bin`, and this from `/etc`" and so on. Again, the disaster recovery people really like that. Complexity means room for error, simplicity means success.

Use Perl Modules Wisely

The CPAN collection of Perl modules has incredible value, saving us from reinventing many wheels. But with every one that we use we add to the dependencies we need to manage. The more dependencies, the less robust the application. If a given module is not part of the standard distribution then ask yourself if you really need it. If the only reason to include `Date.pm`, for example, is to use a simple date conversion function, then think about writing your own version. I know that goes against the grain but in some cases, it can have a big impact on the complexity of your installation.

Archive Local Copies of Helper Applications

If you use third-party software within in your application, make sure to archive copies of the distribution kits for each package. For instance, I use ImageMagick for image manipulation and `gnuplot` to generate scatter plots in a gene expression application. In a recovery situation I don't want to hunt around the Net looking for the right tar files when I should be getting the database back up. Archive tar or RPM files for each package in your application directory, list them in the appropriate `ReadMe` file, and describe what they do and where they are used. You can then include that list directly in the recovery plan.

Archive Local Copies of Perl Modules

The same advice goes for Perl modules, but here things can get a bit messy. The problem is that many of the really useful modules, `Net.pm` for example, require other modules in order to function. So, you need to archive the whole tree of dependent modules. Figuring out what modules you need and whether they are part of the standard Perl distribution is not a simple task. The Module::CoreList can be useful in this regard.

Most of us deal with this by using the CPAN module to download and install our modules. When it works it is the best thing since sliced bread, but having it fail halfway through an install is not uncommon. Debugging the problem requires you to sift through the reams of output it generates and even then the fix is often not apparent. The most common advice found on the web is to run the program again and see if it works the second time around. Sorry, but that just isn't a good answer. In that case you are stuck with fetching the distribution kits for each module and building them by hand. See the `.cpan/build/` directory where CPAN.pm stores and builds modules it downloads.

Ideally, I would walk through the installation of everything I need for a project, fixing problems as I go. Then I would flush all of that out of the system and reinstall everything from scratch using the knowledge gained the first time around. In reality, I don't have time for that. So the best advice I can give is to record every step you take and then edit that log to produce the preferred set of steps that you will follow next time around. Be warned that this does not sit well with disaster recovery professionals. Explaining this in the plan will test your creative writing skills.

Edit the System Configuration Files

Note: My company is called Craic Computing and so you will see the word `craic` dotted throughout the following examples. It simply serves to distinguish my modifications from any system code.

The next step is to modify the system configuration files to suit your application. One major target is likely the Apache `httpd.conf` file. This is where you set up virtual hosts, link directories to web trees, take care of URL rewriting, and set any special options. The default file is already a beast, with around 1,500 lines, so we would prefer to keep all our application-

specific definitions in one place, preferably at the very end of the file.

The `Include` directive is the answer to our concerns. We can place all the application-specific definitions into a separate file and have that text included verbatim through the use of this reference.

```
Include /proj/linux_config/httpd/craic.conf
```

Better still, I can create a separate configuration file for each application, place all of these in the same directory, and refer to that directory with a single `Include` directive. (In this example, `craic_config` is that directory.)

```
Include /proj/linux_config/httpd/craic_config
```

In a similar fashion, we can define system-wide environment variables in application-specific files and include them in `/etc/profile` by sourcing a separate shell script file, or files, using this block of code:

```
CRAIC_DIR=/proj/linx_config/profile

for j in $CRAIC_DIR/*.sh; do
        if [ -r $j ]; then
                . $j
        fi
done
unset j
```

We can even use the same mechanism to set up Samba shares by including an external file in the main `smb.conf` file:

```
include = /proj/linux_config/samba/smb.conf
```

By limiting the changes made to system files to these simple statements we maintain our separation of system and application as much as possible.

Unfortunately, not all system files have an `include` feature. Of these, `crontab` is the most significant. In this case you can still store the configuration data under `/proj` but you will need to cut and paste it into the appropriate file as part of your recovery procedure.

Regardless of how you incorporate the information, you need some way of distinguishing your changes from the default configuration. My preference is to be explicit and to bracket any changes with comment lines like these:

```
## Craic modification BEGIN

[...]

## Craic modification END
```

This also gives me tags that I can look for, should I want to uninstall the changes or check whether they are already in place.

You should store all of the included files on the `/proj` partition. You could create a configuration directory for each application but my preference is for a single directory for all the application settings. In the examples above, this is `/proj/linux_config`. Within that I have subdirectories for `httpd`, `samba`, `mysql`, and so on. Bringing all the configuration data for all my applications together allows me to manage their interactions more readily than would separate directories. Additionally, I can refer to that single directory in the recovery plan, which is reassuring for our friends in IT.

The Installation Script

The disaster recovery plan people really, really want is installation scripts for your applications. It wraps up all the messy

details and it makes the recovery process look much more like the Windows applications they are used to.

By implementing the ideas given above, we can now give them what they want. All the script needs to do is insert the include directives, or blocks of code, into the system configuration files and restart the appropriate daemons. Make sure that your script tests whether the changes have already been made and then creates a backup copy of the system file before doing anything else. Here is an example block of code from an install script, written in `bash`, which inserts an include directive into the Apache configuration file.

```
HTTPDCONF=/etc/httpd/conf/httpd.conf

FLAG=`grep '## Craic modification BEGIN' $HTTPDCONF`

if [ -n "$FLAG" ] ; then
   echo '... no changes needed'

else
   cp $HTTPDCONF ${HTTPDCONF}.bak

   echo -e '## Craic modification BEGIN\n'  >> $HTTPDCONF
   echo "Include /proj/linux_config/httpd " >> $HTTPDCONF
   echo -e '\n## Craic modification END'    >> $HTTPDCONF

   /etc/rc.d/init.d/httpd restart
fi
```

Using comments to delimit each block makes it easy to excise the modification as part of an uninstall script. This is extremely useful during the testing phase of your disaster recovery plan.

Why not just create an RPM or other package for each application? That would be great but it involves quite a bit of effort for an application that will only ever be installed at a single site. To my mind, a simple shell script is at the right level of complexity. Should you choose to go the extra mile and create an RPM installation package, then the ideas described here should serve as a good foundation.

Documentation

We all strive for good documentation but, if we are honest, we don't usually do a very good job of it, especially when our applications are still under active development. So how much documentation is enough in the context of disaster recovery?

Our friends in corporate IT want to know what an application does, where the software and data for it resides, what other software it depends on, and exactly what to do to rebuild it. They want this in a form they can print out and put in a binder on the shelf in the computer room, along with a copy in off-site storage.

I need something different. I want to see `ReadMe` files scattered throughout the directories that tell me what the files represent, what the scripts do, and what other parts of the application and system they interact with. These make up the informal map that another developer can use to decipher my work if I get hit by a bus. Perhaps more importantly, they will refresh my memory three years from now when I need to make some changes.

Don't be shy about what you put in the `ReadMe` files. If part of the application is a complete hack that depends on an obsolete Perl module, or if you know it will crash next time we have a leap year, then say so. You don't need to tell the world about the gory details but you do need to capture that information in a form that you or another developer can access. In the midst of a real disaster recovery, those notes can make all the difference to someone trying to fix things.

Please remember, ALWAYS put a date and a name next to your comments. A problem that required a huge workaround last year might well have been fixed in the current release of the operating system. This is a widespread problem with Linux HOWTOs and web sites. A date helps me assess whether the information is still relevant and a name gives me someone to contact if I need more information.

If you can create and maintain this level of documentation as you develop the application then it is not too much effort to

rework it into the form that the good people in corporate IT are looking for.

Use DNS Aliases for Multiple Applications

If you have several applications and multiple servers then you should consider setting up Apache virtual hosts for each of them along with DNS aliases that relate each application to the physical host.

For example, let us say I have two machines (server1 and server2) with the application `app1` on server1 and `app2` on server2. The default way to access the start page for each application would be to use the URLs `http://server1/app1` and `http://server2/app2`.

If server1 blows up then I either need to replace it or move the `app1` application to server2. But then all my users will have to update their bookmarks to point to the new machine. The better alternative is to create the hosts `app1` and `app2` as DNS aliases that point to server1 and server2 respectively. In the Apache config for each server I create virtual hosts for ALL of these applications, in essence replicating their configuration even though the application itself may not be present. Users now access the applications as `http://app1` and `http2://app2`. The IP address in the DNS alias dictates which machine the user is directed to.

If I need to move either application to another machine I simply install the software, set up the virtual host, and then change the DNS alias to point to the new machine. All the existing bookmarks and links continue to work. Users are none the wiser to the change in venue.

Mirrored Servers

Live replication of applications and data is a great way to ensure that your applications are available. Rsync lets you maintain duplicate copies of directories on different machines, with regular updates. The directory layout I've discussed here fits in perfectly with rsync's abilities. MySQL replication can take that one step further with live mirroring of the contents of a database to another machine. The setup is more involved than `rsync` but it can be well worth the effort.

Be careful not to confuse high availability with disaster recovery. Mirrored servers will do a great job at replicating your data, good or bad. Replication can easily result in two corrupt databases instead of one. Things are different with the high-end commercial databases that you'll find in the banks, but that's not what we dealing with here. On our level, replication is great but nothing beats having a tape on a shelf in off-site storage.

Final Thoughts

Disaster recovery planning should be just as important to developers as it is to corporate IT. While the cultural differences between "us" and "them" can be frustrating, we need to address their needs head-on if our style of application is going to find a place in their world.

By designing our apps with disaster recovery in mind right from the start, we become an ally of corporate IT rather than a thorn in their side. A little bit of forethought pays big dividends.

Robert Jones runs Craic Computing, a small bioinformatics company in Seattle, which provides advanced software and data analysis services to the biotechnology industry. He was a bench molecular biologist for many years before programming got the better of him.

Return to ONLamp.com.