Software Development and Collaboration: Version Control Systems and Other Approaches

Jose Antonio Escobar García
Technische Universität Berlin
csgermanico@gmail.com

May, 2011

***Abstract*—In the current context of software developing, to provide an efficient way to coordinate efforts and organize projects turn out to be essential. In this area, Version Control Systems (VCSs) help users to work in a collaborative fashion in a conflict-free environment. The most used VCSs, such as CVS or SVN are proving to be inadequate to meet the current needs and they are losing ground against other systems like Git. This last one propose a decentralized architecture (instead of a centralized one like in CVS or SVN) to meet the increasingly demand of faster and more efficient VCSs because more and more often software projects are developed collaboratively. But not everything is about software development, today Internet is, more than ever, a social environment and other approaches for collaboration such as Google Wave take another step forward, allowing real-time collaborative editing between different users.**

I. Introduction

Since the beginning of times, humans have had the need to collaborate in order to achieve a goal. The cavemen hunted together, collaborating to bring food to their caves. Usually, the main channel used for collaboration was language but a lot of time have passed since we lived in caves and now we need to collaborate in much more sophisticated tasks. In a globalized society like ours where technology is a part of our daily routine, the common tools for communication between people such as letters are completely obsolete. We need more powerful and modern tools, but not even technologies like fax or email are enough since the standards set by technology require to address a lot of different aspects for collaboration: users and permissions management, concurrent editing, version control, localization, network access, etc. In this way, one of the most common fields where collaboration turns out to be an essential part is software development. But it is not only an essential part but also affects the development itself. Because of this Virtual Control Systems (VCSs) were introduced.

Before VCSs, in order to perform a good version control in software development, a really great effort and discipline were needed from developers. They needed to maintain near identical copies of the work and tag them appropriately. Although this could work for a while, the chances to make mistakes were great, for example, a system where a developer working on a project had a big box of floppy disks tagged with texts like *"Version 0.9 20/03/84"* or *"Version 0.9 17/05/84"*. It is true that the use of floppy disks could look a little cavemen-like but we can also think of a group of developers sending each other hundreds of emails with source code. Therefore, it is obvious that we need to automate these tasks, what is precisely what VCSs do.

One of the first VCSs, *Source Code Control System (SCCS)*[1], was developed in 1972 by Marc J. Rochkind. In his work, Rochkind points out that as long as a version of a program is still supported, changes will be made to these versions. This highlight the importance of an efficient way to manage all these changes. In the year 1990, with the arrival of *Concurrent Versions System*[2], also known as *CVS*, the field of revision control software experienced a great boom. *CVS* works in a server-client manner, where the current versions of a project are stored in a central server and users connect to it in order to obtain, modify and update the source code. It is also one of the most widespread VCS and it is still being used in a lot of important projects but as discussed in further chapters, it is being gradually abandoned by many of these projects because of the problems and frustrations caused by *CVS*. Besides *Subversion (SVN)*[3], *CVS* represents the most famous Centralized Version Control System (CVCS), characterized by its server-client architecture. This server-client architecture presents numerous drawbacks and in an attempt to overcome them, Decentralized Version Control Systems (DVCSs) like *Git*[4] were introduced. One of the main problems of CVCSs is that working with a central server implies that everyone has to connect to this server, which cause a dependence of the network media. More and more projects are migrating[5] their version control system from a CVCS to a DVCS. Projects like *Perl* and others are currently studying the viability of making this change. As big companies and projects are migrating their repositories to DVCSs, this matter deserves to be studied in detail and will be addressed in further chapters.

But not only Version Control Systems are relevant in the field of collaboration. As the Internet grows more and more in its social aspect, new ways of collaboration appear every day such as Real-time collaborative editing. In this line, *Google Wave* allows the users to share and elaborate documents and ideas in real time.

The rest of this paper is structured as follows. In section 2, we provide a general idea of how a VCS work, explaining the main concepts behind them such as *Branch* or *Check-in*. In section 3 we discuss the advantages and drawbacks of CVCSs and DVCSs, pointing out their strengths and their weaknesses

in order to answer the question of why DVCSs seem to gain more and more importance. In section 4 we will show in some detail the three main and most famous VCSs: CVS, SVN and Git. In section 5, we will compare these three main VCSs and explain in detail how data are handle and how some important operations are performed by them. In section 6, we will give an outlook about other collaboration approaches like document collaboration or real-time editing. Finally, in section 7, we will point out some current and future work in the field of VCSs and will conclude by highlighting the main ideas shown in this paper.

II. Introduction to Version Control Systems

The software development process is not static at all. Instead, every single maintained published version requires attention and new features requested by users need to be addressed among other tasks such as bugs fixing, features research or new versions development. All this tasks usually involve a team of people (developers, release managers, testers, etc) who need to work simultaneously on different parts of the development process, usually involving parallel developing. This effort requires the use of Version Control Systems. There are a lot of different VCS tools in the market, both proprietary and open source, but essentially all of them use the same nomenclature with some modifications and nuances.

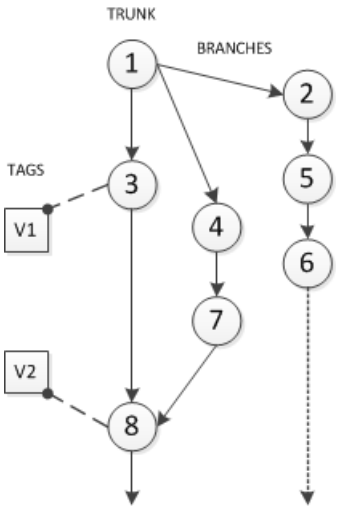

Fig. 1. VCS Repository structure.

Fig. 1. summarizes the basic structure and history flow of a VCS repository. Inside a repository, the current working versions of the source code are stored within the *Trunk* of the history tree. This *Trunk* represents the main development effort. It is possible to see *Trunk* referred with similar terms such as *Baseline* or *Mainline*. In a certain point in time, a version could be branched, creating a new *Branch* to the tree. A branch can be developed parallel to the main version and then be merged to the trunk again or follow a completely different evolution. In some systems, a branch can be also declared as a *Mainline*. A *Revision* is a version of a file. The other main concept behind a VCS tree is *Tag* or *Label*, that identifies a concrete set of files in the timeline with a human readable and comprehensive name such as a version number for a main release or a version name.

When working with a repository, several actions can be performed. The first step to create a new repository requires to make an *Import*, the action of copying a local directory to the repository for the first time which implies that from that point, a version control will be associated to this directory. In the case of working with a previously existing repository, the first step requires to make a *(*Checkout), creating a local copy of the current revision of the repository and allowing the user to work with it. This checkout action is only needed the first time because once we have a local copy, making an *Update* over a file or a group of them, changes made to the file(s) by other users are merged in our copy. The action of *Commit* or *Checkin* is to write changes made in our local copy to the repository. A very important action in any kind of VCS is the *Merge* action, allowing to merge different branches, merge local copies with the working copies on the repository (or the way around) among others. Although *Export* action is very similar to a checkout, the difference between them is that a exported version does not keep any version history. This action is useful for example for obtaining release versions ready to be distributed.

III. Centralized vs. Decentralized systems

There are two main architectures of Version Control Systems: centralized and decentralized. As a direct consequence of different architectures, the way that CVCSs work differs from DVCSs. Fig. 2. illustrates how a CVCS works. The CVCS software runs in a central server where all the version control take place. Users make local copies of the current working version or branches of the repository, making changes to the local copy and then submitting those changes to the central server. In other words, all the maintained files are stored within this server. How the VCS reacts when a conflict arise (e.g., two users submit the same file at the same time or a file has changed in the repository since a user updated it in his local copy) depends on the VCS. One option is file locking, that means that when a user make an update over a file, this file stop to be accessible for the rest of users until this user make a ckeckin. This option is not very efficient and what usually VCSs do is to try a version merging between the user and repository versions of a file, prompting the user in order to try to guarantee the correctness of the merge action. Usually the option to modify the files in the repository is restricted to a selected group of trusted developers, also known as *commiters*.

In the other hand, decentralized VCSs work in a different way, as illustrated on Fig. 3. DVCSs follow a peer-to-peer approach, completely opposite to CVCSs server-client approach. There is not a central server to work with, instead every checkout represents itself a first-class repository, containing all the history. Then, users work on their own local repositories, *pushing* and *pulling* changes as patches with other users repositories. In addition, instead of having a enforced trunk branch, users organized in work groups or communities, by

convention, decide the canonical branch, enabling the possibility for a project of having several mainline branches. Because of this, an efficient way of merging is essential in DVCSs more than ever.

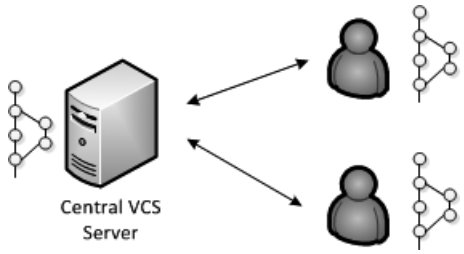

Fig. 2. CVCS Workflow.

It is not a clear decision why some project or company should use a centralized or a decentralized version control system, it depends on the needs. Despite this fact, decentralized (also know as distributed) VCSs are gaining more relevance. Until the ends of 90's, the main available options for version control were based on centralized architecture but in the beginnings of the new millennium, distributed architectures started to gain popularity until the point of possibly become the mainstream of VCSs in the near future. Why is this happening?. There is not a clear answer but maybe learning what are the most important advantages of DVCSs against CVCSs can help answering the question[5][6].

- **Offline operation**. One of the main motivations behind DVCSs is the need of being able to collaborate offline. Most centralized approaches requires a network to operate with the repository server which introduces the problem of depending in a *network of trust* which could not be always guaranteed. Therefore, as in DVCSs each user has a first-class repository, people are able to work offline.

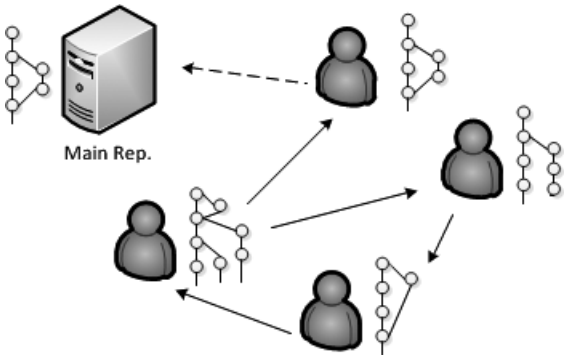

Fig. 3. DVCS Workflow.

- **Fast common operations**. Another consequence of this "network semi-independence" is that as the whole repository is at a local level, the most common operations (i.e., commit, merge, etc) are performed really fast, decreasing the operation time overhead introduced in centralized models where these operations need to be done with the main server through a network connection.
- **No canonical reference copy**. No reference copy of the codebase exists by default, only working copies. This allow to different users to decide which one is going to be their mainline within a repository and also they can work more individually.
- **More security against catastrophic failures**. One security problem of CVCSs is that in a likely scenario where a catastrophic event occurs in the main server, although people working with this repository probably have some copies of it, they do not have the whole version history and maybe not even the latest working copy and as a consequence the complete repository or a big part of it could be lost. The central server is a critical point in the system. In DVCSs, because of people having whole copies of the repository, even if a catastrophic failure occurs to one of them, there still existing probably unlimited copies of the whole repository and most probably some of them completely up-to-date.
- **No central authority**. In DVCSs, every single user is the owner of his own repository and therefore they are full-permission users of it. This fact increases the collaboration rate in projects since no special permissions are needed to submit changes made by one user, like in centralized alternatives where only a few selected and trusted users have the right to submit changes. As a direct consequence, the figure of *commiter* becomes irrelevant (or at least partially).
- **Less necessary resources**. In large projects where a lot of people are working with a single central server, the resources requirements for this server could be really big, increasing the operational cost. With decentralized systems, the requirements of storage space, memory or network capacity are significantly lower (if any).
- **Experimental branches**. Having local copies of the repository encourage user to develop new experimental features. These developers can create experimental branches on their own repository with the purpose of developing or experimenting with a new or existing feature and see the outcome of such modifications at a local level by commiting to local snapshots before including those changes in the main project. Also, as these "experimental branches" remain local, the developer does not feel restrained because of the fear of causing problems in an existing branch of a central repository. In addition, with cheap and fast branching operations, creating or deleting them do not suppose any operational overhead.
- **Simple automatic mergering**. The amount of information maintained by DVCSs make the merging operation very efficient and also automatic, a very important aspect in long-lived branches. This easy mergering also encourages the users to keep their branches up-to-date with the mainline development, reducing in the process the amount of work for committers.

- **Flexibility**. The flexibility of a distributed system make possible to use several workflows in contrast with CVCSs where the only possible workflow is working with a centralized server. In this scenario, the decision of selecting the workflow depends on the people involved in the project. In DVCSs there are a lot of possible workflows but one of the most used is the *Partner workflow*. As described in [7], *"[...]With a Partner workflow, a developer starts a project, then makes a branch. They then merge changes back and forth between branches that another developer is working on, and merge those changes in each time[...]"*.
- **Full version history**. When a user makes a checkout of a centralized repository, he only gets an snapshot of it. In contrast, when doing a checkout of a decentralized repository, the user copy all the previous versions and their history.

The previously highlighted features of DVCSs are only the most important ones. Like in any other technology, not everything are advantages, there exist also a few drawbacks. However, most of these drawbacks can be overcomed with the own capabilities of decentralized systems or with additional tools. Although the question of where are the lastest versions of the project files could look like a drawback, this is only a partial problem in DVCSs. Flexibility in decentralized systems allows to work in a centralized fashion, for example, by having a central repository where all the independently developed repositories push their changes. This problem is also addressed with tools like *GitHub* or *Launchpad* that allows to have a reference repository.

Other important issue of DVCSs in comparison with CVCSs is that a locking behavior is not possible. In a centralized system, we could implement a locking policy for the main repository, locking a file when a user is working on it until he makes a checkin of the mentioned file. Although this behavior slows down development, sometimes can be useful to avoid merge conflicts. In decentralized systems, this behavior is usually not possible because concurrence between DVCS repositories can happen, making possible an scenario where merge conflicts can occur. A security issue using DVCSs for proprietary software is that if for some reason (e.g., hacker attack, organizational mistake, etc.) a repository becomes publicly available, it could be cloned by a malicious programmer with no good intentions.

In any case, choosing one system over another depends on the needs. For a local development that does not involve much people maybe a CVCSs works better since it is more simple to use and organize. In the other hand, for big collaborative projects, specially in the case of open source projects, where people all around the world collaborate and need more flexibility to make changes on the code, a DVCSs seems to be the wisest choice.

IV. CVS, Subversion and Git

A. CVS

Concurrent Versions System, also known as *CVS*, is an open source software for version control. Started as a few shell scripts written by Dick Grune in 1986, it is the most famous exponent of centralized VCSs. Distributed under a GNU license[8], it was first designed and coded by Brian Berlin in 1989, based on a previous existing system called Revision Control System (RCS)[9], which only allowed managing individual files instead of projects as CVS does. As a CVCS, CVS runs on a central server and users, through CVS clients tools, make a local copy of the current working copy of the project stored in the server to continuously update their local copy and commit changes to the server. Usually, the server version of CVS runs over Unix systems but other similar projects such as CVSNT allow to use it under different environments.

When a user successfully makes a check in, the version number of all involved files are automatically increased and a description given by the user, the author's name and the current date are written into a log file. This log file not only allow the system to track different versions, it also provides the option to modify the logging process, allowing to perform additional operations such as triggering email notifications about new versions or showing them in a web-based interface among other possible uses. If a user working in a file in his local copy receive a new update made by other user to the same file, CVS try to automatically merge both versions into one but if it is not possible and there exists a conflict, it is up to the user to decide how to proceed to solve the problem, CVS just mark the conflicting parts of the source code.

Within a CVS repository, different projects stored are referred as *modules*. Inside those modules, source files of projects are stored using a delta compression algorithm in order to achieve an efficient way to store all the data of different versions for a same file. In addition, CVS was one of the pioneers introducing the concept of *branching* to the field of version control systems and the great part of the current branching techniques implementations derive from CVS branching technique. CVS also allows anonymous read access, also known as *Anonymous CVS*, which make possible for users to make a checkout of a project without any kind of permissions needed previously, restricting only the ability to commit changes to the repository.

But as a really mature project, CVS lacks of some necessary features and it seems to appear to be *out-of-date* for the current software development requirements in version control. As mentioned in the own CVS Manual[2], *"CVS can do a lot of things for you, but it does not try to be everything for everyone."*. To better understand this problematic we highlight the main shortcomings of CVS.

- CVS does not version the moving or renaming of files and directories. This could be relevant to the context of the project. For example, in a web project managed with CVS, a complete restructuring of the files structure could

indicate a change of the design pattern or framework used to develop this web.

- Limited support for Unicode and non-ASCII filenames. CVS was initially developed for Unix systems which use UTF-8 encoding but with the increasingly popularity of CVS, it started to be used with other operating systems that maybe do not use this encoding as default.
- A very important issue of CVS is that commits are not atomic. Atomicity means that when a commit takes place everything is transmited or none, which implies that is not possible to have some files modified and other not. In CVS if something goes wrong in the system during performing an operation and it is interrupted, there are no guarantees that the repository will not be left in an inconsistent state, leading to possible repository-wide file corruption.
- Limited tagging operations. Also renames are not supported and doing it manually can split the version history in two.
- Despite the fact that CVS introduces branches, branch operations are really expensive so branches should be short-lived and all the main development effort should be focused in the trunk.

In an effort to overcome all this problems and include new features, a lot of similar CVCSs were developed, projects like *CVSNT*, *EVS*, *OpenCVS* or *Subversion*. All these CVS based systems are usually know as *YACC*, *"Yet Another CVS Clone"*.

B. Subversion

Subversion is a open source version control system, distributed for free under an Apache License[10]. It was first developed by Collabnet in year 2000 and since them it have been continuously supported and improved for the open source community and several companies, launching their latest stable release only a month ago of the writing of this paper. During several years, Subversion have been the de-facto standard for version control of a lot of software projects. SVN was born with a clear and simple objective in mind: become a functional replacement for CVS. It would be very similar to CVS, doing everything that CVS does but implementing new features to fill the gap left behind by CVS and his faults of design. SVN should operate and maintain a very similar structure to CVS so its target audience, all the people working with CVS, could migrate to SVN with only a little effort. Therefore, Subversion is sometimes referred as an "improved CVS". Since its appearance, Subversion have been adopted for a large number of famous software projects such as Ruby, Mono, PHP, Python or the Apache Software Foundation itself (currently hosting the project) and other big companies like Google offers free SVN hosting with initiatives like Google Code.

Subversion is an effort to overcome the main CVS drawbacks[11]. In this matter, SVN is able to rename and make copies of files within the repository since it does not uses a RCS-like file system anymore, allowing also more efficient binary files handling. Instead, SVN keep track of all changes not only in a per-file basis but of the entire directory tree. In addition, SVN keeps an invisible hash table for each file or directory allowing the user to write their own information about them such as permissions, MIME types, owner, etc. There are two main technologies available in SVN for the filesystem: FSFS and Berkeley DB, both allowing important features such as data integrity, atomic writes, recoverability and hot backups. Precisely, atomic operation is another important feature that its predecessor did not has. Now, if something goes wrong during a commit operation, all the process is canceled and the repository remain in an stable state. When commiting, everything is transfered correctly or none.

Different from CVS, Subversion has a native support for client-server operation and allows an advance network layer[12] through the use of WebDAV module from Apache 2. Because of this advanced network layer, there are several ways to operate with a repository. The file system can be accessed directly by the users at a local level through an URL scheme (*"file://path"*) but also it can be accessed (thanks to the DAV module) over the http(s) protocol. In addition to this access method, svn defines its own protocol over TCP/IP that also allows ssh tunneling through a "svn+ssh://host/path" access scheme. Using Apache's solution not only offers a really mature and reliable server communication, it also provide another existing features that can be used through other Apache server modules such as htpasswd or mod_deflate.

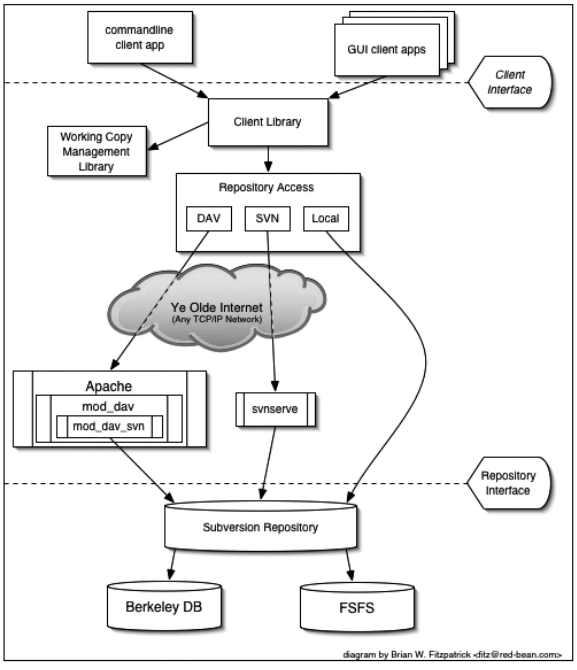

Fig. 4. SVN structure[3]

Subversion has a layered design. All C libraries involving SVN are organized in different directories depending on their function, composing different layers of the system (i.e., network management, filesystem, etc.). This way, developers can easily identify different parts of the project and work on

their own contributions. Fig. 4. summarizes all the different parts of Subversion's design. In the SVN Book[3], a lot of functional issues are explained in depth.

C. Git

Git is an open source distributed version control system, distributed for free under a GNU General Public License v2[8]. Right now it is one of the most famous and widely-used DVCSs. Git was developed by Linus Torvalds for managing the Linux Kernel project. In the beginning, it was maintained by exchanging patches between collaborators but in 2002 it began to use *BitKeeper*[13], a DVCS, as the main version control system. BitKeeper is proprietary software and after some troubles that arose in 2005 between Linux Kernel community and the company responsible for BitKeeper, they broke down their relationship and the company revoked the status of free-use of their tools. After that happened, Torvalds decided to design and implement his own distributed version control system for Linux Kernel maintenance following the principles used in BitKeeper and that same year, the first version of Git was released and it still under development. There is not a clear definition for what "Git" stands for but in the British English slang Git means *"An idiot or contemptible person"*. In Linus Torvalds own words, *"[...]I'm an egotistical bastard, so I name all my projects after myself. First Linux, now Git.[...]"*. One of the most remarkable characteristic of Git is its fast and efficient operation, allowing to perform branching and merging in a really fast way, issue that will be addressed in further chapters. In fact, one of the design criterias behind Git development was to take CVS and learn exactly how not to do things. Torvalds is one of the main critics of CVS. The basic operation of Git is very similar to any DVCS. As explained before, every user make a local copy of a directory and work with it instead of working with a central server. Then the user work in his own repository and submit changes to other users' repositories.

One of the most important differences between Git and almost any other VCS (besides all that implies being a DVCS), is the way how Git treat data. Usually in the most part of VCSs, information is perceived as a set of files and the changes made about them. In Git, instead of keeping a reference to the changes made to files, every time a user commit a new version, Git takes an snapshot of the state of all files and copy them as a new version. A more datailed view to this issue will be offered in further chapters.

As a direct consequence of this data handling, Git becomes a mini filesystem itself, allowing to develop very useful and powerful tools in top of it. Besides the advantages of working with a local repository (e.g., offline working, fast operations, etc.), other important improvement introduced by Git is integrity. Everything under Git control has a checksum, allowing to detect any kind of changes on files and preventing corruption. In fact, Git implements very strong measures to prevent corruption, making it a very reliable system. Other important aspect of how Git works, is the possible states in which a file can be: committed, modified or staged. When a user modifies a file in his working directory, it gets the status of modified and then the user needs to stage the file. When a file is staged, it means that the file is ready to be committed. Once a file is committed, it is safely stored in the repository. Therefore, the basic workflow in Git is to first modify a file, second stage it and third commit it. In addition, when a commit operation is performed no data deletion is possible, which implies that almost every action can be undone, making virtually impossible to make a catastrophic mistake.

As said before, SVN is one of the most used VCS. For this reason, the Git team developed a tool, *Git svn*, that serve as a bridge between Git and Subversion. This tool allows to a user to use Git as a SVN client, with all the advantages that this fact represents. This could be an ideal temporal solution for projects looking to move their source code to a DVCS while they update the infrastructure to support Git. Also, it represents a great opportunity for Git evangelists to divert developers' interest toward a new model of VCS.

V. Comparison of VCSs

Today the main dilemma in the field of version control systems is whether to choose a centralized or a decentralized approach. In section *Centralized vs. Decentralized* we analyzed the main advantages of each approach and pointed out that the current trend is to move toward a distributed architecture. Also we saw that in many cases the decision of which kind of VCSs to use depends on the requirements and also in the preexisting structure. In any case, the current battle is between the two more representative systems of both CVCSs and DVCSs. First, with almost a decade of developing, based in other widely used and mature system and with a lot of big projects working with him, we have Subversion, the de-facto CVCSs industry standard. In the other hand, with only a few years of development but with a solid background and bringing new life to the field, Git, the most representative DVCS right now.

Linus Torvalds, as creator of Git and one of the most important defenders of decentralized version control systems, charge hard against CVS in a bluntly speaking way, *"[...]It's really not very easy to explain why CVS sucks. After all, sometimes people who have used it for decades have a hard time understanding the suckiness[...]"*[14]. But Torvalds not only talk bad about CVS, he also share his opinion of SVN, *"[...]Because my hatred of CVS has meant that I see Subversion as being the most pointless project ever started[...]"*[15]. For Torvalds, the issue is not only the implementation problems of CVS (most of them are solved by SVN), the main problem behind those systems is the centralized view itself.

Some people has argued against DVCSs in general and against Git in particular, that if things are not done properly, it can get very confusing in no time. Having multiple repositories also leads to a lot of almost identical information with a few changes spread among several locations and an extra effort to put everything together need to be done. They argue that the best thing about centralized models is that they keep it simple. The Better SCM initiative[16] has available a

table-like comparison between several version control systems, Subversion, CVS and Git among them. In the end, it is up to the user to try them and decide which one fits better to his needs, although it seems like decentralized systems like Git are the preferred ones among the open source community, a very big community for which collaborating is as essential as the developing itself.

Besides the subjective point of view of the user, that Git is faster than CVS and Subversion is a fact. The particullar way of how Git treat and store data makes possible to perform fast basic operations (since the repository is store locally), very efficient tagging and branching and automatized merging. How all these tasks are performed by each one of these VCSs is detailed in following chapters, allowing to understand why Git is faster than the rest. In general, Git is two or three times faster than Subversion[17] except in those tasks involving binary data where SVN is clearly more efficient.

A. File systems

How a VCS treat data is a very important issue. Data handling and storage has evolved toward VCSs, improving it and creating new ways to perform this task. As mentioned before, CVS was created based on a previously existing system, RCS. Essentially, CVS use this system as file system but with some modifications. The repository is represented as a file system tree with all the directories and for each file within a directory a *"History file"* is created. The history file has the same name as the file under version control but adding ",v" at the end (e.g.: "process.c,v") and contains enough information to restore any revision of the file in addition to a log containing the messages and user names of the authors for each commit operation. In addition, since version 1.7, a special "fileattr" file is stored within the repository. This file makes possible to store attributes regarding other files under version control. All entries in this file follow the schema *ent-type filename attrname=attrvalue*, where ent-type is a character representing the type of an entity ('F' for file, 'D' for directory and so on). History files are also referred as *RCS files* since history files are treated like in a RCS file system[18].

As discussed before, Subversion is an effort to improve CVS and one of the areas to be improved was the file system. Instead of using RCS as file system, Subversion offers two kinds of file systems: Berkeley DB[19] and FSFS[20]. The first one use a database to store everything while the second one use regular files for version control, more similar to how CVS handle data. Which one should be used depends once again on the user, since both of them have their benefits and limitations. In Pag. 148 of Subversion manual[3], a comparison between both systems is given, comparing aspects such as data integrity or sensitivity to interruptions. Despite everything, FSFS file system is the most used since starting on version 1.2 it is used by default and also it is easier to migrate. FSFS use the OS native file system to store all the changes associated to a revision in a file with a revision number. All revision files are stored in a subdirectory of the current file system tree directory. For implementing atomic operations, while performing one, all revision files are stored in a different subdirectory for the transaction and only when it is complete, the content of it is moved to the revisions subdirectory. A revision file contains the revision directory structure, date, author and log information of the revision and the changes made to the file referred to the previous revision, among other information.

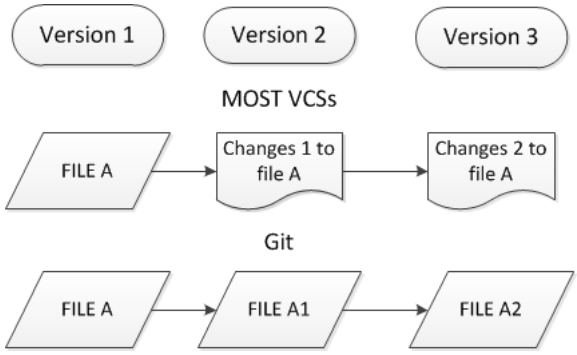

Fig. 5. Data handling comparison.

Git treats data in a different way. Usually, VCS systems store only the changes made in a revision referred to a previous one but instead of doing this, Git takes an snapshot of all files and copy them as a new version. Fig. 5 illustrates this behavior. For keeping all the information and the files histories, Git uses two main data structures: a mutable index for handling files states and an immutable database object which only option is to append new objects. These objects stored within the database are of four kinds: a blob object, a tree object, a commit object and optionally a tag object. A blob object is the file itself and do not has any additional data (e.g., metadata, filename, timestamps). Then, tree objects represents directories and contains the list of elements within that directory and the name of the blob or tree objects (in case of subdirectories) representing them. Therefore, a tree object represents an snapshot of the source tree. Finally, commit objects link different tree objects representing the history. Fig. 6. illustrates this data structure.

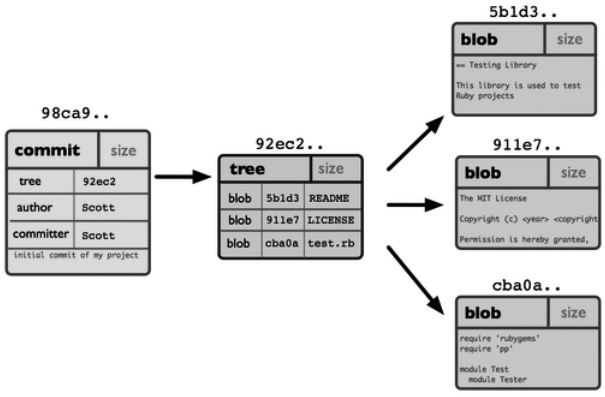

Fig. 6. Git data structure[4].

This allows Git to perform very fast operations (not only because of working with a local copy) because there is no need to rebuild a specific revision by applying different change sets to previous versions, instead Git just access the file for the corresponding commit object of the revision.

B. Tagging and branching

In CVS, when a tagging operation is performed within a repository, all files and subdirectories within tagged directories are stamped with the tag (recursively). In the same way, when branching, the branch tag is also stamped in the files. In the case of branching, they receive an special version number. If a third branch was forked in version 1.2 of the trunk, revisions in the branch will be numbered 1.2.3.1, 1.2.3.2 and so on. This action makes branching and tagging very expensive in an operational sense. In SVN, branching and tagging are identical operations. Whenever a tag or branch is created, SVN just make a copy of all the directories tagged/branched, so the operational cost is the same that making any copy of the directories. Although a branch and a tag are copies of a part of the repository in a certain point in time and they are performed the same way, the difference is that it is not possible to modify a tag but a branch allows users to keep modifying the files parallel to the trunk.

With Git, branching and tagging work in a different way due to its particular way to keep data. When a commit operation is performed, Git creates a commit object containing all the metadata, a pointer to one or several previous commits and a pointer to a tree object with all the information about the files in the revision and the files themselves. A branch is only a lightweight movable pointer to one of these commit objects. By default, Git creates a "master" branch pointing to the latest commit and this pointer move every time a new commit is performed. When a new branch is created, it also point to the latest commit in the moment of its creation. The branch defined as HEAD is the current one and every time a new commit is performed only the pointer of the HEAD branch moves toward it. When switching between branches, the only action to perform is to define which branch is the HEAD. Fig. 7. illustrates this behaviour. As a branch is only 41 bytes (the pointed commit number) written in a file, branching is a very fast operation in comparison to other systems such as Subversion. There are two types of tags in Git: lightweight and annotated. The first one is like a branch which does not change, it is only a pointer to a commit and do not include any extra information. In the other hand, annotated tags are stored as full objects, containing the tagger name, e-mail and date in addition to a tagging message.

C. Merging

Merging is one of the most important operations within a VCS. When merging in CVS, the greatest common ancestor of the branch and the destination revision (usually, the point where the branch forked) is merged with the last revision of the branch. When the merge operation is performed, it is possible that one or more conflicts arise. If this happens, the merge process is abandoned until the conflicts are solved manually. The problem in CVS is that no tracking information of merges are stored so for several merges between the same branches, the correct revisions numbers need to be specified by hand.

In subversion, branches are full copies of the repository in a determined point in time, so when two branches are merged (usually one of them is the trunk), Subversion looks for the differences between the revisions and merge them. The problem was that, before version 1.5, as in CVS, no information about what change sets were merged, by whom or when was stored, making not possible to perform automatic merges and needing to manually indicate which revisions of each branch were going to be merged, introducing the possibility of conflicts made due to human mistakes. Since version 1.5, information for merge tracking is stored and Subversion is able to select the correct revisions to merge by itself, allowing multiple merging of the same branches automatically. Despite this automated process, conflicts can still arise when merging branches. In those cases, Subversion ask the user for the correct course of action, allowing to correct some conflicts "on-the-fly", offering several options for doing it such as visualizing the differences between the conflicting files, opening the conflicting file in a text editor and so on.

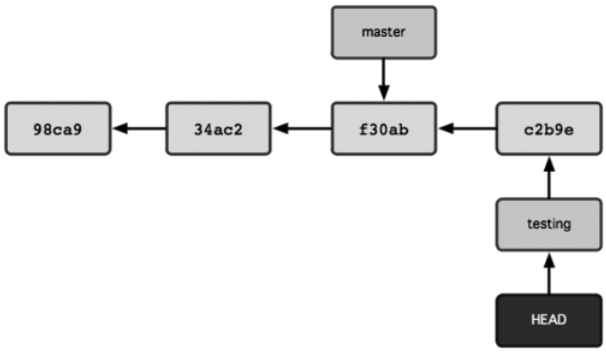

Fig. 7. Git branching[4].

Depending on the situation, Git performs merging in two different ways. In the first one, if the commit object pointed by the master branch is a direct ancestor of the commit object pointed by the branch, Git performs a *fast forward* merge, which means that the only necessary action is to move the pointer of the master toward the commit pointed by the branch. In the other case, if the branch has diverged from an older point, Git performs a *merge commit*. This is a three-way merge between the commit pointed by the master, the one pointed by the branch and the best common-ancestor between them (automatically chosen by Git), merging them and creating as a result a new commit object that has two different parents: the previously master commit and the commit pointed by the branch. This process, as in Subversion 1.5, is completely automatically, although conflicts may arise and in that case the merge process will be stopped until the user solve the conflicts.

VI. Other approaches for collaboration

Although it is a very important field, no all collaboration needs are about sharing source code. Other people may need to collaborate for other tasks rather than software development. For example, within a law firm, lawyers should use some collaboration techniques for sharing contracts to review or elaborate legal documents together. There are several options widely used for document sharing and editing, both in the cloud and offline. Initiatives such as Google Docs or Microsoft SharePoint allow common users to share documents between them. In the case of Google Docs, those documents are stored in the *cloud*, making them available always as long as the user has a network connection. Some privacy and user management policies can be applied to this tools so users can be organized in groups with only read access or to make documents available only for some users. Google Docs offers several type of documents for collaboration, from blank pages to spreadsheets, presentations or drawings.

Nowadays, Internet is more social than ever and in this context, Wikis are powerful tools for collaborative content creation. The most famous Wiki all around the world is Wikipedia, where users can contribute knowledge to an encyclopedia. Wikis could be also used to gather efforts for user manuals writing, F.A.Q. repositories and other purposes. Essentially, a Wiki is a website of interlinked pages that allow editing through a browser. In this context of social collaboration, tools that maybe were not developed with collaboration as a goal are used for this purpose. Social networks like Facebook or Twitter are not only used to share information with friends, they are also reused as a way to generate collaborative contents and also for coordination between groups of collaborators. Cloud storage services like Dropbox or Ubuntu One allow to synchronize local directories with the cloud and allow to the user to share some files or directories, transforming the service in some kind of a client-server collaboration system working with local copies, very similar to software development alternatives like Subversion.

A very striking approach for collaboration is real-time collaborative editing. This approach allows users to create and edit content collaboratively in real time. This means that while a user is writing a document, other users editing the same document are seeing the changes made to the document as they happen instead of waiting for other users to commit their changes. Google wave is a partially open source web-based platform for real-time collaboration among other things and it was released to the general public on 19 May 2010. Email was invented more than 40 years ago and Google Wave intended to be a conceptualization of how email would look today. For this purpose, Wave merges key features of different technologies such as email, instant messaging, social networks, wikis and so on. In Google Wave, instead of static messages containing the whole thread of conversation, the messages are alive and referred as *Waves*. A wave is part conversation, part document and allow to its creators to add more people to it. All users added to this wave can change it in real-time, also being able to revert to a previous state in the changes time-lime of the wave. It is also possible to add multimedia content such as videos or images. Google Wave represented a very powerful and flexible tool for real-time document creation and editing since it also offered communication with social networks and other tools like Google Docs. These waves were XML documents hosted in a central server that allowed concurrent modification. It also offered an API for developers making possible to write extensions and improvements to the system. Besides the web-platform and the API, Google developed a protocol for supporting Wave, the *Google Wave Federation Protocol*[21]. This is an open protocol and anyone can use it to develop their own wave system. Unfortunately, at the beginnings of August of the same year of its release, development of Google Wave was announced to be discontinued. Currently it continues its development under the name of *Apache Wave* by the Apache Software Foundation.

VII. Conclusion and future work

Collaboration between people is a very essential part of human relationships. In the field of software development, in a increasingly collaborative environment, in order to articulate the necessary mechanisms to ensure an efficient and productive interaction between developers, having a tool able to perform this task is essential. Version Control System tools address this problem of coordination. In the field of VCSs, we can identify two main groups of systems: centralized (CVCSs) and decentralized or distributed (DVCSs). The first one propose a client-server architecture where all files of a project and the version history are stored in a central server and users connect to this server to download a local copy of the repository, working on it and submitting/updating changes with the central server. In decentralized systems, every user has a local copy of a complete repository, including all the files and history of versions. In this model, users work with their own local repository, pulling and pushing changes from other users' repositories. Decentralized approaches have several improvements in comparison to centralized systems like the possibility of working offline or the fast operations due to be working locally without the constant need to interact with a central server that could be far away from the user location. Despite all the differences between both approaches, the final decision of which one to use relies in the user as one approach can meet his needs better than the other one.

There exist a lot of implementations of both systems architectures. Among them, CVS and SVN are most used CVCSs, being the second one a natural evolution of the other, improving several lacks of features and operation problems. Today, Subversion represents the alternative used by most projects as version control systems. In the other hand, Git represents the most used DVCS. It was developed following a clear design criteria: look at CVS and learn how not to do things. Git has a very close relation to open source community since the motivation behind its creation was to support the development of the Linux Kernel project, one of the most important open source projects since the 90's. Even though

SVN is right now the most used VCS, it seems that gradually more and more important projects are migrating their version control systems toward a distributed solution such as Git.

Besides software development, collaboration is required in other tasks such as collaborative document editing, real time collaboration or task organization. To address these needs, tools like Google Docs allows collaborative documents creation. But there exist also a lot of different approaches to meet different collaboration needs and Wiki based tools allow users to create and share a knowledge databases. A particularly relevant approach was proposed by Google Wave, which allowed to collaborate on different fields (e.g., document creation, brainstorming, organizational issues) in a real-time fashion where users could saw changes on documents they were working on as they happened.

That distributed control systems are becoming the most popular is a fact and in the near future they could become the preferred versioning systems. But this implies a very important issue. As most of the current projects under development use a centralized system, migrating to a distributed system is not an easy task. Some future work could be focused on proposing and developing efficient methodologies and tools to address this problem and make the task of migrating from one system to another much easier.

Cloud computing represents one of the most active fields today and every day new services based in this approach are born. Using this idea of *cloud*, a mixed approach of both centralized and distributed systems, where every user, despite of working as in DVCSs, would be able to publish their work into their own *cloud space*, allowing other users to access his work at anytime but at the same time acting as a *mail box* for the owner of the cloud while he still working offline.

REFERENCES

[1] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. SE. 1, no. 4, pp. 364–370, 1975.

[2] CVS Project. (2005) Version Management with CVS. [Online]. Available: http://ftp.gnu.org/non-gnu/cvs/source/feature/1.12.13/cederqvist-1.12.13.pdf

[3] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. (2008) Version Control with Subversion. [Online]. Available: http://svnbook.red-bean.com/en/1.5/svn-book.pdf

[4] S. Chacon, *Pro Git*, Apress, Ed. Apress, 2009.

[5] B. de Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, ser. CHASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 36–39. [Online]. Available: http://dx.doi.org/10.1109/CHASE.2009.5071408

[6] I. Clatworthy, *Distributed Version Control Systems - Why and How*, Canonical, 2007.

[7] N. Gift and A. Shand. (2009, April) Introduction to distributed version control systems. Weta Digital. [Online]. Available: https://www.ibm.com/developerworks/aix/library/au-dist_ver_control/

[8] The Free Software Foundation. (1991, June) Gnu public license v2. [Online]. Available: http://www.gnu.org/licenses/gpl-2.0.html

[9] W. F. Tichy, "Rcs: a system for version control," *Software - Practice & Experience*, 1985.

[10] The Apache Software Foundation. (2011, June) Apache licenses. [Online]. Available: http://www.apache.org/licenses/

[11] D. Neary, "Subversion - building a better cvs," *Linux Magazine*, vol. 30, pp. 59–63, 2003.

[12] B. Collins-Sussman, "The subversion project: buiding a better cvs," *Linux J.*, vol. 2002, pp. 3–, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=513039.513042

[13] BitMover, Inc. (2011, June) BitKeeper. [Online]. Available: http://www.bitkeeper.com/

[14] L. Torvalds. (2005, October) Git Mailing List discussion. [Online]. Available: http://marc.info/?l=git&m=113072612805233&w=2

[15] Linus. (2007, May) Google tech talk: Linus Torvalds on git. Google Inc. [Online]. Available: http://www.youtube.com/watch?v=4XpnKHJAok8

[16] Better SCM Initiative. (2011) Version Control System Comparison. [Online]. Available: http://better-scm.berlios.de/comparison/comparison.html

[17] Alexey Bokov. (2010) SVN vs GIT : perfomance tests. [Online]. Available: http://bokov.net/weblog/project-managment/comparing-svn-vs-git-in-perfomance-test/#test1

[18] W. F. Tichy. (1999) man page rcsfile (section 5). [Online]. Available: http://www.manpagez.com/man/5/rcsfile/osx-10.3.php

[19] Oracle Corporation. (2011, June) Oracle Berkeley DB 11g. [Online]. Available: http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html

[20] N. Cocchiaro. (2007) FSFS Filesystem Project. [Online]. Available: http://fsfs.sourceforge.net/

[21] Apache Software Foundation. (2011, June) Google Wave Protocol. [Online]. Available: http://www.waveprotocol.org/