Working together and sharing code with TLA

TLA as your Version Control System

Carlo Contavalli

If you ever worked on a free software project or if you have ever worked as a developer, you probably know that managing source code, patches, and software release cycles is not the easiest task to perform. Things get even worse if lots of people are working on the same project: more code to manage, more people to coordinate, more patches to integrate and mainstream. Even if you don't write software or have never worked on such projects, I'm sure that as an addicted computer user sometimes you felt like "hey, why didn't I make a backup of that document", or "hell... I liked it better before" or "wow... why was that change introduced? I can't remember it...". In this article, I won't talk about the importance of backups, which should already be clear in every reader's mind, but I will introduce a new and very powerful system useful for managing source trees, tracking down changes, and allowing many people or groups to work together or independently on the same projects: TLA, often referred as GNU Arch, a new generation Version Control System (VCS).

Version Control Systems and your happiness

If you don't know about VCS's or SCM's (Software Configuration Managers), just close your eyes (maybe at the end of the paragraph) and start imaging a big software project: lots of developers, lots of source files, roadmaps to inspect closely, bugs to find and fix, patches to be released while new features are being introduced, developed and tested. And yes, all these things being done by different people, for different reasons as time goes on.

As soon as a project starts to grow and spread among your customers or users, it becomes very important to have the ability to track:

- When and why a particular change was introduced?
- Which version of the software each customer/user has installed?
- Which changes were introduced in each version?
- Who it was that actually introduced a given change (often, a given bug)?
- Where a version of the software, which is still in use and which needs a minor change, can be found?

This is where Version Control Systems come to rescue. The purpose of a VCS is to prevent a given project, whether it be a software project or anything else that can be saved as a tree of files on your hard disk, from becoming a nightmare. Once in use, it will take care of:

- Remembering which files a project or a particular version of your project is made of
- Remembering changes that made to any file, allowing you to "undo" those changes and to know who introduced them and why
- Conflicts – preventing different people introducing "incompatible" changes to the same files without knowing
- Introducing automatisms into your software release cycles, by providing "hooks"

- Allowing you to continuously develop two or more versions of a project (often called branches), easing the task of exchanging and sharing code among them
- Easing the task of integrating changes (patches) provided by third parties

Generally speaking, a VCS could be considered some sort of oracle which knows everything about each and every file of your project, which you can ask questions or delegate tasks to (like, "please, apply this patch to the whole project… or please, show me that file as it was 20 days ago"). Once you start using VCS's you will become addicted, and will start using them for everything, even for simple documents or articles (like the article you are reading, and no, I'm not one of the most addicted).

A VCS could be considered some sort of oracle which knows everything about each and every file of your project

TLA as your Version Control System

By now, you should be wondering why I felt such a strong need to talk about TLA in this article (no, the "Free Software Magazine" editor didn't force me, and no, the upstream authors didn't pay me), when so many VCS's are available on the internet.

First of all, TLA is one of the few free (as in free speech) VCS's available that comes with a decentralized development model in mind: it supports multiple archives, possibly managed by different people, created for the same project. It also provides support for archives to import/export patches and pieces of code from each other, without loosing track of who made them and where they came from.

Second, TLA is very powerful in what it does, still being extremely simple, both in its usage and in the way it works. For example:

- An archive is a simple hierarchy of directories, each one containing a .tar.gz with either the sources of the project or some sort of "diff" file and a couple of text files used internally by TLA (usually, the log entry and the checksums of the files). Other VCS's use Berkeley DB databases, SQL backends, proprietary database formats, etc.
- It doesn't need a dedicated server. TLA can store, access and write archives using standard protocols like SFTP, FTP or HTTP (using WebDAV for write support).
- TLA supports symlinks, renaming both directories and files, keeps track of privileges and file system permissions
- It supports GPG signed archives and operations, and more.

Choosing the right VCS is a matter of looking at the development model being used. Obviously, TLA doesn't fit well with every development model. While there are limits that only TLA is able to overcome and models that only TLA fits, there are other VCS's able to overcome some of TLA limits and that fit better with other development models. Be careful before taking any decision in any direction.

Choosing the right VCS is a matter of looking at the development model being used

Starting up with TLA

Without going into the details of the various commands, details you can find in any of the many TLA tutorials, let's see what you need to do and how things work with TLA.
The first time you install TLA, you will need:

- To introduce yourself to TLA, by stating your name, email address, and the address that will be used by TLA to generate the changelogs
- To create your own archive or repository, a place where TLA will store all of your projects, or trees, and all of its own data
- To import one of your projects into TLA, and start working on it

Well, to introduce yourself to TLA, there's not much to say, just run something like:

```
$ tla my-id \
 'Carlo Contavalli <ccontavalli@masobit.net>'
```

Version Control Systems and the Linux Kernel

Almost every project above a certain size is managed using a Version Control System (often called a Software Configuration Manager, or SCM, even if the two terms have slightly different meanings). One such project, is the Linux Kernel.
As many of you probably know, the sources of the Linux Kernel are managed using a proprietary and closed source product, BitKeeper (http://www.bitkeeper.com).
Although this product has proven its reliability and usefulness over the course of several years, many have pointed out that its license, usage conditions and the fact it is closed source makes it incompatible with the development model behind Linux and the philosophy behind free software. These people suggest that the Linux Kernel development project should switch to using a Version Control System completely based on free software, and not rely on the free (as in free beer) availability of a closed source proprietary product.
This issue has been discussed in many occasions (flame wars?) on different mailing lists in the past. Linus Torvalds has explained on the Linux Kernel Mailing List (http://www.tux.org/lkml/) in 2002:

> "Would I prefer to use a tool that didn't have any restrictions on it for kernel maintenance? Yes. But since no such tool exists, and since I'm personally not very interested in writing one, and since I don't have any hangups about using the right tool for the job, I use BitKeeper… "

His position has probably given an additional burst to the development of new, more powerful, and free (as in free speech) VCS's, although the Linux Kernel sources are still (at time of writing) managed using BitKeeper.
On April 6, 2005, just a couple weeks ago, Linus Torvalds finally announced that due to a conflict over BitKeeper usage, the kernel team was looking for alternatives.
At the time this article was written, no choice was yet made upon which SCM to use, but candidates discussed on the mailing list were mainly monotone (http://www.venge.net/monotone/) and bazaar-ng (http://www.bazaar-ng.org/), based on the GNU Arch system.

Creating a new archive is a bit more work. You need to choose a name for it, some sort of label and decide where to store it. Choosing the name for your archive is probably the most important step: as each archive can be mirrored in different locations, or accessed using different methods at the same time (like HTTP, FTP, etc.), its name is the only piece of data that allows TLA to distinguish one archive from one another. It is also very important for any two different archives to have different names: a distributed VCS is able to work on code and trees kept in different archives, and if the archives involved in a project happen to have the same name, big problems can occur.

To avoid this kind of trouble, TLA enforces some policies over archive naming: an archive name must be made of an email address, followed by a "–" and by an arbitrary name, chosen by the user. As an example, a valid archive name could be: "ccontavalli@masobit.net–public".

As you may imagine from the previous paragraphs, the email address is there to guarantee some sort of uniqueness in archive naming. Without it, there would probably be thousands of archives named just "public" or "private", and TLA users would have a lot of trouble dealing with them. Obviously, you should use one of the email addresses you are the owner of, and if you want to keep using TLA without difficulty, you shouldn't have two archives with the same name.

This and many other conventions enforced by TLA have been the origin of many a flame war. If you don't like putting your email address in the name of an archive, just create a mailbox for it, or, like I've done many times for public projects, point it to a mailing list. After all, it's just a label.

Before talking about how to actually create an archive, there is another convention that TLA gurus suggest to use in nam-

ing archives: they suggest to put some sort of number in the archive name, like the year of when the archive was created. Archives tend to become pretty large, even if a very efficient storage method is being used. By putting some sort of number in the archive name, you have the freedom to close or "seal" that archive up when it starts to become too big, and to create a new archive with the same name (but a different number) where the development can easily continue.

Now that you know everything about archive naming, just create one, using something like:

```
$ tla make-archive \
 ccontavalli@masobit.net--2004-public /home/tla
```

Where `/home/tla` is the directory where you want your archive to be stored. If you wanted to create it on a remote server, you could have used something like:

```
$ tla make-archive \
 ccontavalli@masobit.net--2004-public \
 sftp://ccontavalli@a.server.net/home/tla
```

Where "sftp://" could be replaced by "ftp://" or "http://", with no need to have shell access to the remote end.

Always keep in mind that anywhere you keep your archives, privileges must be enforced by the underlying method: the file system, the FTP server or the web server. Make sure that you setup privileges correctly before opening up a TLA archive.

Now that you have your own archive, you can import any project you may want into TLA (or start a new one). Again, the hardest part of the operation is choosing a proper name for the project.

This time, TLA requires you to assign a name made of three parts, divided by "–", like "coolplayer–main–0.0". In this example:

- Coolplayer is the name of the project, it can be almost everything you like, with some restrictions on the characters being used.
- Main is the line of development for coolplayer.
- 0.0 is the version of coolplayer being kept in that archive.

There's not much to say regarding the name of the project: just choose one. It's a bit harder to talk about the name of the development line and the version number, so I'll start with a simple example.

Let's say you're one of the "coolplayer" developers, and the currently released version of coolplayer is "0.0.2". As you work, you're probably adding features to version "0.0.2", and preparing to release version "0.0.3". So, you are working on improving version "0.0". At some time you may start introducing big structural changes, getting ready to release version "0.1.x". If you work on those changes directly into the "0.0" branch, you would probably end up in making the code unusable until your work is completed, blocking bug corrections and all the other developers still working on "0.0". In this case, a good idea would be to create a project called "coolplayer–main–0.1", starting from a particular version of "coolplayer–main–0.0", and adding all of the changes planned for version "0.1" there, still allowing "0.0" to be developed and released without changing the release procedures.

TLA would allow the two branches to import/export patches from each other, keeping track of what has already been done in each of them.

Now, let's say your company is selling "coolplayer" to some of its customers and one these customers wants to buy version 0.0 but with some slight changes. In this case, you could just add those changes to the main version of "coolplayer", but perhaps you don't want to or you are not allowed to. A better solution would be to create a new line of development called "coolplayer–customer1–0.0", derived from "coolplayer–main–0.0", then the changes needed can be introduced while still keeping your boss happy. If a bug fix is introduced in "coolplayer–main–0.0", you can use TLA features to import the bug correction into the line "coolplayer–customer1–0.0", without losing any of your previous changes. TLA will handle this.

By now, you might be wondering what the "main" stands for. You're free to follow any convention you may like; just be sure you do have a convention. Usually any given software project has one "main" line of development which is used to add new features and to prepare for new releases.

In TLA, branches are natural, there's no magic behind them

On other VCS systems, lines of development are called "branches". In TLA, branches are natural, there's no magic behind them. The assumption is that each and every project

will always have multiple lines of development, and could be "derived" from previous projects. TLA always forces you to give it a name.
There is one more thing to be said about branch names: once a version of "coolplayer" is released, we need to tell TLA "here in the archive, you have a version of coolplayer that I want to remember, please save it into... " - where that "into" is usually the name of another archive. So, for coolplayer 0.0, you may end up having the following branches:

- coolplayer–main–0.0: where all the development is made
- coolplayer–release–0.0: where all the coolplayer public releases are stored (the "into" mentioned above)
- coolplayer–customer1–0.0: where the coolplayer with changes for customer1 are kept

On other VCS systems, this operation is called "tagging". On TLA, tagging just means "store this version of the current project in some other branch", which is by all means identical to any other branch. You can then work on any of the above branches as you wish, and easily import/export changes from one branch to another without issue, using standard TLA features.
Now that we've chosen the name for your "branch", or project, let's create it:

```
$ tla archive-setup \
 -A ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

There's one small problem: the branch is completely empty, no files have been put into the archive, so you still can't do much.
To start using TLA, just go into the directory where you started working on your project (or an empty directory), make a backup (just to be safe), and run:

```
$ tla init-tree \
 -A ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

This tells TLA the directory contains all of the sources for the coolplayer project, which needs to be stored in the 2004-public archive. Don't get scared by '{arch}' or '.arch-ids' directory that TLA will create in your source tree. You now need to tell TLA which files and directory you actually want it to keep track of, and which files you want to keep in the TLA archive.
To do so, just run:

```
$ tla add file_or_directory_name
```

After you have added all of the files, you can run:

```
$ tla tree-lint
```

to verify you haven't forgotten anything. Note that tree-lint will output warnings regarding files TLA knows nothing about, and errors regarding files which violate "naming conventions". You can slightly tune "naming conventions" by editing the configuration file '{arch}/=tagging-method', created by TLA in your project directory.
As with any VCS, it is a little tedious keeping it informed about which files are part of the project, considering that every rename or remove should be reported using some sort of command (with TLA, tla rm, tla mv, etc.).
Thankfully, TLA utilities comes to the rescue again: you can easily find scripts like "tla-update-ids", which tells TLA about any new files, and files which have been removed, etc. by reading the output of "tla tree-lint".
Finally, you can tell TLA to upload your project into the archive:

```
$ tla import
```

Working with TLA

You can finally start working on your project as you wish. You have it on your hard drive, so you can just work as usual by editing your local copy of the files.
When you are done, and you have a set of changes you want to be saved on the archive and make them available to other developers, just run:

```
$ tla commit
```

If you set the "EDITOR" environment variable up correctly, your editor will pop up asking you for an entry to be added to the TLA log; this entry should list the reason for the changes.
Otherwise, you will have to edit the file named "++something" in your source tree manually and then run "tla commit" again.
While committing, if some other developer made changes to the project, you will receive a warning and the commit will be automatically aborted. You will then have to manually choose what to do. Probably, you would run:

TLA and naming conventions

TLA is quite a nice tool. However users often feel that the naming conventions enforced by it are tedious and wrong.
Directory or file names which contain characters like "{", ",,", "++", "=" are used by default by TLA utilities, and are sometimes hard to handle.
Commands like "archive-setup" or "make-archive" that perform completely different tasks on "objects" they both call "archives" are not very easy to use.
TLA is evolving, and the naming conventions are being changed for the better as time goes by. As an example, in version 1.3 a "delete-id" operation has been added as an alias for "delete", which will probably disappear in a couple versions.
Other users feel that TLA is too restrictive in naming archives or projects. You may get frustrated with this in the beginning, but after a while you'll probably come to like it, and start feeling that this strictness is necessary.
If you like TLA but not its commands nor its naming conventions, you may want to try "bazaar", a fork of TLA, with cleaner and neater commands and naming conventions.

```
$ tla update
```

to update your local copy of the tree with the changes introduced by the other developers, without loosing any of your own local changes.
If something goes wrong and one of your local changes can't be merged automatically with those introduced by other developers, a .rej file is created, containing the code that caused the problem, exactly as the "diff" and "patch" command would (guess who created that file?).
If you'd ever like to have another copy of the project on your computer or any other computer, you need to perform two steps:

- tell TLA how the archive, as you named it, can be reached
- get a copy of the branch you are interested in

As an example, if I wanted the sources of "coolplayer–main–0.1", I'd need to run something like:

```
$ tla register-archive \
 ccontavalli@masobit.net--2004-public \
   sftp://ccontavalli@a.server.net/home/tla
```

to tell TLA that the archive named "ccontavalli@masobit.net–2004-public" is accessible from my machine by connecting with the SFTP protocol to a.server.net, and:

```
$ tla get -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

to actually get the project into the current working directory. Take note that "tla get" as executed above will fetch the latest version of the project. If you want to fetch a previous version, you just need to know that every commit into a tree has a name, usually "patch-x", where "x" is incremented on each commit, while each import operation has a name like "base-x". So, if you want to fetch your sources as they where imported in your TLA archive, you just need to run something like:

```
$ tla get -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0--base-0
```

As with any VCS system, you also have lot of commands that allow you to browse the content of a given archive or the state of a project. TLA provides for example:

- logs - to have a list of commits performed on a given project, possibly with a small summary of why they were performed (–summary)
- abrowse - to see the list of the projects in a given archive
- missing - to have a list of missing "commits" on your current tree
- and many, many others

Up to now, I have talked about "tagging" but haven't shown how it's done. Suppose that a particular version of one of your projects is ready to be released and you want TLA to remember that version as one of the released versions. To do so, you can quite easily ask TLA to save it in a different branch, for example, in the "release" branch, just to choose one:

```
$ tla tag -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0 \
 coolplayer--release--0.0
```

This way, the latest version of coolplayer in the main branch for version 0.0 is saved in the release branch.

So far, I've only shown TLA being used as I would have used any other centralized VCS system: one repository, several lines of development, developers that get and commit from that single archive.

The "tag" command is at the base of distributed development, since it allows a particular version of a given project to be saved in any other archive, while keeping track of its origin and allowing others to work on that archive:

```
$ tla tag \
  c@m.net--2004-public/coolplayer--main--0.0 \
  s@a.net--private/coolplayer--myown--0.0
```

[in the above example, `ccontavalli@masobit.net` and `somebodyelse@another.net` had to be changed in `c@m.net` and `s@a.net` for typographic needs. Please do not take this as an excuse not to use tla: even if its command lines may sometime get lengthy, tla is extremely useful and powerful and a properly configured environment can be of much help.] After executing the above command, somebodyelse@another.com could work on coolplayer from their own archive. If something good comes out, both projects can fetch changes from each other by using commands like "tla star-merge", "tla replay" and so on, commands that are beyond the scope of this article.

Conclusion

TLA is a complete and extremely powerful VCS. There are lot of utilities available, starting from web interfaces, up to automatic update tools (like tla-update-ids).

TLA also provides many extremely interesting features that were not even introduced in this article, like transparent support for archive mirroring, GPG signed archive support, client-side hooks and advanced merging features.

For those concerned by the long command lines being used, a properly configured environment can be of great help, and a lot more can be taken out of TLA:

- "tla my-default-archive" allows you to tell TLA which archive to use if no -A is used.
- Bash completion for TLA archive names is provided by external scripts allows you to use TAB-TAB to complete your command lines.
- "tla make-log" allows you to write your changelog while working, without having to remember all changes until commit time.
- And there's a lot that can be said about multi-tree projects, revision libraries, and changesets.

The main shortcoming in TLA that I have encountered so far is caused by the lack of server-side hooks. As TLA uses standard protocols, it's not possible to tell TLA to run a given command on the server every time a commit is performed. Also, if a client crashes due to networking problems during an operation on the archive, the archive itself may be left in an undefined state.

Luckily, the simple storage used by TLA can be manually fixed in just a few minutes, and dnotify or inotify support in the Linux Kernel allows simple hooks to be called on the server when needed, even if no direct support is provided by TLA. And, if you really want to, you can always try "arch-pqm".

Copyright information

About the author

Carlo Contavalli works as a full time developer for Masobit Corp. (http://www.masobit.net). He also is an addicted free software activist. He has contributed to projects like PigeonAir (http://www.pigeonair.net), mod-xslt (http://www.mod-xslt2.com), Debian GNU (http://www.debian.org), and many others, by writing code, submitting patches and writing documentation. He has also organized or taken part in various free software related events.