Universidad Tecnológica Nacional, Facultad Regional Buenos Aires

Distributed Version Control Systems

Demian Alonso†, Bruno Bonnano‡, Nicolás Calligaro†† and Nicolás Perez Santoro‡‡

† *demianalonso@gmail.com*
‡ *bbonanno@gmail.com*
†† *nicolascalligaro@gmail.com*
‡‡ *nicolas.perez.santoro@gmail.com*

***Abstract*— Estas instrucciones se presentan para ayudar a los autores a preparar su trabajo en el formato definitivo correspondiente a la materia Arquitecturas de Proyectos de IT. El resumen no debe exceder las 200 palabras.**

***Keywords*— Por favor, incluya hasta cinco palabras clave que identifiquen en forma concreta su trabajo.**

I. INTRODUCTION

Most software projects use Version Control Systems (VCS) to manage revisions of source code. Version Control Systems traditionally track the whole history of the source code in a central server that everyone that can access. Users need to access this central server to work with their VCS. A modern, different approach of doing revision control is using a *Distributed* Version Control System (DVCS). In these systems, there is no central server that contains the history and tracks changes, instead everyone has a copy of the full repository with the complete history. Given the lack of a centralized server they are also called De-centralized Version Control Systems.

In this paper we're going explain what DVCS are, based on the distributed system Git, and compare them to centralized Version Control Systems, highlighting the differences, advantages and disadvantages of both approachs in various contexts, in a both a conceptual level, and a practical level of actual VCS; ultimately evaluating when to use or not Distributed Version Control Systems.

II. DVCS HISTORY

The concept of distributed repositories came up first on the commercial product Sun TeamWare (*good citation needed*), designed by Larry McVoy who later went on to design BitKeeper, another commercial VCS which expanded on those ideas. BitKeeper was used from 2002 to 2005 to manage the Linux kernel source until the license to use BitKeeper was finished. To replace the use of BitKeeper in the Linux kernel, two projects started, Git (by Linus Torvalds [1] and Mercurial, both now mature DVCS. Other open source DVCS in use today are Bazaar, developed by Canonical Ltd. and used to maintain the Ubuntu codebase, and Darcs, based on the idea of tracking only patches and not versions.

For reference, in this paper we'll base our explanations on Git but for most features there are no big differences between most of these DVCS, especially between Git and Mercurial which are largely similar, with most commands having the same names in both tools.

III. ARCHITECTURE

A DVCS has many features that makes it perfect for distributed and non-distributed developments. Almost every feature is based on the fact that DVCS is distributed instead of centralized and therefore if you work distributed then merges would happen on a regular basis and they should be as fast and easy as possible, providing developers with a whole new set of tools and relieve them almost completely from having to code according to the rules required to work with a centralized CVS like system.

A. Distributed

The idea behind being distributed means that no server has the full content of a repository but rather a clone of it with the changes that a particular developer has commited. This allows users to have their own clone on their own computer and work in a offline environment while being able to version their changes. It also makes the whole versioning system faster.

B. Change-sets

You could also create several other clones for every feature or bug that they want to develop and push those changes to another clone, a peer's repository for instance, and pull some changes from another peer to your local clone. To able to do this, a DVCS tracks change-sets instead of changes in individual files so that it can keep track of all the changes needed for a feature/bug to work.

C. Patchs

The idea of having several clones, one for each feature or bug might be problematic so instead of doing that, you could have all your changes on a single clone and then select which one of those changes you want to commit. You can even select which parts of a single

file you want to commit and which ones you don't, then group those changes and commit them in a very atomic way.

D. Flexibility

If you need a centralized repository you could still use a DVCS and all it's features while pushing all the changes to a particular repository. This shows how flexible a DVCS is, to the point of being able to describe it as a super-set of a centralized CVS although in really it is not.

E. Trust Based Security

Security on DVCS is based on the "circle of trust" model which implies that you trust only on a very specific group of people and you only pull from those people which in turn trust in their own group of people and only pull from that other group of people. This creates a chain of trust so that even if you don't trust some one directly by scalating their changes you could end up having his code only after it has been reviewed by the people you really trust.

F. Small, Coherent and Logical Commits

Due to the fact that you can select very specific pieces of your code to build a commit you can make commits very small and atomic making the merging process more easy by selecting only those commits that won't break your code. This technique of only selecting the changes that you want is called "cherry-picking" and it has the advantage of using the revision tree as a series of *changesets* or *patches* instead of just a series of static versions. The distributed model enhances this by allowing to make commits whenever you want to your local copy without breaking anyone else's code. Many little commits can be grouped as a bigger, conceptual commit in the case that you need those changes to occur at the same time and not individually.

G. Tagging and Branching

Most DVCS allows the user to attach an arbitrary description the a tag so you can store the whole release announcements along with the tag of that version. Moreover, as with the commit, the identity of the user who made the tag is stored and can be securely verified by a cryptographical PGP signature that is embedded on the tag

H. Merging

The merging process has two possible scenarios: *Fast-forward merge:* The changes has been applied to only one of the branches and therefore those changes are replayed to the other branch automatically. *Three-way merge:* When changes exists on both branches, the DVCS tool will try to automatically merge them and if any conflict arises then it reports it to you and let you resolve them. The merging process[3] between branches preserves all the history of both branches on many DVCS tools.

I. Pushing and pulling

When we want to share our work with the rest of the team we have two basic commands, *push* and *pull*.

J. Pushing

The push process refers to taking all the changes previously selected and adding them to the repository of another peer. In practice we usually don't do this because we shouldn't push changes to a peer's working copy of the repository so the push process is usually done agains a "central repository" where all the team make their pushes too. This logic is similar to centralized VCS, but we must keep in mind that we could have a lot of "central repositories" for different parts of our project or even for different parts of the team. Testing could have its own central repository, while development has another copy of it and changes could be pushed among them very easily.

K. Pulling

Pull is the opposite action of push and it is the combination of two other commands, "fetch" and "merge". The fetch command takes the changes from a peer's repository and copies them to the local repository. After that, the merge process combines the changes you've just downloaded with the ones on your local repository. You can do those two actions separately or altogether by using the "pull" command.

IV. Comparison

There are several DVCS in the market nowadays and most of them are open, free or have a free license while not being used in comercial environments. While most of them have very similar features, the one that has the greatest number if features is Git with over 150 commands out-of-the-box but at the same time it is the one with the steepest learning curve because it is very different from SVN, which is probably the most widely used VCS. One key aspect of VCS is its performance. It was said previously in this document that the merging process was crucial for a DVCS because it's something that you'll probably have to do several times during a day in a distributed environment and therefore it should be as fast as possible. This is not the case for all DVCS tools. In the following charts[4] we'll compare some of the most widely used DVCS.

This benchmark simulates a linearly growing repository, being sequentially filled with 4000 files. Each file is modified five times. At the end of the test, the repository had 24,000 revisions.

As you can see in the chart above, the time that takes to do a check-in is not constant and differs a lot from one DVCS to another. While Git takes almost nothing to do it no matter the size of the repository, others like darcs take up to 5 seconds.

If you verify the size of the resulting repository on the previous simulation you would find the following:

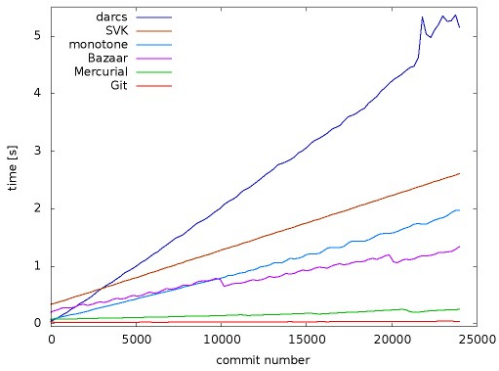

Figure 1: Check-in Dependence on Repository Growth

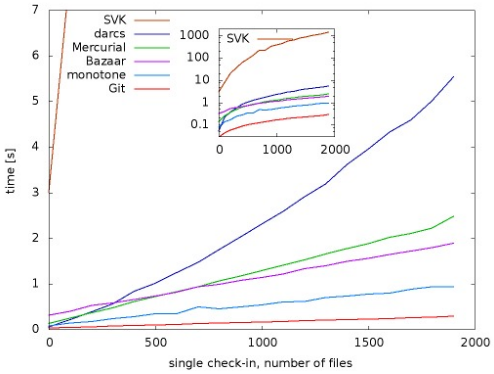

Figure 3: Dependance on Check-in Size

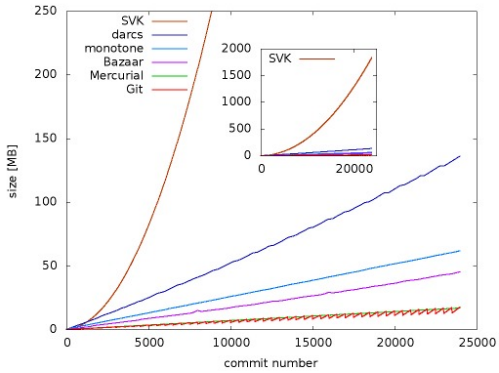

Figure 2: Repository Size Growth

Doing the math you will see that the resulting checkout is about 15MBs. SVK needs almost 1.8 GB to store that repository, while the rest of them needs less that 150MB. You might notice that Git has a "saw tooth" shape. That's because Git regulary optimizes its repository.

The following benchmarks tested how much time it took to add a certain number of files to an empty repository.

Once again, Git is the fastest, taking only 300ms to add 2000 files, while SVK needed almost 23 minutes to add the same amount of files.

A. Commits

In a DVCS the nature of the commits is different to centralized VCS, since commits are made in the local repository only. Commiting in centralized VCS means "to share the change with everyone that queries the centralized server" while in a DVCS means to store a change in the history, which is not shared until it's pushed or pulled into another repository.

Also, DVCS allow to apply a technique called "cherry-picking", which consists of "picking" what changes you want to use in a version, and build a new version containing only those changes. This way the revision tree is seen as a series of *changesets* or *patches* instead of just a series of static versions.

B. Merging

Merging on Git between branches is greatly improved over SVN, because all the history of both branches is preserved over the merges.

C. Sharing changes

If you work simply pulling and pushing against a central repository, those operations do look very much like the operations idea of check out and check in in a centralized VCS, but that's the only way of sharing changes in a centralized VCS. In a DVCS, since every repository is a peer to each other, anyone can share their repository in read-only mode (via ssh, http, or simply sharing the directory) to let other people pull from them.

D. Repository Size

Centralized VCS also have their advantages, you don't need to have the whole history in your personal repository so you can save space, for example in Perforce you can check out only a part of the repository[2] while in Git repositories can't handle millions of file without getting a performance hit on some operations[1].

E. Tool support

Systems like SVN have mature tool support and various plugins, but popular DVCS are catching up. For these tools most complete interface is the command line, but for the usual operations, there are various visual tools to operate with the repository.

Git, in particular, is based on Linux and originally didn't work work out of the box in Windows, but with Cygwin. Now there is a fork of Git that ports it to windows, but the linux heritage is shown and the installation includes a Bash shell. Git also includes a portable GUI written in Tk, invoked with command "git gui" but while it is simple and easy to use, it is very incomplete (for example, it doesn't include the "pull" operation, among others).

There are many other third-party GUIs for Git, but none of them has all available operations nor are them equally mature, in those cases, the command line is the only way to do them. One of the most complete GUIs is SmartGit, but requires to purchase a (rather inexpensive) licence for commercial usage. An alternative for Windows only is Tortoise-Git, which is a fork of Tortoise-SVN that shares the same look-and-feel, while supporting most of the Git operations. There are others GUIs but they are either very uncomplete or aren't really mature and have some kind of bugs (like git-cola) [5].

There are some tools to interoperate with others VCS, easing transitions. Git in particular includes has git-svn which allows to push and pull from svn repositories.

V. POSSIBLE WORKFLOWS

As mentioned before, this tool allows for differents workflows that just weren't possible before.

Changes can be just shared between peers, without needing to access the central repository. Branches can be created instantly, and merging them is easier since the whole history of changesets is preserved instead of just taking the diff of the versions.

Push and pull are much more versatile than checking in and out, since you can push changes into many repositories and pull from many others, not just a central one. In fact, there is no need to push at all: everyone could be just pulling changes from each other. For example, two co-workers could work in a feature, pulling from each other, then let a third one fetch them and do a code review, and then discard it or merge it. A fourth could fetch all reviewed changes into a test version, to do an acceptance test, and then merge the changes into a final version.

A. Peer code review

If doing code review is a common practice, then using a DVCS can aid to this process. Since changes can be shared among any workstation, the reviewer can fetch the changes from the reviewee's workstation. The reviewer might be working on some other feature and may have it's workspace in an unstable state. But this is not a problem since the reviewer can just clone the reviewee's repository and can have it sepparatedly from its current changes. This also prevented them from sharing the code through a public repository, which can present some disadvantages like a third developer using code that is not yet intended for general use.

After the review, they can push changes to some public repository for everyone to use, or the reviewer can push changes to the reviewee repository for further fixes. All of this is completely isolated from the rest of the team and with no commit history loose.

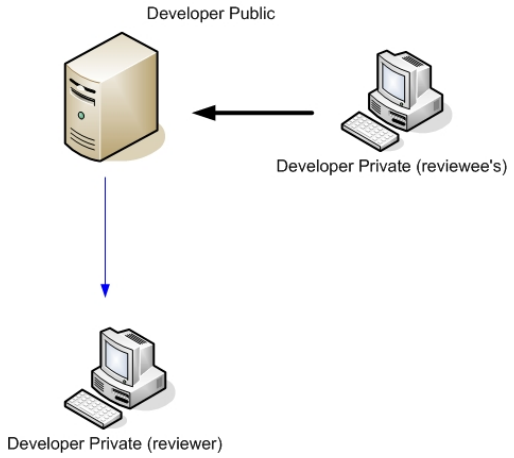

Figure 4: Peer Code Review workflow

B. Centralized Server

The classic workflow where there is one central server which has the whole project is also available. In this case all developers has access to the server and can pull and push changes from it. The main advantages is that any developer can share changes among other developer without needing to use the central server. However once a piece of code is considered finished it should be pushed to this server.

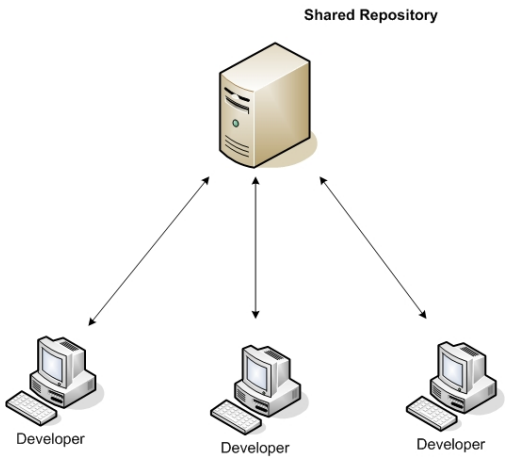

Figure 5: Centralized Server workflow

C. Independent subteams

If the application being developed requires a large amount of people working at the same time, it is very likely that they are working on different subsystems. Each subteam can be working on a different repository, and so the chances of one team messing with the other teams build is completely removed. This helps each team to mantain their own build and if a problem

on the build arises, then it is something to be solved between a reduced group of people.

Then both repositories can be merged togheter into an integration one by pulling from each of them. If there are conflicts which cannot be resolved easily, it is possible to ask the team who caused the conflict to solve it since all history is preserved. Then this team can clone the integration repository and merge it with its working repository. They can ask the person in charge to pull the changes back again.

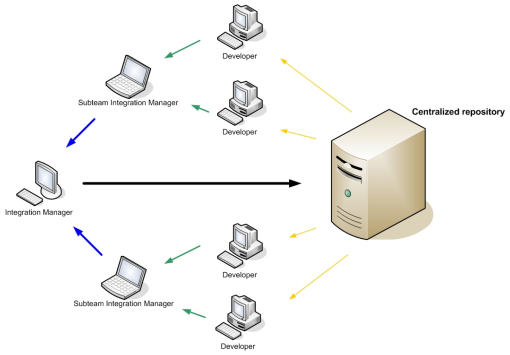

Figure 6: Independent Subteams workflow

D. Centralized repository with integration manager

It is possible to have a single central repository where all developers have read access. Everyone has an official public version from which they can work from. Then, to add new change sets to the official version, an integration manager can pull the changes from each developer (or each team) to have all the changes tested and reviewed. When the build is good enough the integration manager can push the changes to the central repository so all developers have the new version available. It is also possible to have an automatic integration manager. This server will accept pushes, run all tests and if everything is ok it will then push this changes to the central repository. Otherwise it can discard the changes and signal the developer the reasons (most likely through an email).

VI. CONCLUSION

Does everybody need these features? Speed, offline working, and easyness of branching/merging are nice features even if you work simply pulling and pushing changes against a central repository and not every peer out there. In some projects and with some tools, the advantage of going distributed may not be clear. For starters, in many closed source projects there is no need nor desire to have anything other than centralized repositories, and while you can use a DVCS with a centralized repository, there may not be enough to gain to balance the cost of changing. Distributed versioning *is* more complex, and there is a "conceptual overhead"

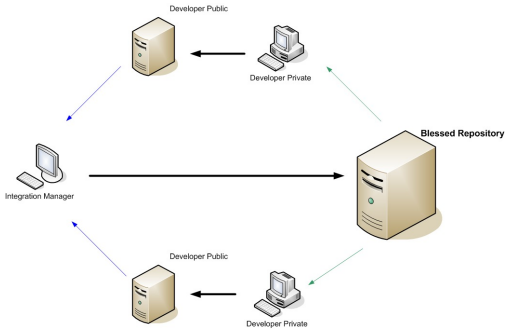

Figure 7: Centralized repository with integration manager workflow

in the usage of it, even if most of the time you can ignore most features and use just what you need (even for one person proyects, Git and Mercurial may be easier to use than SVN).

The general lack of maturity in the tools is also a disadvantage, since the most flexible way of working is the command-line interface. This makes it a bit harder to use because there are many commands, and they have a lot of options and some problems may not have an intuitive solution. There is more to learn and to use.

If there is no need to use anything but a centralized workflow, as in Continous Integration workflows, there might be little justification to use DVCS because changes are supposed to be integrated quickly in the main repository.

Beside these general warnings, Distributed VCS is simply more powerful and flexible than Centralized CVS. Everything that can be done with a centralized schema can be done in a distributed fashion, but the inverse is not true.

REFERENCES

Linus Torvalds *Google Talk presentation on Git*

Perforce documentation on "Editing Client Specs"

http://www.perforce.com/perforce/doc.current/manuals/p4web/help/editclient.html

Git-SVN Crash Test Course *http://git.or.cz/course/svn.html#merge*

DVCS Performance Comparison *http://ldn.linuxfoundation.org/article/dvcs-round-one-system-rule-them-all-part-3*

Kernel.org wiki Git Interfaces Comparison *https://git.wiki.kernel.org/index.php/Interfaces,_frontends_and_t*