An investigation of Git in an XP project

Erik Jansson
ada09eja@student.lu.se

March 4, 2013

Keywords: *Git, distributed version control systems, agile project*

Abstract

Git is a distributed version control system with a different model than that of traditional version control systems (e.g. Subversion or CVS). In the course Software Development in Teams on LTH, the students work in groups of 8-10 people, following the XP-model, to develop a system. Subversion has traditionally been used as the version control system for the teams. This study will evaluate what problems replacing Subversion with Git introduces and/or solves for a team.

1 Introduction

As soon as the number of members in a programming team exceeds one person, there is some need to synchronize and share code among the team members. In order to solve the problems that inherently come with this sort of sharing (e.g. locking or merging, keeping history, tracking authorship, etc.), a version control system is typically used. In the course *Software Development in Teams* on LTH, the students work in groups of 8-10 persons, following the XP-model, in developing a time tracking program for enduro races [8]. Older students that follow the course *Coaching of Programming Teams* acts as coaches for the groups. Section 2.3 describes the project in more detail.

Subversion (a centralized system) is traditionally used by the teams as their version control system. This study will investigate what problems switching to Git (built on a decentralized model) introduces and/or solves for the team. The premise is that it will eliminate some of Subversion's shortcomings (e.g. renaming and removing files, merging, speed, etc.) and induce the team to use branches more liberally, perhaps with the drawback that learning Git can take some time.

The study was performed on one of the teams in the course and a short survey was handed out to the team members a couple of weeks into the project, asking how they perceived using Git, what expectations they had before they began and how difficult it was to use. The result of this study is based on this survey as well as discussions with one other group's coaches which also performed a similar study with their team.

This paper is structured as follows. In Section 2, we outline some of the differences between centralized and decentralized systems, as well as a brief description of the course setting this study was performed in. In Section 3, we

describe in detail how the study was performed and in Section 4 we lay out the result. Lastly, in section 5, we provide a summary of the results and discuss future extensions of this study.

2 Background

This section describes some of the key differences of centralized and decentralized (or distributed) version control systems. It is outside the scope of this article to describe version control systems in detail. We will simply explain the fundamental differences relevant to this paper. Brief introductions to version control systems and their history can be found in [9] and [10]. The two terms distributed and decentralized are used interchangeably throughout this article.

2.1 Centralized Version Control Systems

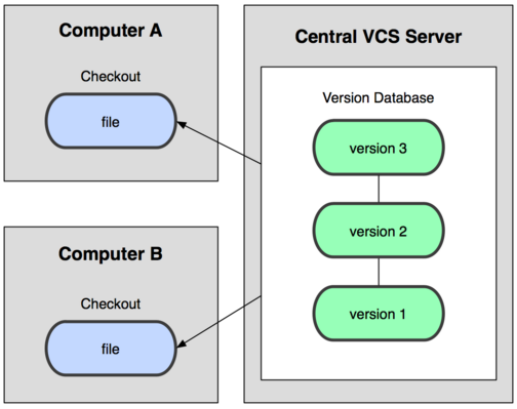

Figure 1: A figure of how a centralized system can work. From [6].

In the centralized model, to which Subversion and CVS adheres, a server that hosts the repository is utilized. In this model, a developer typically checks out a snapshot (called a working copy) from the server and performs work on this snapshot in their workspace. Once they are done, they commit the changes back to the central repository, which the rest of the development team then can fetch from to receive the updated source code in their workspaces. [9]

There are different models on how to handle conflicts in these systems, e.g. *locking* and *merging*. *Locking* means that once a developer has checked out a certain file, no one is able to perform a checkout of that file until it is checked in by the first developer again. On the other hand, the *merging* model allows developers to checkout the same file simultaneously and the system tries to combine the changes (i.e. *merge* them) when the modified files are checked in to the repository. The later model is used in e.g. Subversion and CVS.

The centralized model has some benefits as well as drawbacks. Some of the benefits are:

- The synchronization problem is solved since the source code is placed in a common repository.

- The merging strategy allows multiple team members to work in parallell.
- The development history is kept at the server along with author information which enables tracking of changes.

Some of the drawbacks are:

- Synchronization requires all team members (as well as the central server) to be online while working.
- The connectedness also introduces network latency into the workflow, which can be a problem when applying a workflow where changes are merged often in order to avoid large conflicts.
- The merging strategy can result in conflicting versions of files that require manual merging if the tool cannot perform an automatic merge.
- Branching in Subversion/CVS is a heavy operation which involves copying a lot of files [6].
- If Alice and Bob begin working simultaneously on the same branch and Alice commits her changes first, Bob will not be able to commit his work before he has merged Alice's changes. The only copy he has of his work resides in his workspace so should something go wrong during the merge, his work is not yet permanently stored in the repository and he risks loosing all or parts of his changes [9].

2.2 Distributed Version Control Systems

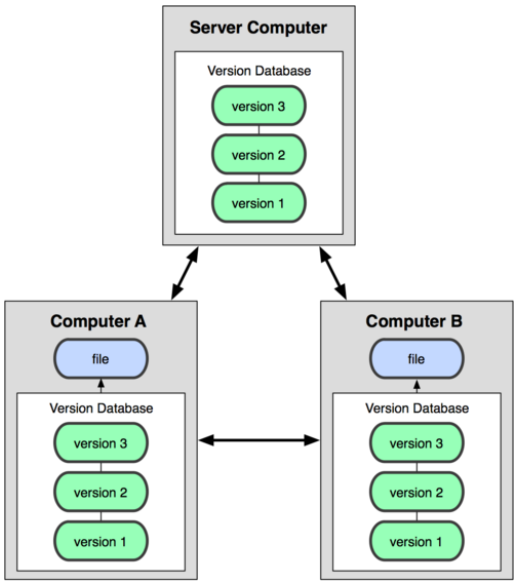

Figure 2: A figure of how a decentralized system can work. From [6].

As an alternative to this model, there are distributed (or decentralized) version control systems (DVCS), like Git or Mercurial (although this report is focused on Git). This model differs from more traditional version control systems in that (as its name implies) it does not depend on a central server hosting the repository. Each team member has a complete copy of the entire repository (including all history, branches, etc.) on their machines. Each team member can work offline, commit to their local repository and then share their code to the other team members. At first, this might seem foreign for a person used to the traditional model but it has many advantages:

- It allows for a much freer workflows (e.g. hierarchical as the Linux kernel uses) [11].
- Having a complete copy of the repository allows for faster operations since there is no added network delay for most of the operations. This is useful when working as a distributed team where latency to the central server might be a problem. In this case, a developer can work mostly offline and limit the number of network calls needed during the day. [7]
- Branching and merging are usually much simpler in distributed version control systems [7]. In fact, merging is the only option for conflict resolving in DVCSs since it is not possible to lock a file in all copies of the repository. This has led to well developed merging algorithms in these systems (although there is no reason why centralized systems cannot have equally good merging algorithms).
- Distinction between committing changes to the local repository and publishing them to the world (e.g. pushing to Github). In for example Subversion, a commit implicitly publishes a change [9].
- The problem Alice and Bob had when using a centralized system is solved. Both Alice and Bob can work and commit locally which gives both a permanent record of the changes they have made in their respective repositories. Once they have this permanent record, they can merge each other's changes without the risk of loosing any code [9]. Git, for example, provides a command (`git merge --abort`) that aborts the merge and restores the repository to the pre-merge state.

In addition, Git in particular has many features that makes it a compelling DVCS:

- In Git, branches are extremely light weight and easy to create and destroy, which encourages a branch-often workflow [6].
- Git has a staging area which makes it extremely easy to commit only a subset of the changes in you workspace, or even a subset of the changes in a file [6].
- Git stores the changes as a graph in which every commit refers to its parent(s). This makes it very hard to loose or corrupt information in the repository (intentionally or not) [6, 11].
- The `bisect` command in Git performs a binary search through two revisions of the repository to find the exact commit that introduced a bug.[9]

Git can also "emulate" a traditional VCS by every team member agreeing on using one repository as a central one, which everybody push updates to and fetch changes from. This could allow for a softer transition from a traditional model. Describing all of Git in detail is to great a task for this report and the reader is encouraged to read the Pro Git book (listed in the references) for a thorough introduction.

There are however, still some drawbacks with distributed systems. Merge conflicts are still unavoidable problems when not utilizing the locking model. If, one does not adhere to best practices, a distributed system can in fact result in even larger merge conflicts (due to longer running branches). Also, when transitioning from a centralized model, there can be a learning curve which is hard to overcome. Of course, some operations do still need network access (e.g. push and pull/fetch) in order to facilitate collaboration with other people.

In recent years, distributed version control systems, and Git in particular, has achieved some significant traction in the open source world [7]. Here is a short list of projects that have switched to Git from other version control systems:

- Drupal, the open source content management system [2].
- KDE, the open source desktop environment [1].
- Perl, the programming language (it reportedly took 21 months of manual work to do it) [4, 7].

Several other projects are at the time of writing discussing moving to Git, e.g:

- OpenMRS (an open source medical record system) [3].
- The Boost C++ Libraries [5].

Even FreeBSD, which for historical reasons has been hosted on CVS (or more recently Subversion), provides a Git mirror[1] (although their main repository is still Subversion based).

The open directory of open source projects, Ohloh, which projects volunteer statistics to, reports at the time of writing that Git is used in 29%[2] of the projects listed in the directory which makes it the second largest VCS.

2.3 Project Setting

The study was performed during a course on the Faculty of Engineering's Computer Science department at Lund University (abbreviated LTH), which all second year students of the five year master program in Computer Science are required to take. The students work together in a team of 8–10 persons, following the XP model, to develop an application. Each team has a customer (played by an employee at the university) who prioritizes what stories (i.e. features) should be implemented. Each team also has two coaches who are older students taking a course called *Coaching of Programming Teams*.

Each week is laid out similarly; during the Monday, an eight hour work day is simulated in a computer room, on Wednesday the team has a two hour

[1] `https://github.com/freebsd/`

[2] `http://www.ohloh.net/repositories/compare`

planning meeting where the previous Monday is discussed and, with input from the customer, the next week's stories are planned and estimated. During some of the Monday sessions, the teams are expected to release a version of the application to the customer. The team members are also required to do a "spike" outside these hours each week. This can consist of anything between evaluating the usefulness of a new library and investigating a tool for performing static code analysis.

For more information regarding this course, see [8] which discusses it in detail.

3 Method

This section will describe how the study was performed during the course. All results are given in Section 4.

As described in Section 2.3, the subject of this study was a group of eight students at the five year computer science program on LTH. During the development sessions, Subversion was replaced by Git as the team's version control system. To allow for an easy transition into a decentralized model, and to provide easy access to the repository from home, Github[3] was used as a central repository to which everyone could push to and fetch from.

During the first meeting, the team received a short (10-15 minutes) introduction to how Git differs from the centralized model they had learned during the course introduction, as well as some of the basic features of Git (i.e. staging area, how to commit, pull and push, etc.). They were also instructed to read an introductory text (or follow an interactive guide) about Git and create a Github account before the first laboratory session the following Monday. In order to make it easy for the team, a list of resources was put together for them to choose from. These are listed in Appendix A.

A Github repository was set up before the programming labs and at the first laboratory lesson each team member's Github account was added to the repository as collaborators, granting them read and write access to it.

During the laboratory session, learn by doing was applied and the team began working directly on the project. When problems with Git came up during the day that the students could not solve themselves, the author helped them out. The choice was given to the team whether to use Git via the terminal or the Eclipse plugin EGit. Some team members chose the plugin while others did not even install it and preferred using the terminal instead.

At first, the plan was to use so called *feature branches*, i.e. each new feature (or story) is implemented in a new branch which, when finished, is merged back into the master branch. The reason for this was to keep the master branch clean from unfinished implementations and to compartmentalize the development. However, it was discovered during the first laboratory session that this was difficult for the team to understand and the idea was dropped. Instead, branching was reintroduced after a couple of weeks when the team was more familiar with using Git in its other aspects.

Three weeks into the project, a short questionnaire (listed in Appendix B) was handed out to the members of the team asking if they had any previous experience using a VCS, what expectations they had on Git and how they

[3] `https://github.com/`

had actually experienced using Git. The first question (whether they had any previous experience with version control systems or not) was chosen to get an overview of what knowledge they had about the subject before the project. The second and third questions (about their expectations on and experience of Git) was chosen as free text questions in order to allow the team members to write freely and give as much information as possible.

The result of this questionnaire, as well as the author's observations during the project and discussions with coaches from another team performing a similar study is what the results of this study are based on.

4 Result

A large majority of the team (75%) responded that their expectations on using Git instead of Subversion were positive. One quote in particular that stood out was: "A bit cooler than what we did on the CVS lab". One person responded that they did not have any expectations since he/she had never heard of Git and one responded that it felt "a bit tricky". This was encouraging since a negative view when beginning to use the version control system could eviscerate it's success.

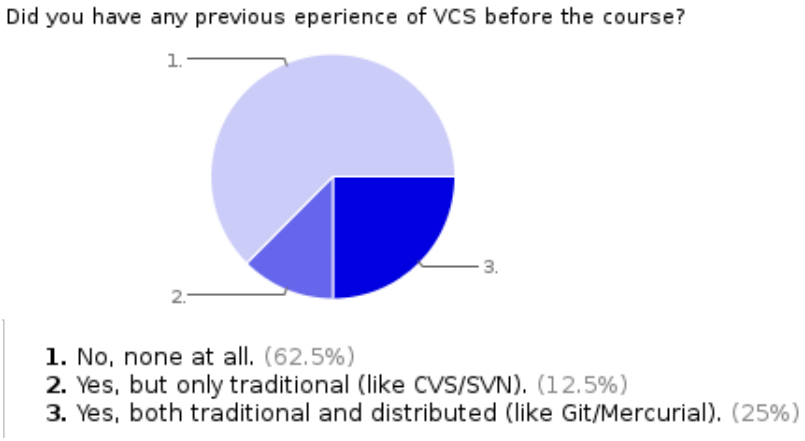

Figure 3: The team's responses to whether they had used VCS's before.

For a majority of the students (five of the eight), this course was their first introduction to using a version control system at all (see Figure 3). In the beginning of the course, the students receive a two hour lecture on the topic of software configuration management[4] and a two hour laboratory session[5] introducing them to CVS. Therefore, at the start of the actual project, they had only been practicing very briefly the centralized model and not at all the decentralized model.

This was noticeable at first where both Git's staging functionality and whether they were committing to their local repository or pushing to Github was a bit

[4] http://cs.lth.se/kurs/eda260_2012/forelasningar/
[5] http://cs.lth.se/kurs/eda260_2012/labbar/

confusing for the team. However, all except two team members responded that the basic features Git was easy to learn which points to this being a minor problem. After a week, these problems also started to go away.

The fact that they receive such short introduction to version control in general was noticeable during the project (although it got better towards the end). Such simple best practices as keeping commits small and to one feature at a time was very difficult for the team to follow. A course in configuration management (which covers version control) is given at LTH but most students does not take it until later into their studies. This could be a valuable course to give earlier in the student's education to eliminate this problem altogether.

Branching tended to be a problem for the students at first. Mostly, they were confused about some of the following:

- What branch they were on,
- when to merge a branch,
- what to do with a branch once it was merged, and
- how to solve a merge conflict (i.e. fix the conflict in the file, add it to Git's staging area to mark as resolved and commit the merge).

At the beginning, they were also unsure about the benefits of branches and were thus reluctant to perform even the small extra work associated with branching. Because of these problems, using branches was dropped for a while and reintroduced after three weeks of using Git. At this point, using Git was more familiar for the team and it was easier to grasp. There was still some confusion when it came to what command line parameters should be used when pushing changes to (and fetching changes from) a remote branch on Github. Once these were explained, branching worked very well for the team. Some of them even began performing more advanced workflows, such as performing a *rebase* on the master branch (i.e. pulling in the new changes from the master branch into their branch) in order to solve the merge conflicts in their own branch before they merged with the master branch. This helps keeping the master branch clean by resolve all conflicts outside it.

At this time, they also realized the benefits of branches and began using them more. Many of them did spikes that required programming on a separate branch to avoid cluttering up the master branch with experimental features such as using the GSON library to read and write JSON files.

Using Github to synchronize between the team members turned out to be very smooth. Since Git is distributed, it would be possible to skip the central repository and pull changes from each other in the team. However, this was deemed to be too much overhead for the team. SSH-keys for each team member would have to be shared among all other team member's computers in order to fetch and push changes to and from the other team member's repositories. It is important to point out that the distributed property of Git simply gives the *possibility* to use many different workflows, one should certainly chose the one which is the most suiting for the occasion. In this case, the team is large enough that using a central repository is valuable but not large enough that using a hierarchical model is needed. The Github repository also facilitated access from home which was helpful when they were doing spikes from home.

During the laboratory lessons, the team which shared our computer room, and many of the other coaches which the author has discussed the topic with, had numerous problems while trying to rename or remove files in Subversion. Sometimes the files would not disappear from the repository and sometimes both the old file and the renamed file would appear in the repository at the same time. None of these problems were experienced by our team since Git tracks content and not files. This enables it to discover that content has been moved from one file to another very easily.

Some coaches even reported that Subversion had lost files entirely. This also never happened to our team which is attributed to Git's data integrity (Git uses the cryptographic hash SHA1 to be certain that objects, called blobs in Git, never change or disappear once in the repository).

Despite the problems discussed here, almost all team members replied to the questionnaire saying that using Git was a positive experience. Many of them noted that having the local repository as well as the remote repository on Github made them feel safer. They felt more adept at trying out some commands since they knew that they could always remove their repository and fetch a fresh copy from Github without tainting the repository for the rest of the team. The same safety might not be experienced with Subversion since many of the commands in Subversion alter the entire repository (e.g. simply creating branches).

Some of the team members complained that they did not felt that they had enough time to learn as much of Git's features as they wanted. At the end of the project, four team members noted that they had begun using Git at home or in other courses since it was "very easy and convenient to use". One of them had even begun reading the 288 pages long Pro Git book (the last one on the list of resources they were given).

5 Conclusion

In conclusion, all team members noted that using Git was a positive experience and many of them commented that from what they had heard of the other team's problems with Subversion, they were lucky to use Git instead. Unfortunately, using Git with all of the features that makes it appealing requires much more time to learn it than was available in this study. In a possible future study, it would be interesting to follow a team that gets a more thorough introduction to Git (e.g. branching, rebasing, cherry-picking, and best practices) and version control systems in general and see what the result would be in that case. Hopefully, they would be able to use more advanced features at the beginning of the project. Perhaps adopting a working model with a master branch that is always release ready and tested and a development branch, from which feature branches are forked and merged, which is merged back into master when tested and reviewed.

The author's premise before the study was that branches was something that would be heavily used during the project, since branches are one of Git's "killer features". However this was not really the case, although branches were used by the team members in the later part of the project. The lack of heavy branching is attributed to not having received enough training in source code management overall as well as the project being a small one in which heavy use of branches gives small return on investment.

Lastly, we find that Git is a very viable replacement for Subversion (it can even emulate Subversion fairly well), with the reservation that the students need a bit longer learning period than what they are currently given in the course. They could however benefit from having this longer learning period even when they use Subversion and this might not be a problem.

As a last anecdote, during the retrospection at the end of the last laboratory lesson, the author listened while another team discussed what they had learned during the project. The first item that came up was: "Subversion sucks".

A Git resources

This list of resources was put together for the team's convenience when they were learning about Git. The list is sorted by how advanced the resource is (easiest is first).

- http://try.github.com/ (An easy interactive online tour of Git)
- http://rogerdudler.github.com/git-guide/ (An easy guide that covers most of Git's features)
- http://gitimmersion.com/ (An easy interactive guide of Git that one can follow on one's computer)
- http://learn.github.com/p/intro.html (Github's learn pages)
- http://git-scm.com/book (A book written by Scott Chacon at Github)

B Questionnaire

The short questionnaire handed out to the team contained these three questions:

- Did you have any previous experience of version control systems before the course?
- What expectations did you have when you learned that You would use Git in the project instead of Subversion?
- How have you experienced working with Git?

The questions where free text and the concious choice was made to keep the number of questions to a minimum in order to receive more comments on each question. Had there been many questions, the risk was that the team members would tire half way through the questionnaire.

C License

References

[1] Getting Started/Sources. http://techbase.kde.org/Getting_Started/Sources. Retrieved on 2013-02-17 20:37.

[2] Migrating Drupal.org to Git [Complete]. http://drupal.org/community-initiatives/git. Retrieved on 2013-02-17 20:28.

[3] Migrating to Git. https://wiki.openmrs.org/display/docs/Migrating+to+Git. Retrieved on 2013-02-17 20:48.

[4] Perl 5 now uses git for version control. http://use.perl.org/articles/08/12/22/0830205.shtml. Retrieved on 2013-02-17 20:39.

[5] Why Should Boost Move To Git? https://svn.boost.org/trac/boost/wiki/Git/WhyGit. Retrieved on 2013-02-17 20:51.

[6] Scott Chacon. *Pro Git*. Apress, 1st edition, Aug 2009.

[7] Brian de Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, CHASE '09, pages 36–39, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Görel Hedin, Lars Bendix, and Boris Magnusson. Teaching extreme programming to large groups of students. *Journal of Systems and Software*, 74(2):133–146, 2005.

[9] Bryan O'Sullivan. Making sense of revision-control systems. *Commun. ACM*, 52(9):56–62, Sep 2009.

[10] Nayan B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, Jan 2010.

[11] Diomidis Spinellis. Git. *Software, IEEE*, 29(3):100–101, May-June 2012.