Distributed Version Control Systems

Damian Alonso, Bruno Bonnano, Nicolás Calligaro and Nicolás Perez Santoro

***Abstract*—Abstract to be done**

***Index Terms*—to be done**

I. Introduction

VERSION Control Systems (VCS) usually track the whole history of the source code in a central server that everyone that can access. This central server contains all the changes and branches of the code and the users that can read the versions or commit changes have to connect to this repository. The most popular today example of this approach is SVN, which is a widely used VCS.

Distributed Version Control Systems (DVCS) are a modern way of managing revisions of software that do not have a centralized server, for this reason they are also called De-centralized Version Control Systems. Everybody has a copy of the full repository with all changes made in the 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.

We're going to do a comparison between centralized and decentralized control version systems, highlighting the differences and advantages and disadvantages of both approachs in various contexts, both on a conceptual and practical level. In this paper we'll base our explanations on Git but there are no big differences between most of these DVCS, especially between Git and Mercurial that are largely similar, with most commands having the same names in both tools.

II. Architecture

In a DVCS like Git, everybody has the full repository in their hard drives. Since there is no central server, all the operations on the repository are done offline. This means that actions like commiting do not depend on having a network connection, since commits are made on your repository.

Changes can thus be shared between everyone passing changes between repositories, without the need of making those changes in a central repository.

A. More Atomical Commits

In a centralized VCS, most of the time commits aren't done until you are sure that the change is well written and can be actually shared to everybody else, since the commit is going to be shared as soon as it's done. In a DVCS the nature of the commits is different, since commits are made in the local repository only. Therefore, they can be more atomic, because commits aren't huge changesets but can be logical, coherent and smaller changes.

Many little commits, if needed, can be grouped as a bigger, conceptual commit. To do this in a centralized VCS, one would have to create a new branch of the source code, and then merge the branch back to the trunk. But creating and merging branches all the time is not effortless, and branches can be seen by everybody that has access to the repository, so this is not always done.

Doing logical commits allows you to do 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 (this also makes commits easier).

B. Tagging and Branching

Centralized VCS's like Subversion makes certain copies through history called tags. On Git we have more tools, with one of them you can have an arbitrary description attached in the tag. In fact, you can store the whole release announcements there. Moreover, as with the commit, the identity of who made the tag is stored and this identity can be securely verified if we add a cryptographically PGP sign to the tag.

C. Merging

Merge on Git between branches is greatly improved over SVN, because all the history of both branches is preserved over the merges. When we make a merge we have two scenarios: Fast-forward merge: we refer to this when the changes only was made on only one of the branches, they are simply replayed on the other branch. Three-way merge: When changes exists on both branches, they are merged and git will report any conflict and let you resolve them.

D. Pushing and pulling

When we want to share our work with the rest of the team we have two basic concepts, *push* and *pull*. Push refers to taking all my changes and add them to the repository of another peer. But in practice we don't do this becouse we shouldn't make a push on a peer working copy, instead it, we push in a "central repository" where all the team make their pushes too, this logic is similar to tra-

ditional VCS, but we need to keep in mind that we could have a lot of "central repositories" for different parts of our project. Pull is a composite action of "fetch" and "merge", fetching takes the changes from selected repository of a peer and copies them to a local repository, and then those changes are merged with the sourcecode of my local repository. In the same way, another peer can pull on my working copy and get my code and changes without any problem.

If you work against a central repository, pulling and pushing do look very much like the idea of checking out and checking in, but that's the only way of working 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 just sharing the directory) to let other people pull from them.

III. Should I use it in my project?

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 coworkers 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.

B. 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 con clone the integration repository and merge it with its working repository. They can ask the person in charge to pull the changes back again.

IV. Then, why should I use it in my project?

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" 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). 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].

Another issue with these systems is that the tool support and various plugins just isn't as mature as systems like SVN, which are much more mature in that regard. Git, in particular, is biased on Linux and doesn't really work out of the box in Windows, but works with Cygwin. Mercurial is much more portable, but some tools are also incomplete, like hg-subversion (a tool to interoperate with svn). Git has git-svn which allows to push and pull from svn repositories, easing transitions.

V. Conclusion

to be done

References

[1] Linus Torvalds *Google Talk presentation on Git*
[2] Perforce documentation on "Editing Client Specs"
http://www.perforce.com/perforce/doc.current/manuals/p4web/help/editclient.html