Version Control Systems

Stefan Otte
Computer Systems and Telematics
Institute of Computer Science
Freie Universität Berlin, Germany
otte@inf.fu-berlin.de

***Abstract*—Classic centralized Version Control Systems have proven that they can accelerate and simplify the software development process, but one must also consider distributed systems in this analysis. What features can distributed Version Control Systems offer and why are they interesting?**

This paper describes the general concepts of the centralized and the distributed approaches, how Concurrent Versions System, Subversion and Git implement these concepts.

I. Introduction

By developing programs, software engineers produce source code. They change and extend it, undo changes, and jump back to older versions. When several software engineers want to access the same file, concurrency becomes an issue.

Version Control Systems (VCSs) enable the acceleration and simplification of the software development process, and enable new workflows. They keep track of files and their history and have a model for concurrent access.

There are two different approaches to VCSs:

1) the *centralized* model with the *centralized Version Control Systems (CVCSs)* and
2) the *distributed* model with the *distributed Version Control Systems (DVCSs)*.

The following sections will describe the advantages and disadvantages of the different approaches and how programs employ these approaches.

Using the right VCS eases the development process, while choosing the wrong stalls it. Therefore it is crucial to understand what the programs can and cannot do.

The remainder of the paper is organized as follows. Section II introduces the general concepts of VCSs. Subsequently, Section III discusses the implementation of these approaches using *Concurrent Version Systems (CVS), Subversion (SVN)* and *Git*. The paper ends with a comparison of these programs in Section IV and with a conclusion in Section V.

II. Basic Concepts

This Section gives an overview of the centralized and the distributed approach.

It is important to keep in mind that not necessarily all VCSs support the features described below, or they may handle some details differently (see Section III).

To emphasize the aspect that the software engineer is using a VCS and to ease the reading of the paper the software engineer is called the *user* from now on.

Even though, this paper describes VCSs in the context of software engineering, VCSs can be very useful in other areas of work too. They can not only handle source code and text files, they can also handle different file types.

A. Different Terms, Same Meaning

When reading about version control systems, terms like *revision control systems (RCS)*, *software configuration management*, *source code management (SCM)* or *source code control* come across. Even if these terms sound different, they mean basically the same and can be used interchangeably. However, in some contexts they focus on different aspects of the program, e.g. they run unit tests before or after every commit, or they automatically build an executable file.

Even though these types of features may be important or even critical for some purposes, this paper will focus on the versioning aspect and will therefore only use the term version control systems (VCSs).

B. Centralized Version Control Systems

Centralized version control systems are called centralized because there is only one central *repository*.

The repository is basically a server on which all tracked files including their *history* are stored. The repository can be accessed via LAN or WAN from anywhere in the world.

Every stage in the history of a file is identified by a *revision* or *version* (typically an integer). A revision consists of the file and some metadata. The metadata

that is stored can differ depending on the program (see Section III). The newest revision is often called *head* or *HEAD*.

The user can *checkout* single files as well as the entire repository and he can specify the revision he wants to retrieve from the repository (e.g. head or any other revision). By doing a checkout the user retrieves a *working copy* on his local computer. A working copy does not include any history but it enables the user to edit the files.

After editing the files (e.g. implementing a certain feature) the user *commits* the changed files to the repository. He also adds a *commit log* or *commit message*, which is a description of what has changed since the last version. This is a part of the metadata which is saved in the history. By committing the changed file from the working copy into the repository a new revision is created. The revision number rises.

The changes between two revisions of a file are called *diff* or *delta* (see Section II-E).

After a commit from user A, user B does not have the current revision of the files anymore. User B needs to run a *update* to retrieve the current revision.

If user B worked on the file which user A changed, the VCS tries to *merge* the files automatically. *Conflicts* may arise if the files contain incompatible changes. The user needs to resolve these conflicts manually.

The main development takes place in the *mainline* or the *trunk*.

If there is a need for an independent line of development, the user can start a *branch*. Branches are often used to implement a feature. During the process of implementation the branch may become unstable, but it does not affect other users. Once the implementation is completed and is tested, the feature from the branch gets merged back into the trunk.

Some revisions can be more important than others, e.g. a release of a program. VCSs offer the opportunity to *tag* revisions (to give them a catchy name). The revision can be accessed by using the tag (e.g. Version 1.4) instead by its revision number.

The best practice for handling trunk/branches/tags is to create a folder for each category.

Most of the operations above require network access. Therefore the network is always the bottleneck in CVCSs.

C. Distributed Version Control Systems

Distributed Version Control Systems are a new approach to VCSs. Most CVCS terms also apply to DVCSs. The differences are mentioned below.

The fundamental difference between DVCSs and CVCSs is that DVCSs do not require a central server that contains the repository. Every user has the complete repository on their local computer, it is called *local repository*. There is no need for a heavy weight server program. One simple command sets up the local repository.

Having an entire repository instead of only a working copy takes up more space. DVCSs use data compression for the repository to reduce this effect.

A local repository enables the user to work completely offline. Most operations are much faster because they are local. For example, to examine the differences between two files is a simple operation. Depending on the size of the file it only takes a fraction of a second. However, if one of the files is stored on a server the file needs to be downloaded first. The network unnecessarily slows down the workflow in centralized environments.

In DCVSs, commits, branches, tags, and checkouts fulfill the same tasks as in centralized environments except that they only communicate with the local repository instead of with the central repository.

However, at some point the user needs network access to share his repository with others. User A can make his local repository available over network or he can create a *remote repository* on a server, which is more common.

User B is now able to *clone* the remote repository of user A. From that point on, user B has a local repository and can work offline.

In order to stay up to date, user B *fetches* the changes from the repository he initially cloned it from.

To publish changes user B *pushes* his changes into a remote repository of user B. Of course, user B needs write access to push his changes into the remote repository of user A. Alternatively, user B can publish his changes in his own remote repository. User A can now fetch changes from user B. Note that DVCSs do not have an *update* command.

Every local repository is independent from all other local repositories. Therefore merging is much more important than in the centralized world. In general, DVCSs have better merge capabilities because they are aware of the entire history and track branches and merges.

Because of the local/remote repository structure and good merge capabilities, DVCSs are very flexible and can adjust to many workflows (see Section III-C9).

D. Merge Concepts

Even though this paper is not about merge algorithms, there are some fundamental concepts that need to be explained in order to understand VCSs.

The classic approach in scenarios that contain concurrency is the *lock-modify-unlock* mechanism. It is often called pessimistic approach. A resource gets explicitly locked by a process. Only this process can modify the resource. After modifying the resource the lock gets removed. It guarantees, that no merging is needed, because a file can be checked out by only one user.

This approach does not work for VCSs. The users tend to forget to remove the lock and the file can not be accessed by anybody.

VCSs work with the *copy-modify-merge* mechanism. It is often called optimistic approach. A resource can be copied and modified by more than one process but may be merged afterward.

The optimistic approach proved to be better suited for VCSs. It required less administration and allows a better workflow. Although, some CVCSs offer also the lock-modify-unlock mechanism, because it can be helpful in some scenarios.

E. What is Delta Encoding?

VCSs need a space efficient way to store different versions of the same file. Delta encoding is a way to store only the changes of different versions of a file instead of the entire file. The difference is called *diff* or *delta*.

Delta encoding is very efficient for source code because the part of the source code that changes is normally relative small.

To illustrate this method, here is an example. Every UNIX offers a tool diff. It shows the differences of two files.

```
1 public static void main(String[] args){
2     System.out.println(args[0]);
3 }
```

Listing 1. Diff example version 1

```
1 public static void main(String[] args){
2     System.out.println(args[0]);
3     System.out.println(args[1]);
4 }
```

Listing 2. Diff example version 2

Delta encoding stores the difference from line 3

```
1     System.out.println(args[1]);
```

Listing 3. Change between Listing 1 and Listing 2

III. Programs

This Section describes in detail two programs representing the centralized approach, Concurrent Versions System (CVS) and Subversion (SVN), and one program representing the distributed approach, Git.

A. Concurrent Versions System

1) History of CVS: Concurrent Versioning System (CVS) was one of the first VCSs that applied the centralized model [1] [2].

The groundwork of CVS was created by Dick Grune (it was called *cmt* at that point) at the VU University Amsterdam in the year 1984 [3]. He developed the system to be able to work together with two of his students on a compiler project. The progrm turned out to be very useful. After finishing the compiler project in 1985 Grune cleaned up the shell scripts of cmt, renamed it Concurrent Versions System and published it [4].

Cmt was built on top of RCS, which was a VCS designed to handle single textfiles. RCS itself was developed by Walter F. Tichy. Cmt/CSV extended RCS to handle more than one file. CVS still uses the RCS fileformat.

In 1989 CVS was rewritten in C by Brian Berliner [5] [6]. At that time he worked at Prisma, a commercial company. Prisma needed to update a large number of source code files to the kernel of SUN OS and therefore used CVS to perform this task. Prisma experienced a great improvement of the development process by working with CVS and also "cvs prevented members of the kernel group from killing each other" [6]. They made the code freely available so that others can enhance CVS to meet broader needs.

Over time CVS evolved into a very stable VCS which, for example, was used by sourceforge.net to host over 100000 FOSS projects, but also was used by many commercial companies.

Ben Collins-Sussman points out that:

> "CVS and its semi-chaotic development model have become cornerstones of open-source."[7]

The typical setup of FOSS project is the legacy of CVS. It also introduced the concepts of branches that exists today in almost every VCSs.

2) Limitations: Because of its dependency on RCS, CVS has many limitations. Here is an incomplete list of the major shortcomings of CVS.

- CVS was unable to handle binary files because of the RCS file format. RCS changes line endings and replaces some keywords in text files. Because CVS did not know if a file was a text file or a binary file, it handled binary the same way as text files. That is why CVS shredded binary files if line endings or certain keywords appeared in a binary file.
  Today the file extension is used to determine if it is binary or not. It is also possible to explicitly mark a file as a binary file. The binary files do not get destroyed.

TABLE I
PROPERTIES FOR CVS, SVN AND GIT

Category	**CVS**	**SVN**	**Git**
Distributed/Centralized	centralized	centralized	distributed
Initial Release	1985	2001	2005
License	GNU GPL	GNU GPL	GNU GPL
Version	1.11.7	1.5.6	1.6.1
OS	Windows, OS X, UNIX	Windows, OS X, UNIX	Windows (limited), OS X, UNIX
Atomic commits	no	yes	yes
Move and renames without loosing the history	no	yes (some exceptions)	yes (some exceptions)
Partial checkouts	yes	yes	no
Tracking of branches and merges	no	yes	yes
Revision numbers	increasing integer	increasing integer	SHA1
Local repository	no	no	yes
Central repository	necessary	necessary	availables
Workflow	static	static	flexible
Userinterface	Sometimes confusing	good	rather complex
Consistency check	no	no	yes (SHA1)
Tagging	expensive	cheap	cheap
Branches	expensive	cheap	cheap
Performance	very slow	slow	very good
Recognizes binary files	yes	yes	yes
Handling of binary files	no diff	diff	diff
Compression of repository	no	no	yes
Two phase locking	available	available	no
Path based authentification	yes	yes	no
Hosting support for FOSS projects	dereasing	Sourceforge, BerliOS and many more	Gitarious, GitHub, repo.or.cz
Webinterface	yes	yes	yes
GUIs	many	many	some
Third party support	excellent	very good	bad

Still CVS is not able to create a diff of the binary file. For every change in a binary file the entire file needs to be uploaded and stored on the server. This leads to rapidly expanding repositories.

- When CVS was developed, almost every commit consisted of only one file. It was sufficient at the time and there was no need for *atomic commits* (all or nothing is transferred and stored, see Section III-B4). Today, software projects in general are bigger and commits often consists of more than one file. This makes atomic commits necessary.
Without the capability of atomic commits a part of

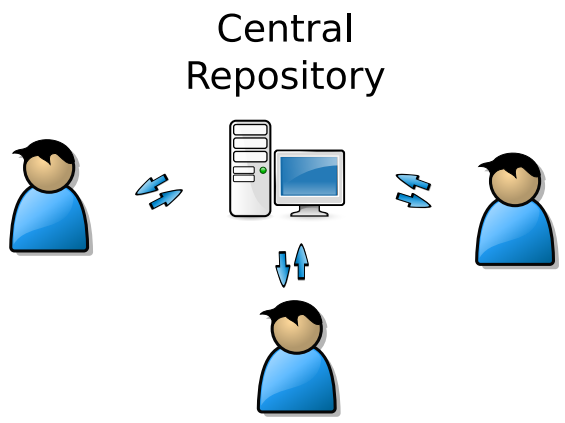

Fig. 1. A Central repository in a Centralized Environment

a commit can be blocked by the need to merge conflicts. While some files of the commit are already in the repository, some are not. Commits such as these can cause repository inconsistency or corruption. While user A tries to resolve the conflict, user B can make his own commit into the inconsistent repository. His changes may conflict with the partial commit of user A. In CVS there is no way to determine if the repository is in an inconsistent state.

- Tagging is expensive. By tagging some metadata is added to all files, the bigger the directory, the longer the operation takes. Because branching is realized though the tagging mechanism, branching is expensive as well.
  Furthermore, many users criticize the concept of branches and tags to be confusing.
- CVS does not support the moving and renaming of files without loosing the history of the files.

Even though CVS was widely used and enabled a workflow that did not exist before, with all its shortcomings it was only a matter of time until other projects tried to overcome CVS. Today, most of CVS's founders and ex-main developers use CVS's successor Subversion.

B. Subversion

1) History: Subversion (SVN) was planned as a free successor of CVS [7] [8]. It is basically CVS without its shortcomings.

SVN was initialized by CollabNet in 2000. CollabNet offered a collaboration software suite called CollabNet Enterprise Edition which included CVS. Because of the limitations of CVS and CVS's domination of the market at that time, CollabNet decided to write a system from scratch that is like CVS but without its problems. The majority of the users believe that CollabNet succeeded in their goal.

The first version of SVN was released in August 2001 in a self-containing SVN-repository. Today SVN is a normal FOSS project maintained mostly by volunteers, but CollabNet still funds the project.

2) Differences to CVS: The initial version of SVN already eliminated the major flaws of CVS and also improved the usability.

Here is an incomplete list of the changes compared to CVS.

- SVN does not use RCS. It was implemented using Berkeley DB (see Section III-B4) as file system for the repository (see Section III-B3). Ever since version 1.1, another file system has been officially supported: FSFS (see Section III-B5).
- Atomic commits: there are no longer partial commits like in CVS. Either the entire commit is stored in the repository, or nothing is.
- Renaming and moving of files and directories is possible without loosing the history (but there are still some limitations).
- Binary files are fully supported and only the delta is transfered into the repository. Binary files are recognized automatically.
- Cheap tagging and branching: although SVN has no notion about branches/tags it can make copies. Copies work as branches/tags because they are stored in branch/tag directories and are treated as branches/tags.
- Revision numbers apply to entire trees, not just files. That implies that revision n and $n+1$ can be the same.
- With version 1.5 SVN gained the capability of semi-automated tracking of branching and merging. This improved SVN's merging dramatically (see Section III-B8).
- More disconnected operations are available (see III-B6).
- There are many more differences such as interactive resolution of file conflicts, partial checkouts and path-based authorization controls.

3) Design: The limitations of CVS's fileformat RCS inspired the developer of SVN to use a highly modular design for SVN.

SVN consists of a number of libraries divided into three layers:

1) The *client layer* is on the client side. The user interacts through the client layer with the repository.
2) The *repository access layer* is responsible for the different methods of accessing the repository. It is located on the user's computer.

3) The *repository layer* manages the file system of the repository on the server side.

Every library has a clean interface and a well defined purpose.

`libsvn_client` is the interface to all client programs. SVN's standard client program `svn`, as well as most GUI-clients use this libraries.

The user needs to do two things: manage the working copy and somehow communicate with the repository. `libsvn_client` uses `libsvn_wc` to manage the working copy.

By communicating with the repository the user leaves the client layer and enters the repository access layer. `libsvn_ra` abstracts from the way of communicating with the repository. Depending on the protocol `libsvn_ra` loads different libraries: `libsvn_local` for local access to the repository, `libsvn_dav` for WebDAV access, `libsvn_serf` for an experimental WebDAV implementation, or `libsvn_svn` for SVN's own protocol. All libraries, except `libsvn_local`, require network access. The network libraries require additional libraries. The HTTP protocol requires the Apache webserver to load some libraries (e.g `mod_dav_svn`) in order to communicate with the repository and to allow public access. The SVN-protocol needs a running SVN deamon, svnserve, to access the repository.

In the repository layer `libsvn_repos` defines the interface and the logic of the repository. `libsvn_fs` abstracts the file system. It is a module loader like `libsvn_ra` and enables SVN to use different file systems for different purposes. Currently `libsvn_data` (the Berkeley DB) and `libsvn_fs` (the file system FSFS) are supported and shipped with SVN. The file system is not a real file system, it is virtual.

`libsvn_fs` makes it possible to replace the shipped file systems with another one. Google, for example, implemented for its FOSS hosting platform 'Google Code' [9] its own proprietary file system. It scales better on Google's servers than BDB and FSFS.

SVN's design is very different from the design of CVS. CVS was built upon RCS, whereas SVN was designed to be very modular.

4) Berkeley DB: Berkeley DB (BDB) is a small, high-performance embedded database [10]. It was developed at the University of California, Berkeley, and has a long tradition in the UNIX world. It was turned into a commercial project by Sleepycat Software and later bought by Oracle Corporation.

> "The Oracle Berkeley DB family of open source, embeddable databases provides developers with fast, reliable, local persistence with zero administration. Often deployed as "edge" databases, the Oracle Berkeley DB family provides very high performance, reliability, scalability, and availability for application use cases that do not require SQL." [11]

The database does not support SQL, but is accessible by its own internal API. BDB guarantees the ACID properties:

- Atomicity: 'all or nothing', a transaction is done completely or not at all.
- Consistency: 'keep consistent DB consistent'.
- Isolation: solve concurrency conflicts that may arise with concurrent operations.
- Durability: 'now and forever'.

SVN's commits have a higher scope than BDB's commits. That means that BDB's atomicity properties do not lead directly to atomic commits in SVN. Nevertheless, they ease the implementation of atomic commits.

BDB is capable of backups at runtime (called hot backups).

Some people criticized SVN for the BDB storage system. Harmful SVN crashes can lead to a non-recoverable repository. In contrast, RCS files are human readable and can be fixed with a simple text editor relatively easy.

Ever since version 1.4 of SVN (or version 4.4 of BDB) BDB has gained the ability to automatically recover the database after a crash.

BDB has some portability issues. It runs under different operating systems, but it is not possible to transfer a BDB repository that was created on Windows/one CPU architecture to UNIX/another CPU architecture and vice versa.

5) FSFS: SVN 1.1 introduced a new repository storage system: Fast Secure File System (FSFS). It is not a database, it is a flat-file file system.

The way SVN uses FSFS also guarantees the ACID properties for SVN operations. Most important of which is, the atomicity property: incoming transactions are stored in a temporary directory and are moved into the repository directory later.

In comparison, FSFS is more flexible than BDB. The storage format can be transfered to different OSs and CPU architectures. FSFS has no database overhead which results in smaller repositories.

Today FSFS is the standard fielsystem for SVN.

6) The .svn: Much like CVS, SVN provides for every directory in the working copy a hidden directory named `.svn`. The content of `.svn` is supposed to be changed with `svn` commands only, not by hand.

The `.svn` directory contains:

- Which files in the working copy represent which files in the repository.

- A prisitine file of every file in the working copy.
- User-defined properties.
- Which revision of each file and directory is present in the working copy.
- Some more metadata.

The unchanged copy in `.svn` enables the user to run certain commands without network access (like `svn diff`, `svn status` and `svn revert`). It also allows `svn commit` to send only the diff instead of the entire changed file. The bigger the files get, the more important this feature becomes.

7) Commit and Update in Detail: As already mentioned, in the `.svn` directory there are unchanged copies of each file in the working copy plus information about which revision the file is based on.

With this information SVN can determine in which state the file is in and what needs to be done.

- *Unchanged and current*: commit and update will not do anything because there are no changes.
- *Locally changed and current*: update will not do anything. Commit will publish the changes.
- *Unchanged and out of date*: a newer revision is in the repository. Update will download it. Commit will not do anything.
- *Locally changed and out of date*: commit will fail because it is out of date. Update will download the new revision and try to merge it. If there are conflicts the user needs to merge these.

8) Merging: Before version 1.5, merging just applied the changes of two versions to a revision. `svn diff v1 v2` shows the differences between two files. `svn merge v1 v2` applies the differences to the current revision.

Merging therefore was hard because the user needed to figure out which revisions were important. Important are all revisions that changed the code of the branch. Unfortunately, SVN did not keep track of branches and merges. Applying the same change twice resulted in two instances of the change the current revision.

Version 1.5 introduced merge tracking. This means that already applied changes are not included a second time. The changes simply get skipped. Internally this feature is very complex and therefore is not described here.

9) Workflow: The workflow for CVS and SVN is very similar and only differs in detail. No problem arises when applying the SVN workflow to CVS.

In companies there is one repository for each project. It is normally divided in trunk, tags and branches. The users commit into the trunk in which the main development takes place.

Some companies have strict commit policies. The user is not allowed to commit his changes until the changes pass the test suite. Depending on the time it takes to complete the test suite the user is simply not able to commit every change. He needs to wait until all changes finally pass the test suite. This results in one big commit instead of many small ones. A big commit can change the mainline in a way that other developers could have problems to integrate their changes.

To avoid these problems to a certain level, the user can start a branch. The user does not break the trunk by committing, only his branch. He can continue to develop and commit into his branch. When all changes pass the test suite, he can merge the branch into the trunk. In theory this model works well, but before version 1.5 the merging process was very laborious and needed careful planning.

SVN 1.5 and its automatic merge tracking simplifies this workflow. The user does not need to plan the merge in detail anymore.

A typical setup in the FOSS world looks like this:

- There is a central repository, which is, for example, hosted at Sourceforge.
- The core developers have read-write access to the repository.
- Everybody else, including users of the FOSS program have only read access to the repository.

In contrast to companies which normally have high speed LAN the repositories in the FOSS world are only accessible though a slow internet connection.

The workflow for the core developers is very similar to the workflow for developers in companies.

For users who want to contribute the workflow is radically different. They can checkout a working copy but they are forced to write the entire feature or patch in one session without being able to commit their work. Depending on the size of the change it can be hard to reintegrate the change back into the repository. The core developers prefer to review a series of small commits instead of one big one. However, the workflow in centralized environments amplifies big commits.

Another issue in the FOSS world is whom to grant commit access. A commiter can cause damage to the repository, if accidentally or on purpose does not matter. It is hard to determine if a person is trust-worthy or not. This leads to small group of users with write access.

This shows that the centralized model divides the community into two parts: the minority with commit access to the repository and the rest who has only read access. This it at least a psychological barrier.

Figure Figure 1 shows the typical set up in centralized environments.

C. Git

1) History: Git is probably the most famous and most widely used DVCS [12] [13] [14] [15]. It was initiated in 2005 when the development process of the Linux kernel lost its VCS.

The Linux kernel is one of the biggest FOSS projects and the rate of code changes is very high. Since 2002 the kernel developers used BitKeeper for tracking the Linux kernel. With this method the development scaled better than before.

BitKeeper is a DVCS under a commercial license. However, BitMover, the development company of BitKeeper, published it under a license that made it possible to use BitKeeper for the Linux kernel development.

On April 6th 2005 BitMover dropped the free license. On all of a sudden, the Linux kernel no longer had a VCS.

Linus Torvalds was looking for a VCS which fulfilled the following requirements. It must

- be reliable,
- have a good performance,
- be distributed, and
- not be like CVS [14].

After testing most VCSs, he decided that none of the VCSs fulfilled the demands of the Linux kernel development. He started to write his own VCS.

On April 18th, after only two weeks of development, Git was published in a self-containing repository. It has been officially in use for the Linux kernel development since June 16th 2005.

In the early stages, Git was hard to use. Version 1.5, which was released in February 2007, brought major usability improvements.

Today Git is maintained by Junio C Hamano.

2) Git's Properties: Here are some properties of Git:

- Unlike most VCSs, Git does not use delta encoding to store files, it stores snapshots of all files in a tree structure (see Section **??**).
- Git tracks content, not files. If only one file changed, the entire repository changed. Therefore it is not easy to track single files.
- Git can import complete repositories from many VCSs (CVS, SVN and more). It can completely replace a SVN client (`git svn`). Therefore it offers some of the distributed features in a centralized environment.
- Git has only a few commands that need network access:
  - `git clone`: initiates a repository from a server,
  - `git fetch`: fetches all revisions the user does not have yet from a server. It does not change the local branches,
  - `git pull`: is a `git fetch` followed by a `git merge`. It merges the downloaded changes into the current branch.
  - `git push`: pushes the changes the user made into the server repository.

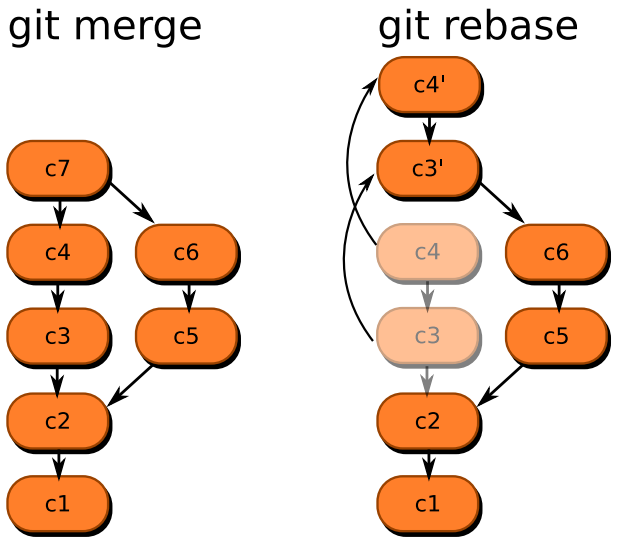

Fig. 2. Merging and rebasing in Git

3) Merging: Branches and merges are a central concept in Git. Branches in Git are not stored in directories like in SVN, they are stored in virtual containers. The user only sees the branches by using the Git commands: `git branch`.

SVN's improved merge capability was added on top of SVN. In contrast, Git was built with the focus on fast and easy branches. Because Git is aware of its history and tracks branches and merges Git's merging capability is very good. It does not need to access the network to get all needed revisions for a merge, because all revisions are local. This results in very fast merges.

Git also offers a alternative to the classic merge: the *rebase*. It applies the commits of branch A on top of branch B instead of just merging them. A linear history is created instead of a parallel one (see Figure 2).

4) Plumbing and Porcelain: Git follows the Unix tradition: 'do one thing, do it well'. This is why Git includes about 150 commands. Most of these are so called *plumbing* commands which are low-level commands. The rest of the commands are *porcelain* commands. They are high-level commands with a more intuitive interface. The user only needs the porcelain commands to use Git.

Internally the porcelain commands use the plumbing commands. In some cases it can be very effective to refer

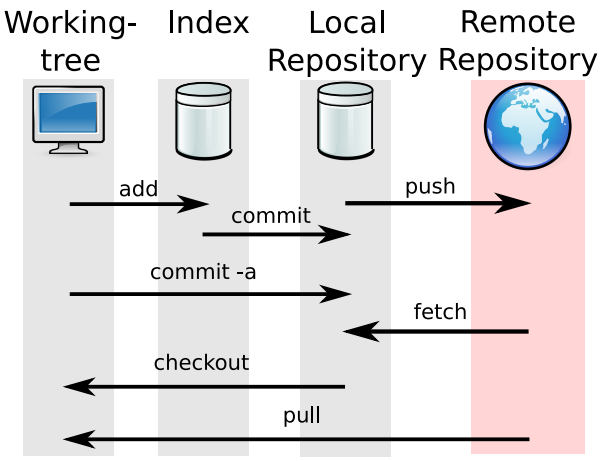

Fig. 3. Repository structure in Git

to the plumbing commands to do certain tasks.

5) The SHA1 Hash: For this section, think of all tracked content of Git as objects (see Section III-C7).

The *SHA1 hash* (SHA1), a cryptographic hash function, is a core feature in Git. An SHA1 string is a string of 40 characters. It identifies and ensures the consistency of tracked content.

Objects are stored with their SHA1 string as their name. Two objects with the same name have the same content and vice versa, objects that are different do not have the same SHA1.

The SHA1 concept allow to easily determine if two objects are the same by comparing the name. It also offers an easy integrity check of objects. If the name of an object does not equal the SHA1 of the object the object is broken.

Git does not use integers for the revision numbers, it uses the SHA1 string of the objects as revision. Revision numbers that increase by one at each commit do not work in a distributed environment because of Git's non-linear nature. Locally, integer revisions would work, but by accessing the network conflicts would arise. Numbers might not be unique anymore. The SHA1 method leads to unique revisions and therefore identifies every objects.

It is not necessary to refer to the complete 40 character long SHA1, a partial SHA1 works too. The only condition is that it must be unique. Normally a partial SHA1 of seven bytes is long enough.

6) The Repository Structure: There is only one hidden directory for every repository: `.git`. It is situated in the root directory of the project. No additional hidden directories clutter the project directory.

Git's repository structure consists of four parts:

1) The working copy in Git is called the *working tree*. It has the same function as the working copy in CVCSs.
2) The *local repository* is, like in CVCSs, the place where all files including history and metadata are stored. It is situated in the `.git` directory.
3) The *index* is a staging area which contains everything that is going to be committed to the local repository with the next commit. It is situated in the `.git` directory.
4) The *remote repository* is just a copy of the local repository. It is stored on a server for public access. This part is optional.

Because of this design, the granularity of committing in Git is higher than in SVN.

Figure 3 the illustrates structure of the repository and the semantics of some common `git` commands.

7) The Object Model: Internally, Git handles four basic object types: *blob* (binary large object), *tree, commit and tag*. These objects are immutable and are stored in the object directory `.git/objects`. The name of each object is the SHA1 string of the content of the file.

Every tracked file is represented by a blob. More precisely, a blob is the compressed content of a file plus a header. The filetype does not matter.

A tree consists of a list of its contents which can be either blobs or trees. The blobs and trees in the listing are represented by the name of the objects, the SHA1.

Commits are a pointer to the root tree of the repository plus some metadata consisting of a commit message, author plus date, commiter plus date, parents (the previous commit) and the initial pointer to the tree.

Tags are simply pointer to an object plus some metadata, consisting of a tag message, name of the tagger, name of the tag, and the tagged object (mostly a commit).

There is a fifth object that simplifies the life of the user: the *reference*. References are simple movable lightweight pointers to a commit. It is just a file with the SHA1 (the object it point to) as content. References keep track of where the HEAD, the remote repositories, and tags are. They are stored under `.git/references`.

Because of the combination of the object model and the SHA1 concept, an SHA1 of a commit determines all included trees, files, branches that got merged, and the entire history. This is useful if a user accidentally deletes/looses his project and can find a repository in the internet with the same SHA1. He can be sure that it is the same project with the same history.

These five objects form a directed acyclic graph (see Figure 4).

8) Loose and Packed Objects: All objects are stored in `.git/objects` and accessible by its SHA1-name. Objects can be either loose or packed.

Loose objects take up more space. The SHA1-name gets split up into two parts: the first two characters

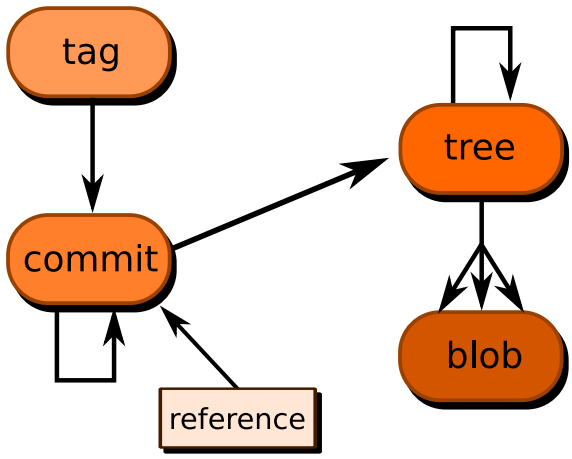

Fig. 4. The four basic objects in Git

are used for the directory, the rest specifies the filename. For example, the loose object with the SHA1 bf07cb62777644bf1ab0da1d5a16938462545d is stored under `.git/objects/bf/07cb62777644bf1ab0da1d5a16938462545d`.

When the number of objects reaches a certain limit the garbage collector starts packing the loose objects. Packed objects are stored under `.git/objects/pack`. There is always a `.pack` file containing the objects and a `.idx` file containing an index for performance reasons. The user can also invoke the garbage collector manually with the command `git gc`.

9) Workflow: All CVCSs have similar workflows. In contrast, DVCSs are very flexible and one can adjust the set up of DVCSs to fit ones needs.

First of all, there is not necessarily a need to setup an external server.

Most users have a branch for the main development (like trunk in CVCSs) and branches for certain topic ('feature X').

A very common model is to have an additional shared repository for each project (see Figure 5). Every users has his local repository in which he commits his changes. They also have write access to the shared repository. To ensure they stay in sync the users push their commits from the local into the shared repository. This model is very similar to the model in centralized environments but adds the advantages of local repositories.

Quite a few larger projects work by the *lieutenant-dictator model*, e.g. the linux kernel (see Figure 6). Each developer only cares about his subsystem. The lieutenants who are in charge of this certain subsystem forward the files to the dictator. The dictator only has to deal with the lieutenants and not all users. This system works by the network of trust principle. Each user only pulls from users he trusts.

In the end the dictator publishes all files in his public repository and everybody pulls from there.

Another model is the *integration manager model* (see Figure 7). Every user has his own local and public repository. All users develop on their own and publish their work in their public repositories. They then ask the manager to integrate their changes. The manager can cherry-pick the commits he wants. He integrates the changes and publishes the new revision in his public repository. Everybody pulls from there.

The scenario from Section III-B9 applied to the distributed world looks like this. The strict commit policy of the company still restricts commits to the central or shared repository, but the users are able to commit to their local repositories. They document their work by many small commits.

In the FOSS world it looks like this. Even though, there are still certain users that have write access to the main repository, DVCSs do not split the FOSS world into two parts. Everybody can get the repository and can commit changes to the local repository. This workflow leads to patches that are divided into smaller parts and therefore are easier to review and integrate. Git also makes it possible tune the commits, to divide and merge commits afterwards, with the goal to get a logical unit. These logical unit are even simpler to review than normal commits.

IV. Comparison

Table I shows the main properties of CVS, SVN and git.

SVN is replacing CVS. It surpasses CVS in almost every discipline: it is faster, very extendible, gains valuable features (e.g. merge tracking) and, most importantly, does not have the shortcomings of CVS.

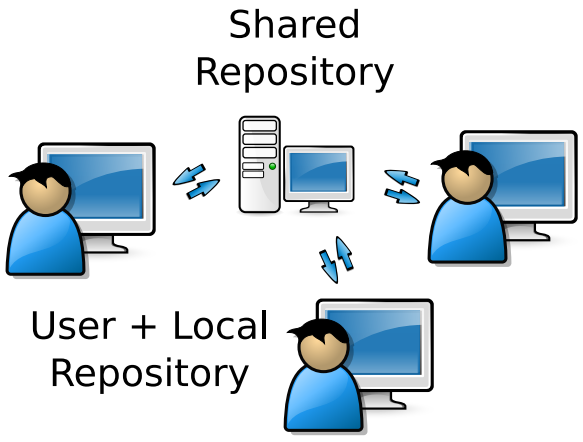

Fig. 5. Shared repository in Git

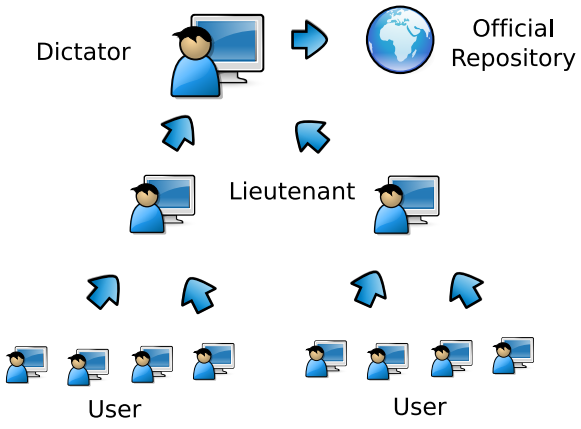

Fig. 6. Lieutenant-dictator model in Git

The only category CVS can beat SVN in is the third party support. The third party support for SVN is very good but CVS has been on the market for about two decades. Countless companies developed programs to simplify and extend working with CVS. If the user does not depend on a certain third party product there is no reason to use CVS anymore.

CVS has strong support in the FOSS community. Many public hosters for FOSS projects are available, such as sourceforge, Google Code and BerliOS.

Git introduces some new and interesting features: local repositories, strong support of branching and merging, and high performance. These features enable new workflows that can be easily adjusted to ones needs. It is also possible to keep the old workflow (central repository) and extend it a little bit (local repository).

Git is a relatively new program so not many third party programs are available yet. However, the git community is making big strides towards the mass market. Several GUIs like *Git Extension*, *GitX* and *QGit* are quickly gaining maturity and making git available for a broader spectrum of users. Many big FOSS projects just recently migrated to git: Perl, Ruby on Rails, Android, WINE, VLC, Fedora and many more. Meanwhile public hosters for FOSS projects are available such as repo.or.cz, Gitorious and GitHub.

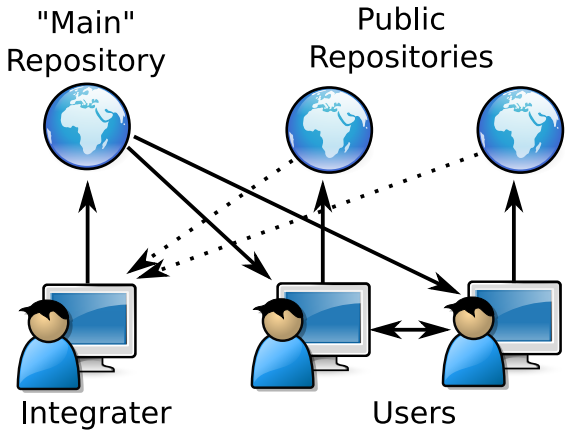

Fig. 7. Integration manager model in Git

V. Conclusion

There is no doubt that VCSs simplify software development and every project should use a VCS.

CVCSs like SVN have already proven to work and accelerate software development. SVN's concepts are simple to understand and easy to apply. The third party support is very good. However there are some limitations:

- the inability to work offline,
- the network is always the bottleneck,
- the weak branching and merging capabilities in comparison to DVCSs (however SVN improved), and
- FOSS projects are split in two parts: commiter vs non-commiter.

Git's non-linear nature needs some more thinking, but, in return, offers some great improvements. It nullifies the limitation of CVCSs.

By using git (or any other DVCS) FOSS project can improve their workflow and be a little bit more 'open'. It makes it less difficult for normal users of the program to contribute to the development.

In commercial companies the ability to commit locally can improve the development as well. An implementation of a feature can consist of a row of a small logical unit (commits) instead of one giant commit.

For single developers, git (or DVCSs in general) is clearly the first choice. The setup is trivial compared to SVN.

DVCSs with their valuable features can definitely be an extension of CVCSs and therefore should be considered as substitution.

References

[1] Cvs homepage. [Online]. Available: http://www.nongnu.org/cvs/
[2] M. B. Karl Franz Fogel, *Open Source Development with CVS*. Coriolis Group, 2001.
[3] D. Grune. Concurrent version system cvs. [Online]. Available: http://www.cs.vu.nl/~dick/CVS.html
[4] Grune, "Concurrent versions system, a method for independent cooperation," 1986.
[5] B. Berliner. [Online]. Available: http://www.brianberliner.com/2006/09/12/ask-brian-cvs-open-source-and-startups-today/

[6] P. Inc and M. D. Blvd, “Cvs ii: Parallelizing softwaredevelopment brian berliner,” 1989.
[7] B. F. Michael Pilato, Ben Collins-Sussman, *Version Control with Subversion*, OReilly, Ed. OReilly, 2008.
[8] Subversion homepage. [Online]. Available: http://subversion.tigris.org/
[9] Google. Google code. [Online]. Available: http://code.google.com/
[10] Oracle, “A comparison of oracle berkeley db and relational database management systems,” Tech. Rep., 2006.
[11] Oracle berkeley db. [Online]. Available: http://www.oracle.com/technology/products/berkeley-db/index.html
[12] Git - fast version control systems. [Online]. Available: http://git-scm.com/
[13] R. Schwartz. (2007) git. Google Tech Talks.
[14] L. Torvalds. (2007) Linus torvalds on git. Google Tech Talks.
[15] T. Swicegood, *Pragmatic Version Control Using Git*, 2009.