Distributed Version Control Systems

Dipl. Medieninformatiker (BA) Daniel Kuhn
Master degree course Computer Science and Media
Hochschule der Medien
Nobelstrasse 10, 70569 Stuttgart
e-mail: dk047@hdm-stuttgart.de

July 2010

Abstract

Traditional centralized Version Control Systems (VCS) as e.g. CVS and SVN, which were used since the 1980's in almost every company, are facing competition by a new approach to source code management. Distributed Version Control Systems like Bazaar, Darcs, Mecurial and Git are changing the way repositories are stored and distributed by representing a new and more contemporary mindset. Today Distributed Version Control Systems have therefore become very popular in the open-source community with many developers using them.

This paper aims to show the requirements, advantages and disadvantages of Distributed Version Control Systems and discusses the differences to Centralized Version Control Systems us the software Git as an example. The document shows and explains the functionality of the system and takes a look under the hood by not focusing on the use of the tool, but by rather analyzing how the tool handles its tasks.

1. Introduction

Since the 1970's Version Control Systems have been used to archive and manage different versions of files, especially source code. Version Control started in 1972 with the release of the first system called Source Code Control System (SCCS). SCCS was developed to manage source code, configuration-files or documentations and focused on managing single files and single users. SCCS has brought features like: letting developers keep track of the history of a file, using checksums to detect data corruption, using a forward delta system and providing support for binary files [3] [4].

Another early version control system is the Revision Control System (RCS), developed in the 1980's. Just like SCCS, RCS is a file-based, single-user oriented version control software that simply stores different versions of files in the file-system. But unlike SCCS, RCS keeps no track of the history of a file and does not provide checksums for integrity checks. Both SCCS and RCS are Version Control Systems that use a locking mechanism to lock files for other users when they are edited.

1989 the more familiar Concurrent Versions System (CVS) was developed as an improvement of RCS. CVS showed many additions like the support for multiple files and the support for multiple users. But the major advantage of CVS was support for merging concurrently changed files instead of using a locking mechanism. This enabled developers to work collaboratively on their projects. But CVS had some restrictions concerning the handling of directories and binary-files. With the growth of the internet, users of CVS criticized the lack of supporting a native remote access in CVS and demanded for a solution. So in 1994 a group of developers implemented an extension to use CVS over TCP, which was a large improvement to CVS, which could now be used easier over the internet.

Later in the year 2000 a company called CollabNet decided to develop a replacement system for CVS. So Ben Collins-Sussman, Brian Behlendorf and Jason Robbins of CollabNet started the Subversion (SVN) Project. The subversion CVS is still being used today and considered an industry standard. Subversion is an Open-Source Software in accordance with the Debian Free Software Guidelines (DFSG)[1]. Subversion brought many advantages to developers like better file-handling, including a directory structure, which lead to more focus to the project instead of only files. SVN has real atom commits ensuring consistency, better branching and tagging and an abstract repository interface for an easy development of different access methods. [5]

As of today, Centralized Version Control Systems - especially Subversion - are the top-dogs in companies. With Bitkeeper, developed in 1997, there was another approach to source control management: Distributed Version Control Systems. These systems have become more and more popular with the development of Git, Mercurial, Bazaar and Darcs.

[1]http://www.debian.org/social_contract#guidelines

Git is a Distributed Version Control System (DVCS) in development since 2005 as an open-source project started by Linus Torvalds. The Git project was started with the intention of replacing the proprietary Bitkeeper DVCS used to maintain the linux kernel and free to use for open-source projects. After the Bitkeeper developers decided to change their licensing policy Linus Torvalds project leader of the linux kernel, started looking for an appropriate replacement, which he ultimatly couldn't find. Therefore Torvalds decided to design the Git DVCS to meet the needs of the kernel and its developers.

2. Polymorphism of Version Control Systems

One of the easiest "versioning systems" is local and manual versioning. Everybody has used it, maybe without even thinking about it as a kind of version control. Local or manual versioning means to make a copy of a document and save it under a different name (e.g. myImportantLetter_20100202). This version control system does neither perform well, nor is it efficient or safe in any way. But it is easy to use and understand and may fit personal versioning. However when it comes to a professional level a better version control system is needed. So it becomes quite obvious that version control can be done in many ways and should always fit the use cases by fulfilling its requirements.

One of the common purposes of Version Control Systems is to enable groups of people to work together on (any kind of) files. They can change, archive, merge, branch or synchronize files of which the system keeps a history. Another purpose is archiving and backupping. Content in the system should possibly be safe and free of data-loss. Combining the two aspects of collaboration and security leads to Centralized Version Control Systems (CVCS). Traditional systems like CVS and more common SVN are centralized, containing only one main repository per project usable by every member of the group. CVCS are used in most of the companies and therefore have reached their maturity by being considered an industry-standard. Aside from CVCS there is another approach to achieve version control in a slightly different way. Distributed Version Control Systems like Bazaar, Darcs, Mercurial and Git use decentralized repositories for versioning data which is spread among users.

2.1 A quick comparison between centralized and decentralized VCS

First I'd like to give a more general view of centralization and decentralization. Comparing the systems most of the time leads to the same problems. Not only for Version Control Systems, but for almost every system sharing data between nodes over distances. When it comes to data storage and delivery one has to think about general questions like

1. Security – is my data safe?
2. Performance – can I deliver the data quickly?
3. Reliability and Availibility – can the data be accessed 24/7 without downtime?
4. Consistency – is my system state consistent over every node?
5. Maintenance – can I maintain my infrastructure easily?

These five points are the general requirements of a version control system. However some of these requirements are orthogonal to each other. A system which is performing well, highly secure, highly reliable, easy to maintain and always has a consistent system state is hardly possible to design without spending a tremendous amount of money.

Fact is, if you use a centralized system, it is easy to backup the data and maintain the servers and storage systems. But when it comes to scalability and distribution, centralized systems will not perform well – which can also lead to general failures. On top of that there is always the risk of a single point of failure.

The same questions arise when comparing CVCS with DVCS. A centralized system will always be easier to maintain. Servers are mostly in one place and contain many repositories for multiple projects. This makes it easy to backup the whole data in comparison to a distributed system. DVCS are taking another approach on the backup aspect and consistency. The idea is that if you clone a repository to your local drive, it contains all the history (including branches and merges) from that one developer who's repository was cloned. But you never see the code of all developers working on the same project. A developer can implement a feature and say "hey I've got the feature ready, pull the code if you like" – and if a developer likes to pull that feature, he does and merges the code right into his repository. So there is not just a single development version, like one might be used from CVCS, but rather multiple versions of the code. The result being nobody knows the whole system state. This approach to development involves a change of the mindset used in the traditional centralized systems, because Distributed Version Control Systems focus not on backup, but rather on easy versioning and the sharing of features to a program instead of having only the one version.

Linus Torvalds (in Google tech talk [7]) sees the fact that everybody pulls from everybody as an advantage, because your code is distributed and backuped by other developers and you don't have to care about it. But I disagree. This may work for famous developers like Mr. Torvalds, but perhaps not for the "nooby-open-source-project-contributer". But even if everybody pulls your code, this involves only the code of a feature that you released and not your code you are working on in your development branch (which is only stored on the local harddrive). If you encounter a data loss on your machine and you don't have a local backup, your progress is lost because nobody pulled from you – yet.

In fact, the same can happen when working with a centralized repository, but if you use a development branch in the central repository you can commit unfinished code without interfering with other developers (like in local branches in DVCS) and without losing your progress in a machine crash. And assuming that the centralized repository gets backuped, and the backups are stored in a safe way, the probability of data-loss is almost 0%. Sure if you haven't backuped your central repository and you suffer a crash on the server, your data is lost, too. But as mentioned before, backupping a central server environment is not so much up to a developer, as it is in a distributed environment.

But thats a common problem in distributed systems. On the one hand you want to know the status of your system, including all nodes. On the other hand if you want to have the knowledge of the system-state, you need one central node with all nodes knowing about it or many messages going between all nodes, so that every node knows about the state. This again leads to other problems how data is passed, how nodes know about each other, problems with latency and dying nodes, etc.. The DVCS abandon this need of knowlegde of the nodes by not doing it via software. Instead every user trusts another user, which builds a "ring of trust". Besides, like in a centralized environment it is necessary to have one or more persons in charge of the release-management, who decides what comes in the new release and who has to know who has developed the features.

2.2 Requirements in different roles

Ian Clatworthy in his Paper "Distributed Version Control Systems - Why and How" [6] describes different perspectives on software development and therefore the requirements of a version control system depending which role one is taking. He sees four views: the developer view, the release manager view, the community view and the senior management view.

Developer: A developer sees the operations he can do better and faster like working disconnected, branching and merging easier just as working together more easily with co-developers.

Release Manager: The release manager has an interest in how the software gets packaged and which features are in the new version. In the ideal world of a release manager, one would build "each new version by picking and choosing [the features] like a lego brick".

Community: The community view addresses the individual developer as the source of the whole project and therefore the need to combine and motivate more developers to contribute to their project. The view is less focused on the features, but more on the mindset behind the distributed working. As described in the paper, Mark Shuttleworths (CEO of Canonical Ltd. developing Bazaar) driving motivation for developing a DVCS is its "positive impact on open source communities" by using a more like "wiki approach": Which means in an open source project every newcomer won't be restricted and thus is able to do changes and – if the code and feature is good, somebody is going to pull it from you and it might end up in the release.

Senior Manager: the senior manager view goes to personnel. The open-source communities have proven that it is possible given "the right people, the internet and the right toolset" to make great projects reality by being spread around the globe. This means that using DVCS new ways of building companies and teams rise in the means of outsourcing, offshoring, agile and home-office.

If one is looking at these four roles, there are different requirements. The developer wants to work fast and easy, the release manager wants to build his release easily, the community wants to share their code and combine it easily and the senior manager wants to have fast and productive employees. So if you now consider which Version Control System is best for which requirement you see, that there is not the single one solution, but rather the right solution for the right purpose and maybe even the combination of two solutions. A developer might be better off with a DVCS, but in contrast he may have a higher training period and needs a deeper understanding of Version Control, than using for example SVN, which may make the senior manager unhappy. A Community wants to share their code, but when it comes to the release it may be easier to have a central repository and the knowledge about every feature. Combining DVCS for developers and CVCS for the release management could be a solution worthy of further investigation.

1. Maintenance – A centralized system will always be easier to maintain. Servers are mostly in one place and containing many repositories for multiple projects. Distributed Version Control Systems are maintained by every developer and not by a single system admin.

2. Backup – centralization makes it easy to backup the repositories in one place and store them in a safe way. Decentralization spreads copies of the code (partly or entire) to every developer which might have a local copy. This makes the repository "eventually backuped".

3. Acceptance and Support – CVCS are considered an industry standard and have been in use for more than 20 years. It is clear that there is today more tool-support for centralized systems than for decentralized ones – but latter support is steadily increasing

growing. Additionally centralized systems are easier to understand and more appropriate for a one-place-development company, but If you look at large open-source projects a centralized system in one place might have disadvantages in speed and availability

4. Release Management – In CVCS it is easier to release a new stable Version. The release could be done by everybody who has access to the repository. In DVCS forming a release is more complicated because there is no "single one" version. For a release someone has to be the release manager who pulls the code from the developers, merges it, tests it and releases it.

2.3. Social aspects and security

In a company it can be difficult to build and maintain an access control infrastructure. You have to grant and revoke rights to systems and servers for every single developer – including the version control system. This can quickly lead to complicated access control policies, without knowing who has access to which system. As an example: I work at a company with only one sysadmin, who has to maintain the whole infrastructure including the creation and maintenance of CVS rights and key management.

In large companies there may be more restrictions to "who writes and who don't" than in a middle-sized company. So maybe you don't want do give every developer in you company write access, so you are picking some developers who do a great job an give them the rights. But if you make this circle to small, you may get problems with other developers. If you make it to big, you have too many people that are committing and interfering with each other. Same problems are more concerning in open-source projects, because you may not give a new contributor complete write access to the repository.

So in distributed systems you don't have to care about commit access. You always have commit access to your repository, you can branch, you can merge and when your work is done you can tell everybody to pull from your repository. You don't need a sysadmin to give you access to repositories – the whole distribution process and "rights-management" is done by the developers amongst themselves. So no special write access and no politics is needed.

In a distributed system "security" and code reliability are archived through a network of trust. Every developer trusts a few other developers from which they are pulling from. And these people, trust other persons, and so on. So trust means you trust the decisions of other developers implicitly including their trust to others.

2.4. Summary

Searching the web for a good and objective comparison of centralized and decentralized CVS sometimes looks like there is a kind of religious war going on between the supporter of each party. The DVCS worshippers blame CVCS to be the devil in the source-control management universe and see DVCS as the only way version control should be done.

Personally, I have been using MS Visual Source Safe and SVN now for quite some time, and I am happy with SVN. But when I heard of distributed systems especially Git, I got curious. Now, after a closer look, I can say that I really like the way how Git gets work done, how it handles the data and how easy it can be used. But I don't think that Git will relieve SVN of its duty, like it is often announced in the web. I rather think, that Git and SVN perfectly can run side by side. Git is a good way for large distributed projects (especially open-source projects) and developers who want more. And SVN works great in a company having few locations or even only one, by providing an easy to understand, fast to learn and easy to use version control without fancy features, which fits the need of most developers. Combining the advantages of a centralized system with the decentralized ones, by using Git for every developer and SVN for the release repository, where all stable and beta features are managed, is in my opinion a useful setup – considering that Git provides a native SVN adapter. It allows developers to have a performant, available and consistent system by using the advantages of a central backuped, easy and reliable system.

Requirement	DVCS	CVCS
release management	O	X
usage	O	X
training period	O	X
featureset	X	O
performance	X	O
data security	X	O
maintenance	O	X
backup	X	X
collaboration	X	X

Figure 1: Easy feature comparison between CVCS and DVCS

3. A Technical View on Git

3.1 Requirements

By starting with no dedicated Version Control Software at all, the linux kernel was maintained the first 10 years by using tarballs and patches, followed by the use of the commercial Bitkeeper software. Even with no Version Control

System, by using patches, there was a decentralized character of how the patches were distributed, and like Bitkeeper, Git was designed with a focus on a decentralized development and source code management. But from a technical view Git is very different from Bitkeeper. A lot of the flows used in Git, came directly from the workflow learned using Bitkeeper. The design and implementation of Git was focused on

1. Distribution - Developers of the kernel are spread all over the world. Indeed, Git was designed to fit the linux kernel developers' needs, but many developers of large open-source projects (i.e. Samba, Gnome, KDE, Ruby on Rails, Mozilla, ...) are not working in the same city or even in the same country. Distribution also brings advantages and of course some disadvantages with it (discussed later).

2. High-Performance - In a distributed environment speed is key. Developers don't want to wait minutes till an operation is done to check-in, check-out, branch, merge or diff.

3. Reliability - Developer "A" needs to be sure that every line of code he puts in the version control system is stored in a safe way so that Developer "B" is using the exact same code if he gets the code out of the VCS. If memory corruption or disk corruption occurring, you may never know it, unless you see the corruption in the files when you check them out.

4. Availability - even today a developer may not be always connected to a network or the internet. But even without being connected he should be able to version his code.

5. Content - A Version Control System should not be focused on files but more on the content in the files.

3.2 Performance

After looking at the needs of a Distributed Version Control System, especially of Git, it is useful to take a closer look at the technical side. Contrary to other DVCS like Mercurial or Bazaar, which are mostly written in Python, Gits major part is written in C. The code of Git is a mix of C, Perl, TCL, C++ and Bash-scripts, with about 55% pure C code, 18% Bash-script, 14% Perl and 6% C++. About 20% of the lines of code are dedicated to test-suites [2]. Due to its precompiled nature, C code is generally faster than code written in Python, which is an interpreted scripting language. As the following table shows, these performance differences are noticeable, if one is comparing between the three DVCS. Each of the systems were tested 8 times with a repository containing 12456 changesets and about 30.000 files.[2]

Commands	Git	Mercurial	Bazaar
Status	0.564s	1.333s	1.941s
Diff	0.609s	1.843s	2.847s
Clone	11.650s	23.755s	240.010s

Figure 2: Performance comparison of Git, Bazaar and Mercurial [2]

As one can see in the table, Git is more than twice as fast (at the diff operation even more than three/four times) compared to Mercurial and Bazaar. One of the reasons is the use of C instead of python. Another reason could be design and implementation differences, which's comparison might be difficult. Another interesting DVCS is Darcs[3] when it comes to the used programming language. Darcs uses the functional programming language Haskell[4] which is highly focused on concurrency and parallelism. A functional language was chosen to realize the patch-applying mechanism, efficiently and elegantly. Darcs patch-applying algorithm is based on the mathematical patch-theory. [9] But the downsides of Haskell are speed and memory cost, which makes it not as performant as Git. A polish blog compared Darcs and Git in detail, with the conclusion that the main difference is the difference between science and engineering:

> "Darcs represents what is best in the science, beautiful ideas and [long] waiting for the reply, approaching infinity. Git, on the other hand, represents engineering - down-to-earth, mundane, hairy duct-tape-driven architecture, responding within seconds.
> Choice between darcs and git is simple. We study darcs, we use git." [11]

For a more detailed benchmark on Darcs, see [10]. Brief summary: One key role of the performance of a DVCS, is the use of the appropriate programming language, but there are more parts not regarding the implementation but the design of the system itself. For example, today the linux kernel is the biggest open-source project with 22.000 files (in May 2007). As mentioned in the beginning, the community has had Git in use now for 5 years, and they are doing 4.5 merges in average every day. These merges are done automatically by Git or have to be done manually. So there is a need for a good performance even with that many files in the repository. In centralized systems developers have to go through the network to do branches, merges or diffs and most of the time the network becomes the bottleneck. Thus

[2]The full conditions of the benchmark can be viewed at http://www.infoq.com/articles/dvcs-guide#sectionbench

[3]http://www.darcs.net

[4]http://www.haskell.org/

not going over the network is a huge advantage. Especially if developers are spread over countries. In decentralized systems every developer has his own local repository on his hard-drive. This makes all operations very fast and a developer can even work without a network connection. But there are disadvantages as well, as discussed in 2.1.

3.3 Distribution and Availability

As described, developers may be spread all over the world in a distributed scenario. This means they sometimes do not have access to a central repository. The solution to this problem in a Distributed Version Control System is to give every developer has his own local repository on his hard drive - with no central repository where every developer checks-in and checks-out. Instead, a developer can share his code with other developers, with them deciding if they want to pull the code or not. The positive side-effect of this form of distribution is availability. As a result of a local repository check-ins, check-outs, merges, branches and diffs are cheap operations which don't const much time because one does not have to go over the network. Besides performance advantages, working locally brings advantages in availability. A developer doesn't need to be connected to the internet or a company network (i.e. over VPN), but can rather work offline - even on an airplane.

3.4 Reliability

Reliability means that the code put into the repository is stored in a safe way and is exactly the same when pulled out of the repository. Reliability also means, that the code is safe through the means of backup and distribution. Reliability in Distributed Version Control Systems is achieved through 3 things: distribution, trust and consistency. Because developers share their code, the code will be replicated at many locations ("natural replication of data") [7]. In a perfect scenario (assuming a large project) there is no possibility of a single point of failure, because all content of every developer gets distributed to at least one other developer, where in a case of data loss the developer can restore his code. The aptitude of this assumption will be discussed later.

The consistency aspect of reliability is achieved through technical features. Git for example creates a cryptographically secure hash with the SHA-1 algorithm through the path in the tree and the content of a Git-object (see section 4). If a Git-object is created on the check-in of some content, a SHA-1 hash is created through the content. On an operation like for example check-out, Git checks the content against this SHA-1 hash and tells the developer if any kind of corruption occurred. So one can trust the code pulled from a repository - whether it is local or remote. Other Distributed Version Control Systems like Bazaar and Mercurial in fact use the same technique. But data-corruption is certainly not the only reason why this feature is really important. The consistency adds security to a system: If someone tries to modify code to be malicious on purpose and distributing it, hoping the code gets to the final product - as it happened to the linux kernel developer team using bitkeeper back then - which even detected the break-in attempt with a simple crc16 check - he will fail because the attempt can be detected and restored.

3.5 Content

Git has a different approach on treating content than Subversion does. Subversion uses a "Delta Storage System" which means it stores files and sees every check-in as a modification on a certain file. Due to that Subversion stores only the difference between the initial file and the modified file. Git in contrary doesn't save the differences but rather snapshots of what all files in the project are looking like. So Git doesn't treat content in files and folders, it treats all content as binary large objects (blobs) which are only referenced by a tree-object.

The disadvantages of a storage system like Subversion is, that if you move a file to another folder (on the file system), Subversion doesn't notice that the file only has been moved by containing the same contents as it did before. Instead it marks the file as deleted in the old directory and adds the file on the new location to the repository - with all history lost for this item. But it isn't as bad as it sounds, because Subversion has a command (svn-move) that moves a file to another directory, but if you use "normal" file system operations like mv or if you use for example the subclipse-plugin[5] for eclipse, the svn-move command will not be used!

If you move a file in Git, Git recognizes that the object already exists because it uses a SHA-1 hash to identify the content-object. For example if you have two identical files in different directories, Git will use only one blob for both because the object is totally independent from its location in the object tree. So the history of a file won't be lost. This approach is fast. because only the 40-bit hash has to be compared and effective, because duplicate files are stored only once. Furthermore Git compresses all objects which is very effective on text-files. This drastically reduces disk space usage of the whole repository.

If one takes a closer look at Subversion, one can see that Subversion is referring not to single files, but rather always referring to the whole repository. On every change the whole repository gets a new revision number. The highest revision number represents the newest files in the repository. On every check-out, update or commit of a file, Subversion

[5]http://subclipse.tigris.org/

stores a copy of the file in a second local (and hidden) directory ".svn", to keep track of the changes made on the local working copy to the last copy of the repository. On a commit, the delta of the working copy to the version in the repository is calculated on the client. Only the delta of the two files is sent to the Subversion-server. But as in Git, check-ins are atomic, which means, if one file-commit fails, the whole commit will be undone.

One can understand the approach of Subversion, using deltas to save disk space. If one keeps in mind that Subversions development was started in 2000 and is the derivative of CVS, one can imagine that the Subversion was focused on disk space as well with users having a hard disk drives with only a few gigabytes that time.

For the full detail of the Git object model, see section 4

3.6 Summary

To summarize the last section, Git is a Distributed Version Control System, meaning every developer has his own local repository. Local repositories provide a fast access time to the repository and allow disconnected operations. This provides for good performance and availability. Gits object model ensures, that data is stored efficiently, compressed and secure, with a special focus on the content. This provides reliability and security, so that repositories are efficiently stored locally and can be shared between developers in a distributed scenario. Another performance issue, contrary to other Distributed Version Control Systems, is the use of the C as programming language. Git uses about 55% C code which is faster than Python or Perl used in Mercurial or Bazaar. A special DVCS is Darcs, which uses Haskell.

4 Git Object Model

The following section is a short summary of how the Git object model looks like and should only serve as a brief introduction. More detailed information can be found in the online-books: Git-Book [13] (official) or Pro-Git [12].

4.1 Git Objects

All physical files which are under revision control by Git are stored as git objects. An object is identified by a SHA1-hash of its content. The content of an object is gzip-compressed. Besides the content an object has a content size and a type. Git defines three types of objects: blobs, commits and trees. A blob-object represents the content of a file. A commit-object contains information about the root tree which was committed, the author and the committer. And a tree-object contains information about its subtrees and the blobs in the tree. Branches, remote-tracking branches, and tags are all references to commits.

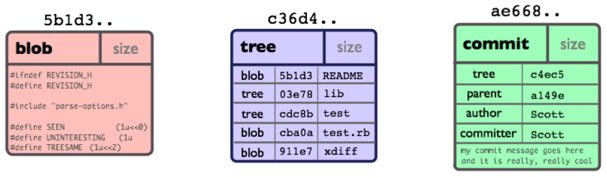

Figure 3: Example Git objects

Every file in Git is represented as a Git-object, but Git internally distinguishes between two kinds of objects: loose objects and packed objects.

4.1.1 Loose objects

A loose object is a simple representation of a Git-object, which stores the compressed data in a single file on the disk. If the object type is a blob, it contains the file data, if it is a tree it contains the tree infos. Each object is written to a separate file in the filesystem and all of these objects consist of just the compressed data plus a header identifying their length and their type.

So every time a new file is added to Git and gets committed, Git creates a new object for every file, a tree object for every folder, a commit object for every commit and creates or updates the head pointer to that commit.

There are a few commands in git, where one can manually lookup how git arranges the data. The following commands show how the contents of a commit, a tree, and a blob look like.

Commit example

```
# git show—ref ——heads
31d74b06c98ed779c5b3dd0565f8517c0c0b82f3  refs/heads/master

# git cat—file —t 31d74b06c98ed779c5b3dd0565f8517c0c0b82f3
commit

# git cat—file commit 31d74b06c98ed779c5b3dd0565f8517c0c0b82f3
tree 553d7d3af042dbc372b47d33020b51d6f7a70165
parent c928d27d2503ec1a18f8c245e2f1a996bfe1d793
author daniel.kuhn <daniel.kuhn@c30caeda—8a65—0410—9622—a86fec2c9975>
1257266756 +0000
committer daniel.kuhn <daniel.kuhn@c30caeda—8a65—0410—9622—a86fec2c9975>
1257266756 +0000

my commit message
```

Explanation:
git show-ref –heads: shows the head-commit
git cat-file -t: shows the type of an object
git git cat-file commit: shows the contents of the commit object

Tree example

```
# git ls—tree cdf19197574e343118a7ab9779984b01d426e3d7
100644 blob 7b3d68982c93d0321c39b76d53d4c144c8cfdce7    RestConfigService.as
100644 blob 41ebe106eb4bfa00743979c2ec7cd0aca3a34dfa    RestPhotoService.as
100644 blob bd4b2d13d03151091d57dfcaa7abdf4d41d61cd9    RestPopeService.as
100644 blob 98f1b8fd1b35b53e430ed80134b487f7d07621f5    RestRequest.as
```

```
100644 blob 494383e12bb97455da7e35330dcc811e0b581eb7    RestSkinService.as
100644 blob 3e811f7cd98c730033c74e769074835ea1154e85    RestUserService.as
040000 tree e8edd262b155bc05985433ad0dfa967698230ff8    rpc

# git ls-tree e8edd262b155bc05985433ad0dfa967698230ff8
040000 tree ae6a8a0ec96db652847084c9f7c405afefcfce3c    events
040000 tree c7ec288108f19df8c81cd9df2cdad262dbeac980    rest
```

Explanation:
git ls-tree: shows the references of a tree object

Commit example

```
# git cat-file blob 7b3d68982c93d0321c39b76d53d4c144c8cfdce7
/**
 * @author: Daniel Kuhn
 * Clientside connector to the editor-backend REST-service.
 *
 * */
package org.dom.rest
{
...
```

Explanation:
git cat-file blob: shows the contents of a blob object

4.1.2 Packfiles

Besides the loose objects there are the packfiles, which are little more complex. In general git stores every new version of a file in a new loose object on the disk, even if there are no big changes done to it. At this time there is no delta system to store only the differences (e.g. like it is in Subversion). This approach has two major advantages: it is cheap considering write/read and fast, but it is a waste of disk space with the project growing. Therefore git uses a packfile which is generated when the command "git-gc" is called. Git then rewrites the loose objects to the packfile format to save only the parts that have changed over different versions, with a pointer to the file it is similar to, using "a rather complicated heuristic to determine which files are likely most similar and base the deltas off that analysis" [6]. This packfile is only an internal format of git and doesn't change the way to access the files manually. It does however affect the speed of the file access, because it is more expensive to read a packfile than loose object files.

For every packfile created, Git creates two actual files on the filesystem: the packfile-index (.idx) and the packfile (.pack). Both files are beeing stored under the .git/objects/pack/ directory. The .pack file contains the actual data, accompanied by metadata like version, entries and a checksum. Every data entry in the .pack file contains its own metadata like object type and size.

[6]http://book.git-scm.com/7_how_git_stores_objects.html

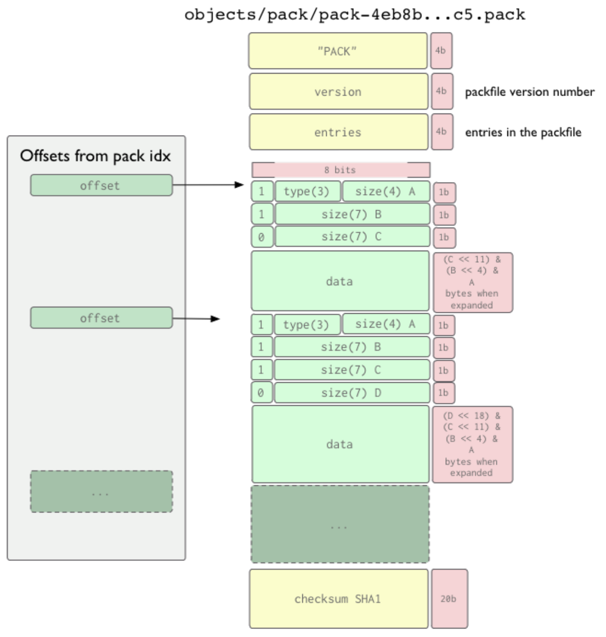

Figure 4: packfile format

The index of the packfile contains a fanout (size), the sha1, the crc and the packfile offset (32bit or 64bit when larger then 2 gigabytes) of every object packed in the packfile. Which means, that if you have 5 objects in one packfile, you have 5 fanout entries, 5 sha1 entries and so on. The index futhermore contains a header and a trailer with the .pack-file checksum and the index-file checksum.

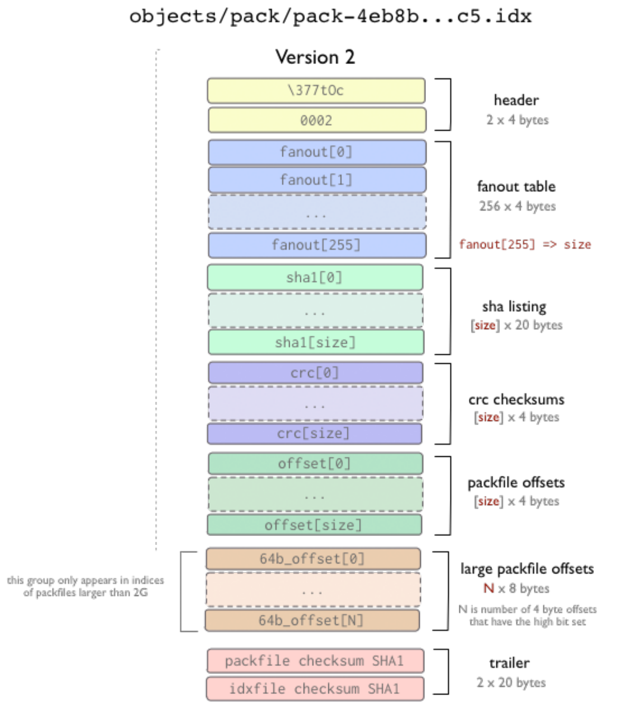

Figure 5: packfile index format

There is a very detailed, down to the bit, explaination of the packfile-format, which can be found under [7].

4.2 Index

A Git repository contains one (hidden) ".git" directory, which has the following structure:

```
– .git
–– HEAD (file)
–– branches (directory)
–– config (file)
–– index (file)
–– info (directory)
–– logs (directory)
–– objects (directory)
–– packed–refs (file)
–– refs (directory)
–– remotes (directory)
```

Git contains an index which is stored as a binary file in the ".git" directory (.git/index). It contains all blob objects, each with their SHA-1, their permissions, their path-name and some additional data (such as the modification date) sorted by their path-name.

The Git index has the following purposes:

1. Tree-object generation – a commit generates his tree objects from the index
2. The index defines the working tree
3. Fast-comparison – the modification date is used to determine quickly which files in the directory differ from the ones in the index. This is faster than a one by one comparison or a SHA-1 calculation
4. Merging – it can effectively represent information about merge conflicts between different tree objects. During a merge, the index can be storing multiple versions of a file

Unlike Subversion, which stores a .svn folder in every folder inside the repository, git contains only one .git folder in the repository root directory. Furthermore the .git folder contains all information about the whole repository, whereas the .svn folders contain only the information for its folder it is in, and contains only information about the last revision. The difference is sure an effect of having a local repository or a central one. But the a central place to keep information makes it also easy to maintain.

4.3 Commit process

If a file, which is already under version control and has been commited, has changed , Git generates a new blob-type object with a new SHA-1 name and stores it in the objects folder. For every folder -from the root directory to the directory the file is in- new tree-type objects are created. So if a file lies in gitroot/models/storage/ two new tree-type objects are being created. The new tree objects contain references to every blob or tree object, including the newly created ones. Besides the tree objects a new commit-type object is being created with a reference to the root tree-type object, the commiter, the author and the SHA-1 hash of the parent commit object. Afterwards the actual branch object (eg. Master) is led to the new commit object. During this commit process old objects are neither being deleted nor packed.

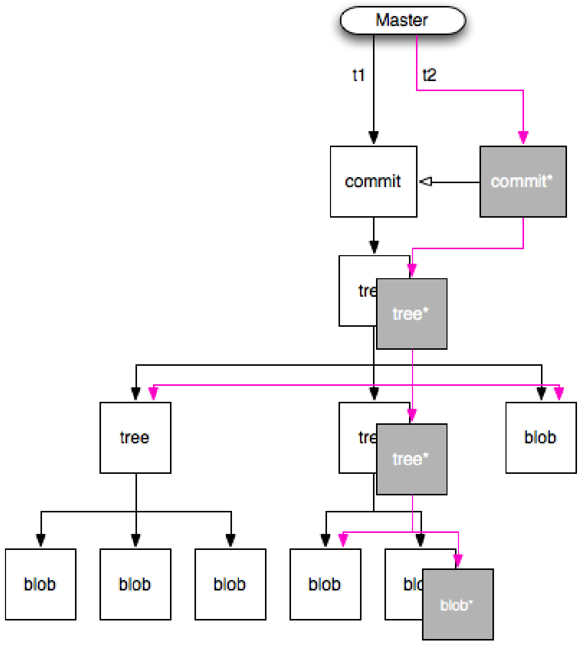

Figure 6: Example commit process

4.4 Transfer protocols

Git can push and pull data from and to repositories in three different ways: over HTTP, over the Git protocol or locally.

Local: A local push or pull via a local path is the easiest way to sync data between repositories. By just calling #git push (or pull) /somepath/somerepository, Git syncs the local repository to another local one. This is the easiest and fastest way and even possible over a network share (eg. samba) or via sshfs.

HTTP: By fetching Data over HTTP you can pull data from a repository on a webserver without having a separate git server installed. If the .git directory is accessible via HTTP Git can pull the data by calling #git http-fetch http://someweburl/. HTTP access to a Git repository is always read only. Github for example offers the choice between HTTP or Git-protocol for non-contributers.

[7]http://book.git-scm.com/7_the_packfile.html

Git Protocol: Fetching data with the upload pack is the best solution and more efficient than HTTP. Git opens a socket over ssh or over port 9418 (when using the git:// protocol). Unlike HTTP the client does not download the data, but it rather sends a request to the Git-server and asks for the repository. The server now processes the request and turns it into a git-upload-pack call which streams the packfiles down to the client. Pushing data over the Git protocol is quite similar. The Client sends a request to the Git-Server which opens a receive-pack instance, which is started up if the client has access to. When the client calls #git push git://someurl/project.git the client starts up a send-pack process and the server starts up a receive-pack process. Afterwards the client streams the packfiles to the server.

More information about the Git transfer protocols can be found in the Git Community Book under http://book.git-scm.com/7_transfer_protocols.html.

5. Conclusion

Centralized VCS systems are designed with the intent that there be only one central repository. All developers work from that source and make their changes, which then become a part of that source. The only real difference between all the centralized VCS is in the workflow, performance, and integration that each product offers.

Distributed VCS systems are designed with the intent that one repository is as good as any other. Sharing the code between these repositories is just a form of communication over certain protocols. Any semantic value like trust or security is not defined by the software itself.

Distributed Version Control Systems solve problems differently than centralized VCS. Comparing the two systems in the first place feels like comparing screwdrivers with hammers, but if you dig deeper, they are just two different systems for one and the same problem: collaboration on sourcecode. Both systems solve the problem. They just do it with different characteristics in the means of backup, security, reliability, speed, distribution and easiness. Which system (CVCS or DVCS) is best depends on the environment it is supposed to be used in and on the preferences of the people in charge. In my opinion, the "best system" does not exist, because I think one can reduce it to distribution or centralized. Things like reliability, speed and security which are often advertised as the key features of for example Git can easily be implemented in svn too. Git however, being the newer system and having learned from the mistakes made in other VCS, has a great advantage. In implementation and design Git is clearly the better system than e.g. SVN right now.

The real choice between using CVCS or DVCS is organizational. If your project or organization wants centralized control, then a DVCS is a no-go. If your developers are expected to work all over the country or world, without secure broadband connections to a central repository, then DVCS is probably the solution. There is always the choice to use both systems, and if you do, you will need very strict rules and workflows.

References

[1] Patrick Braga, "C vs Python: Speed" *http://theunixgeek.blogspot.com/2008/09/c-vs-python-speed.html*, from 2008/09/07 (last visited: 2010/03/11).

[2] Sebastien Auvray, "Distributed Version Control Systems: A Not-So-Quick Guide Through" *http://www.infoq.com/articles/dvcs-guide* (last visited: 2010/07/04).

[3] John Brown, "CSV Home Website" *http://www.cvshome.org/entwicklung.html* (last visited: 2010/07/04).

[4] "SCSS - The POSIX standard Source Code Control System" *http://sccs.berlios.de/* (last visited: 2010/07/04).

[5] Thomas Weber, "Subversion - Der Nachfolger f§r CVS" *http://www.trivadis.com/uploads/tx_cabagdownloadarea/Subversion_Art.pdf*, from 2005/02/16 (last visited: 2010/07/04).

[6] Ian Clatworthy "Distributed Version Control Systems Ð Why and How" *http://ianclatworthy.files.wordpress.com/2007/10/dvcs-why-and-how3.pdf* (last visited: 2010/07/04).

[7] Linus Torvalds, "Goolge Tech Talk - Linus Torvalds talks on Git" *http://www.youtube.com/watch?v=4XpnKHJAok8*, from 2007/05/03 (last visited: 2010/07/04).

[8] Randal Schwartz, "Goolge Tech Talk - Randal Schwartz talks on Git" *http://www.youtube.com/watch?v=8dhZ9BXQgc4*, from 2007/10/12 (last visited: 2010/07/04).

[9] Mark Stosberg, "Interview with David Roundy of Darcs on Source Control" *http://osdir.com/Article2571.phtml*, from 2004/11/23 (last visited: 2010/07/04).

[10] "Benchmark results" *http://lists.osuosl.org/pipermail/darcs-users/2008-December/016757.html*, from 2008/12/23 (last visited: 2010/07/04).

[11] "Darcs vs Git: mathematician versus engineer" *http://www.3ofcoins.net/2008/12/16/darcs-vs-git-mathematician-versus-engineer/*, from 2008/12/16 (last visited: 2010/07/04).

[12] “Pro Git ÐÊprofessional version control” *http://progit.org/book/* (last visited: 2010/07/04).

[13] “Git Community Book” *http://book.git-scm.com* (last visited: 2010/07/04).