View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Oulu Repository - Jultika

OULUN YLIOPISTO
UNIVERSITY of OULU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vili Seppänen

Open Source Version Control Software

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
March 2015

Seppänen V. (2015) Open Source Version Control Software. University of Oulu, Department of Computer Science and Engineering. Bachelor's Thesis, 22 p.

ABSTRACT

The environment around open source version control software is very opinionated and therefore it is hard to find unbiased comparison between different open source version control software. This Bachelor's thesis provides background and covers the basics of version control systems. Thesis also categorizes and differentiates the main types of version control systems, by investigating the way they handle repositories and by categorizing them to centralized and distributed. Finally, this thesis provides the unbiased technical comparison of the selected open source version control software and a way to map a suitable one for a software project. Comparison of technical details is collected into tables for easy interpretation and the main differentiators are explained more carefully. Mapping is achieved by pairing the major characteristics of different software projects with the technical features of version control systems and then with version control software that best supports these specific features. Pairing of the software project and the version control software is further refined with technical details that are not covered by the needs of major characteristics of the software project. Selection of the open source version control software is restricted to the four most popular ones.

Keywords: CVS, Subversion, Git, Mercurial

Seppänen V. (2015) Avoimen lähdekoodin versionhallintaohjemistot. Oulun yliopisto, tietotekniikan osasto. Kandidaatintyö, 22 s.

TIIVISTELMÄ

Avoimen lähdekoodin versionhallintaohjelmistoista on vaikea löytää puolueetonta vertailua, koska mielipiteet niiden ympärillä ovat hyvin polarisoituneita. Tämä kandidaatintyö tarjoaa taustatietoa ja käy läpi versionhallintajärjestelmien perusteet. Lisäksi tutkielma luokittelee ja erottaa versionhallintajärjestelmien päätyypit tarkastelemalla järjestelmien tapaa käyttää versionhallinnan varastoa ja kategorisoimalla järjestelmät sen mukaan keskitettyihin ja hajautettuihin. Tämä kandidaatintyö esittää myös puolueettoman teknisen vertailun yleisimmistä avoimen lähdekoodin versionhallintaohjelmistoista ja tavan kartoittaa ohjelmistoprojektiin sopiva versionhallintaohjelmisto. Tekninen vertailu on koottu taulukoiksi tulkinnan helpottamiseksi. Lisäksi pääeroavaisuudet on käyty läpi tarkemmin. Sopivan versionhallintaohjelmiston kartoittaminen ohjelmistoprojektiin on toteutettu yhdistämällä ohjelmistoprojektien tunnusomaiset piirteet versionhallintajärjestelmien ominaisuuksiin ja tämän jälkeen valitsemalla versionhallintaohjelmisto, joka toteuttaa kyseiset ominaisuudet. Tämän lisäksi versionhallintaohjelmiston valintaa tarkennetaan ottamalla huomioon ne tekniset eroavaisuudet, jotka eivät tule esiin tarkasteltaessa ohjelmistoprojektin tunnusomaisia piirteitä. Avoimen lähdekoodin versionhallintaohjelmistoista mukaan on valittu vain neljä käytetyintä.

Avainsanat: CVS, Subversion, Git, Mercurial

TABLE OF CONTENTS

ABSTRACT 2
TIIVISTELMÄ 3
TABLE OF CONTENTS 4
FOREWORD 5
ABBREVIATIONS 6
1. INTRODUCTION 7
2. VERSION CONTROL SYSTEMS 8
   2.1. Background 8
   2.2. Main types of version control systems 8
3. VERSION CONTROL SOFTWARE 12
   3.1. Repository operations 12
   3.2. Technical Status and Interfaces 14
4. PROJECT DEMANDS FOR VERSION CONTROL SYSTEM 16
   4.1. Open and closed source projects 16
   4.2. Local and multisite projects 17
   4.3. Small, medium and large amount of code 17
   4.4. Number of developers 17
   4.5. Platform support 18
5. MAPPING SUITABLE VERSION CONTROL SOFTWARE 19
6. CONCLUSION 21
7. REFERENCES 22

FOREWORD

The subject and the aim of this thesis are heavily inspired by my personal work history. While working as build manager in various projects I noticed that the selection of the version control software was often an afterthought or not thought at all. The version control system selected was usually the same that was used before, with no other fact to back up the selection. My personal opinion is that, careful selection of the version control software, backed up with technical facts, can have a positive effect in any project. I hope that this thesis encourages and helps to do so.

The aim of this Bachelor's thesis is to provide background and cover the basics of version control systems. Aim is also to categorize and differentiate the main types of version control systems, provide unbiased technical comparison of major open source version control software and at the end provide a way to map a suitable open source version control software for a software project. Aim is not to go through every technical detail, but to select the ones that can be compared and the ones that create the greatest differences between version control software.

I would like to thank my supervisor Dr. Tech. Acting Professor Jari Hannuksela (University of Oulu, Department of Computer Science and Engineering) for support, advice and feedback during the project.

Oulu, 24.3.2015

Vili Seppänen

ABBREVIATIONS

CVS	Concurrent Versions System

1. INTRODUCTION

Not so long ago there was only one major player in the open source version control software field and that was CVS (Concurrent Versions System). CVS started its life as a bunch of shell scripts in 1986 and evolved from that to be the de facto open source version control software with barely any competition until the rise of Subversion in the mid-2000s. Subversion began growing its user base and surpassed CVS. New de facto open source version control software was born. Again, there was no real competitor to challenge Subversion in quite a while. It was not until 2005, with announcement of Git and Mercurial, when Subversion met its major competitors.

Probably because of these long reigns of CVS and Subversion, they have grown large amount of devoted supporters. Git and Mercurial have grown their own supporter base through big open source projects. Large supporter bases and competing software usually lead to quite polarized atmosphere, but with CVS, Subversion, Git and Mercurial it does not end there. From the start, Git and Mercurial were created as competitors, to be the next Linux kernel version control system, Git winning at the end. This obviously created friction between Git and Mercurial camp. Another factor that has created discord in the open source version control scene, is not so praising comments about CVS and Subversion by Git creator Linus Torvalds [1]. This all adds up to a very opinionated environment where it is hard to find unbiased comparison between different open source version control software.

The aim of this Bachelor's thesis is to provide background and cover the basics of version control systems. The goal is also to categorize and differentiate the main types of version control systems, provide the unbiased technical comparison of major open source version control software and at the end provide a way to map a suitable open source version control software for a software project. The aim is not to go through every technical detail, but to select the ones that can be compared and the ones that create the greatest differences between version control software. Selection of the open source version control software is restricted to the four most popular ones.

2. VERSION CONTROL SYSTEMS

2.1. Background

In its simplest form, a version control system provides a basic principle and method of storing files and changes done to them. This is achieved by using a repository. The repository contains the most recent version of each file and the change history that has led to that representation. Usually every change includes additional information such as the author and short description.

This all enables the version controls system to provide its most important features. It helps each contributor to understand the evolution to the current state of the software. It helps each contributor to understand the solutions selected. In addition, it states the author of each change. History tracking also enables to re-create earlier states of the software by removing the changes done after the wanted state. One more important feature is that the version control system provides a way for multiple contributors to work on the same project or even on the same file at the same time in organized manner [2]. The approach how this is done is the main differentiator between version control system types.

2.2. Main types of version control systems

Version control systems can be divided into centralized and distributed version control systems by studying the way they handle repositories and sharing of changes between contributors.

A centralized version control system relies in one central master repository. Development is done against checkouts taken from the master repository. Checkout is a copy of files of the situation of master repository at the moment checkout is created [3]. After checkout is created, it can be updated to receive the latest changes from the master repository [4]. Contributors create changes against the checkout and after the change is complete, it is committed. When change is committed, it is uploaded to the master repository and if it does not conflict with other changes in master repository, it is accepted in [2]. Other contributors receive the change from the master repository when they update their checkout. This workflow is presented in Fig. 1 [5].

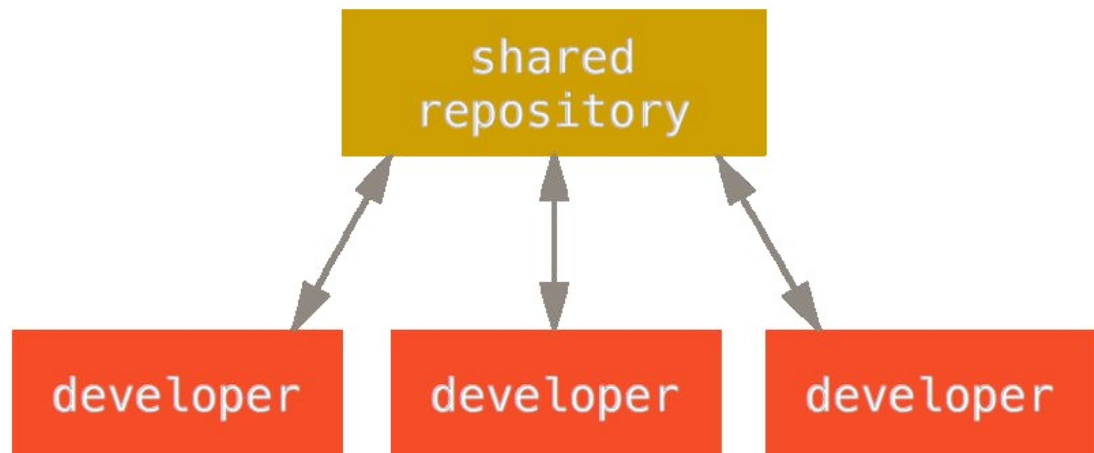

Figure 1. Centralized Workflow.

Because centralized version control systems rely on one repository that contains the correct state of the project, it is common practice restrict write accesses so that only trusted contributors are allowed to commit changes [3, 5]. This is not usually a problem in commercial development where all of the developers can be considered trusted, but can lead more difficult ways of sharing solutions in open source development where you cannot consider everybody a trusted contributor.

A distributed version control system does not require the central repository; instead, each checkout is a repository by its own right, containing the files and complete history. Each contributor commits changes in one's local repository. Change can then be shared by providing address and access to that repository for other contributors so they can pull the changes they want to their own repositories or even clone the whole repository for themselves [2]. Since every contributor owns their own repository and everyone can choose what changes to pull from others, there is no need for trusted committers and the quality of the change comes more defining factor.

Version control systems provide the parallel evolution of software in form of branches. In the centralized version control, it is common to have a main branch that contains the current development. Other branches can be created, for example, to maintain the old version of the software or to develop a new feature without risking the main branch development. In new feature branches, the goal should still be to merge it to the main branch once done [3].

In the distributed version control situation is quite different. Distributed version control systems actually encourage creating a new branch for each change [2, 3]. Everybody has their own repository and therefore they own all of the branches in that repository. Because of this, no one branch can be automatically considered to contain the mainline situation of the content, like in the centralized version control. Similar development lines are still needed. This is achieved by selecting principal branches. For example, a developer group selects Lucy to maintain the main development branch as the principal branch in the blessed repository that only she has write access to. After this, Lucy is one that reviews suggested changes and selects changes to be pulled from developers and merged to the main development branch. Other developers can then pull the main development branch changes from blessed repository. The same kind of selection process is done for all needed development

lines, selected persons are called integration managers [3, 5]. This is called the integration manager workflow, presented in Fig. 2 [5].

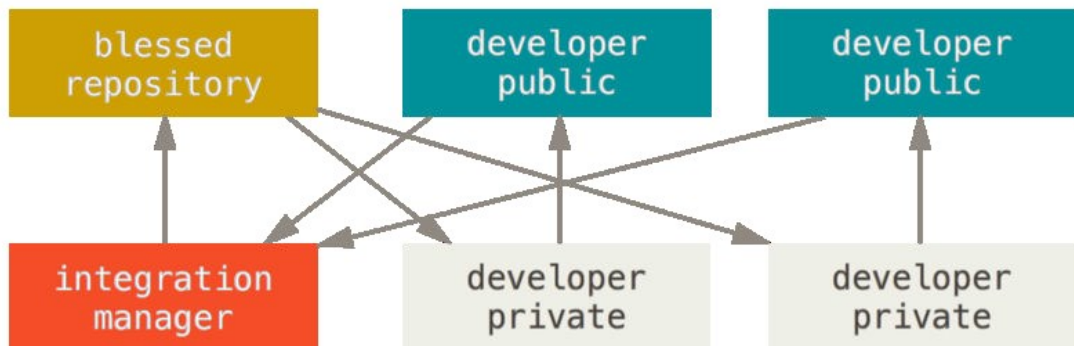

Figure 2. Integration manager workflow.

For larger projects with hundreds of contributors and massive amount of changes, it is not feasible for one person to handle reviewing and merging all of the changes to the principal branch. The benevolent dictator workflow, presented in Fig. 3 [5], addresses this problem. First, the project is divided into several parts and integration manager is selected for each part. These integration managers are called lieutenants. Lieutenants pull changes from developers, reviews and merge them in their own repositories. Above all lieutenants is one more integration manager called benevolent dictator. Benevolent dictator pulls changes from lieutenants' repositories, reviews and merges selected changes to the main development branch in blessed repository. Other developers and lieutenants can then pull the main development branch changes from the blessed repository [5].

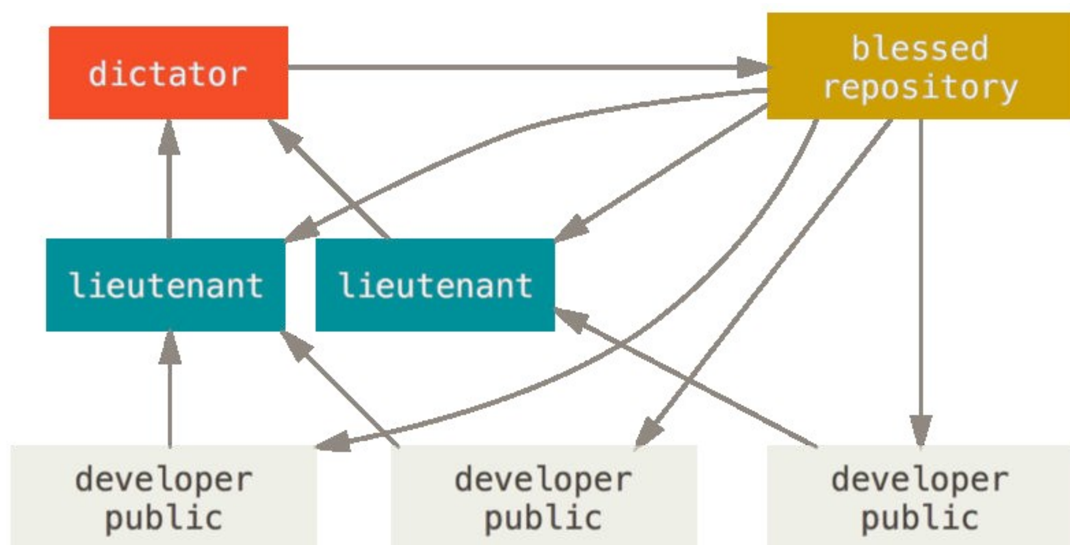

Figure 3. Benevolent dictator workflow.

A distributed version control system can also be used in the centralized manner. One or few central repositories are provided where all other repositories are cloned from and contributors are allowed to push the finished changes to the central repositories [2]. This kind of working is a challenge in respect of code correctness of

changes. For ensuring it, usually some kind of staging branches and more complicated code review methods and tools are used.

In centralized version control, the code correctness of changes has to be ensured by the professionalism of trusted contributors and by following quality terms agreed by them. In distributed version control systems, the correctness of changes relies heavily on the professionalism of the person responsible for each principal branch.

There are also differences between centralized and distributed version control in respect of content stored. Differences come most obvious when binary files are stored in version control. Version control systems use methods like storing only the differences and compressing to keep history data as space saving as possible. These methods work very well most of time, but not for binary files. Binary files do not usually compress efficiently and differences between versions can be substantial. This causes the storage space needed for history data to increase rapidly when large binary files are stored in version control and modified often. This kind of overhead is not a problem for centralized version control system since history data is only stored in central master repository, but in distributed version control system the overhead is paid in all of the local repositories [2].

Another thing where binary files do not perform well is merging. Essentially one cannot take two binary files developed parallel to each other and merge them to get one binary file with all of the changes. Development of binary file needs to be linear. In centralized version control systems, this can be achieved by the lock system in the central master repository that enables only one person at the time to edit each binary file in given branch. In distributed version control systems, the linear development of binary files has to be achieved by agreement between contributors, because the distributed version control system cannot provide the lock system [2].

3. VERSION CONTROL SOFTWARE

The most popular open source version control software are CVS, Subversion, Git and Mercurial. From these CVS and Subversion, represent centralized version control systems. Git and Mercurial fall under distributed version control systems.

CVS or Concurrent Versions System is the oldest of the four and can be considered as a baseline for open source version control software where all others are compared. CVS provides the basic functions of version control software, but not much more. Nevertheless, CVS is still popular in legacy projects because it is stable, does the job and migrating to new version control software is no minor effort.

Subversion was created to be the successor of CVS, to improve its features and to add much more, while staying as similar to CVS as possible. This helped Subversion achieve firm place among version control software and to attract a substantial user base. Currently, Subversion is the most popular centralized open source version control software [6].

Git and Mercurial development started after the free offering of Bitkeeper, the version control software used in Linux kernel development at the time, was discontinued. Both projects had a similar aim, to offer distributed, free and lightweight version control software to replace Bitkeeper, especially in Linux kernel development. The Linux kernel project selected Git and doing so secured steady foothold for it among open source version control software. Between Git and Mercurial, Git is currently more popular. The offering of both is quite similar and there is no clear agreement in which one is better. Git is widely considered more powerful in advanced use, while Mercurial is considered easier to learn and to have better Windows support.

3.1. Repository operations

Even when all of four selected version control software provides the basic functions of a version control system and the respective characteristic of its centralized or distributed nature, there are major differences between them. This can be clearly seen if version control systems are compared in respect how advanced their repository operations are. This comparison is presented in Table 1 [7].

Table 1. Version control software repository operations comparison.

Repository Operations	CVS	Subversion	Git	Mercurial
Atomic Commits	No	Yes	Yes	Yes
Files and Directories Moves or Renames	No	Yes	Limited	Yes
Intelligent Merging after Moves or Renames	No	No	No	Yes
File and Directory Copies	No	Yes	No	Yes
Remote Repository Replication	No	Yes, via tool	Yes	Yes
Propagating Changes to Parent Repositories	No	Yes, via tool	Yes	Yes
Repository Permissions	Limited	Yes	Limited	Limited
Changesets' Support	No	Yes	Yes	Yes
Tracking Line-wise File History	Yes	Yes	Yes	Yes

Atomic commits are very important for the stability of version control software. CVS is the only one of the four not to support atomic commits. This means that CVS repository can be left in inconsistent state if the commit operation is interrupted. Part of change implemented and other parts not. Not supporting atomic commits can cause serious problems if version control software is run in an unstable environment.

Moving and renaming of files and directories under version control is not the same as in the normal file system, but the history of the changes has to be preserved. CVS not supporting this causes projects using CVS to lose great deal of history data and crippling the very essence of version control if renames or moves are frequent. From the rest, Subversion and Mercurial support this feature and Git has only limited support for renames. In Git, renames are not recorded as such but renamed files are detected by the similarity of the file. This can cause problems if the object changes significantly and it is renamed at the same time.

Intelligent merging after moves and renames is supported, if the system can correctly merge changes done to the original object to the renamed or moved object. For example, two branches are created, master branch and the bug fix branch. The master branch includes change that renames the object and the bug fix branch includes a change that modifies the content of the object. When the bug fix branch is merged to the master branch, changes done to the object in the bug fix branch should then be visible in the renamed object. If version control software does not support this feature, in a project where renames are frequent and developers are not aware that this feature is not supported, it can lead to a number of costly conflicts.

Like moving and renaming objects under version control, copying emphasis the same factor, is the change history preserved? This is more of a nice to have feature than one that leads to trouble. Copying files under version control can be found useful if there is need to have two different files derived from one file in the same branch. For example, two different compilation configuration files are needed, but only with minor adjustments. In this case, it makes sense to make a copy of the original file and make the needed adjustments to the copy. If file copying was supported by version control software, both files would have full change history, but if it was not supported, the copy has only the history of changes done after the copy was created.

Ability for remote repository replication and propagating changes between repositories is especially important in geographically divided projects. Distance in physical world usually means bigger latency in network traffic and slower the

connection to the repository is more time it takes to make even the simplest operations with the version control software. This can be solved by replicating the local master repository for each production site and propagating changes between them. This way the latency is only between replicated master repositories and does not affect the user of the version control system. Naturally, this is not a problem with distributed version control since all contributors have their own local repository and can use version control regardless of network connection.

Defining different permission to different users to different parts of the repository can sometimes be important. For example, project might contain parts of software that only accredited developers should modify. In distributed version control, it is not possible to restrict developers from doing and to committing changes to the restricted parts of the software. The only way to restrict changes is not to include unwanted changes in the principal branch, but this does not restrict developers from sharing unofficial changes with each other. Even if this kind of sharing does not affect the principal branch, it can lead to situations where other changes are dependent on unofficially developed changes.

Changeset is a way to group number of modifications, which are relevant to each other and modify a single or multiple files, to one atomic change. In Subversion, Git and Mercurial changesets are created on each commit. In CVS, changes are file specific and changesets are not supported. Changesets enable much easier way of sharing complete solutions or fixes instead of sharing number of separate changes to different files.

3.2. Technical Status and Interfaces

Considering technical status CVS, Subversion, Git and Mercurial are all at a good level. The comparison of version control software technical status is presented in Table 2 [7]. All four software are mature. Documentation is widely available and extensive. Although Git documentation can be confusing at times. Mercurial excels in the ease of deployment, providing binary packages for all popular platforms. CVS and Git come near behind, Git needing some extra work when used with Windows. Subversion is not as simple to install, server side requires additional installation of its own proprietary server or the Apache 2 module.

Table 2. Version control software technical status comparison.

Technical Status	CVS	Subversion	Git	Mercurial
Documentation	Excellent	Very good	Good	Very good
Ease of Deployment	Good	Moderate	Good	Excellent
Command Set	Good	Good	Excellent	Good
Networking Support	Good	Very good	Excellent	Excellent
Platform support	Good	Excellent	Good	Excellent

The real difference in technical status can be found when investigating the command set and platform support. Even though they all have good command sets,

Git goes further and provides a very feature rich command set. This is usually stated as the tipping point in favor of Git.

Subversion and Mercurial support the most common operating systems effortlessly. CVS also supports most platforms on the client side, but the server is designed to run on UNIX derived operating systems. Git supports UNIX derived platforms well, but need extra effort to run on Windows.

When using any software, interface is important. Comparison of interface availability is presented in Table 3 [7]. CVS, Subversion, Git and Mercurial all provide a command line interface. This is a very important feature considering automation. In addition to this, there are web interfaces and normal graphical user interfaces available for all four, included either in the basic installation or as an additional package.

Table 3. Availability of version control software user interfaces.

User Interfaces	CVS	Subversion	Git	Mercurial
Availability of Web Interface	Yes	Yes	Yes	Yes
Availability of Graphical User-Interfaces	Yes	Yes	Yes	Yes

4. PROJECT DEMANDS FOR VERSION CONTROL SYSTEM

Version control systems are mostly used in software projects and characteristics of software projects have much variance. Software projects can be open source community projects, closed source commercial projects or something in between. They vary from local projects where all developers work in the same physical space to geographically divided multisite projects. Software projects can be small or large in respect of developers and amount of code. Project developers can use variety of operating systems, and tools used in project has to provide support for those, or the project might dictate the supported operating systems. Naturally, this variation of software projects also reflects to the requirements of version control systems and therefore it is important to map the special needs of each major characteristic.

4.1. Open and closed source projects

The key for a successful open source project is building up a community. This sets certain demands for the version control system. The version control system should be easy to learn, so it would not discourage anyone from participating. Sharing and developing the solutions and bug fixes together is essential, therefore sharing code changes should be effortless [8]. On the other hand, no one except the project owner and separately selected people can be considered as trusted developers and granted full access to the master copy of the project. The version control system should be able to handle this restriction without sacrificing other usability.

Ease of sharing clearly points towards distributed version control systems. From distributed version control software, more gently sloping learning curve favors Mercurial. Restricting access is easy with centralized version control systems and with distributed version control systems the master copy can be protected by using principal branches. Considering these facts, it can be said that Mercurial provides very good support for the characteristics of open source software project, Git good support and Subversion and CVS moderate.

Closed source projects are different. Only selected developers are allowed to participate in the project. These selected developers are trusted with access to the master copy of the project and expected to create quality changes. An easily approachable version control system and effortless sharing is nice to have features, but not specially demanded by the nature of closed source development. Overall, there are no special demands for version control software, derived from the nature of closed source software project. Therefore, CVS, Subversion, Git and Mercurial can be considered to provide very good compatibility with the characteristic demands of closed source project.

4.2. Local and multisite projects

When considering the demands for a version control system in respect of local and multisite software projects the situation is quite similar to comparing closed and open source development. Local software projects do not enforce any special needs for version control software. Multisite projects either need fast and reliable connections to the central repository or good support for repository replication.

Support for repository replication rules out CVS and while Subversion supports replications via separate tool it is not at the same level of comfort as with Git and Mercurial. This leads to conclusion that, CVS, Subversion, Git and Mercurial provide very good support for the characteristic needs of local software project. Git and Mercurial provide very good support for the characteristics of multisite project, Subversion moderate and CVS poor.

4.3. Small, medium and large amount of code

If it is known from the start, that the project is going to be small in respect of the amount of code, the ease of deployment is essential. For example, if the intention is to maintain a set of small scripts, it is an overshoot to set up a central master repository and run server software to provide access to it. It makes more sense to have a local repository that is easily shared if necessary. The minimal setup and ease of sharing favors distributed systems. From medium to large projects, all version control systems are usable.

Git and Mercurial are distributed version control software and can be considered to offer very good support for the small project characteristics. CVS and Subversion are centralized version control software, require client server setup, and therefore offer only moderate support for the characteristics of projects with small amount of code. CVS, Subversion, Git and Mercurial all suits projects with medium to large amount of code well, so the compatibility with those kinds of projects can be considered very good.

4.4. Number of developers

In projects with few developers, the situation is like in projects with a small amount of code. It is unnecessarily time consuming to setup the central master repository when sufficient version control system can be achieved with distributed version control using local repositories and sharing changes between them. From that, we can conclude that Git and Mercurial provide very good support and CVS and Subversion only moderate support for the characteristics of project with few developers.

In projects with large numbers of developers, the amount of network traffic and server load, caused by developers using the version control system can grow to be substantial. Using the distributed version control system and local repositories reduces the network traffic and the server load to minimum, because most of the version control actions are handled locally. Therefore, it can be said that Git and Mercurial compatibility with the characteristics of project with large amount of developers is very good while CVS and Subversion are only moderate compatible.

4.5. Platform support

Wide platform support is essential requirement for version control software, if it cannot be dictated, what operating system all involved developers should use. In centralized systems, server side support and client side support has to be considered separately. Subversion has better server platform support than CVS, which does not really support Windows on the server side. CVS and Subversion both, have good platform support on the client side. From distributed version control software, Mercurial has better Windows support than Git, while both support UNIX derived platforms well. Therefore, Subversion and Mercurial platform support can be considered very good, while the lack of real Windows support drops CVS and Git platform support from very good to good.

5. MAPPING SUITABLE VERSION CONTROL SOFTWARE

When mapping the version control software that suits the needs of the specific project, the easiest way is to start by defining major characteristics of the project. For example, the project is a large open source project with plenty of developers, all using Linux, located in various countries. Then by using Table 4, derived from the project demand analysis and showing the comparison of version control software towards project characteristics, it is easy to rate different software in order. In this case, Mercurial would be the best choice, Git coming second then Subversion and CVS.

Table 4. Version control software compatibility comparison.

Project characteristic	CVS	Subversion	Git	Mercurial
Open source project	Moderate	Moderate	Good	Very Good
Closed source project	Very Good	Very Good	Very Good	Very Good
Local project	Very Good	Very Good	Very Good	Very Good
Multisite project	Poor	Moderate	Very Good	Very Good
Small project	Moderate	Moderate	Very Good	Very Good
Medium/Big project	Very Good	Very Good	Very Good	Very Good
Few developers	Moderate	Moderate	Very Good	Very Good
Plenty of developers	Moderate	Moderate	Very Good	Very Good
Platform support	Good	Very Good	Good	Very Good

Mercurial clearly stands out in Table 4 with very good support for all major characteristics. Actually, just by looking at Table 4, Mercurial seems like an obvious choice for any project. However, individual projects might have technical needs that can justify selecting version control software that does not have as high marks in compatibility comparison. Technical differences not considered in compatibility comparison presented in Table 5 [7].

Table 5. Technical differences of version control software.

Feature	CVS	Subversion	Git	Mercurial
Atomic Commits	No	Yes	Yes	Yes
Files and Directories Moves or Renames	No	Yes	Limited	Yes
Intelligent Merging after Moves or Renames	No	No	No	Yes
File and Directory Copies	No	Yes	No	Yes
Repository Permissions	Limited	Yes	Limited	Limited
Changesets' Support	No	Yes	Yes	Yes
Documentation	Excellent	Very good	Good	Very good
Command Set	Good	Good	Excellent	Good

Mapping the most suitable version control system using Tables 4 and 5 can be achieved by first rating different version control software in order by using Table 4. Table 5 can then be used to rule out version control software or to change the order of them. In earlier example, we got from top to bottom order, Mercurial, Git,

Subversion and CVS in the last place. Let us say that a project environment is such that atomic commit support is needed and we want to emphasize a feature rich command set. Atomic commit requirement, rules out CVS and feature rich command set moves Git before Mercurial, the order now being Git, Mercurial, and Subversion at the last place.

In addition to the factors covered by using Table 4 and 5, it is advised to further analyze the project for other factors that can influence the matching of the software project with optimal version control software. These factors can be related, for example, to the project personnel, the content that is going to be stored in version control, availability and cost of needed hardware. Defining factor being that outcome, when paired with any of the version control software, cannot be defined as good or bad, but have to be evaluated case by case. One good example is binary files. If the software contains numerous binary files and those are going to change a lot, it leads the history data of the repository to crow quite large. When using centralized version control, this overhead is paid only once but with distributed version control, it is paid in every local repository. This might or might not be an issue depending on the hardware the project has available.

6. CONCLUSION

This Bachelor thesis provides background information on version control systems and goes through the basic concepts. This was achieved with brief introduction in Chapter 2 and by deepening the understanding when new item was introduced. The second goal was to differentiate and categorize the main version control system types. This was done by investigating the way the version control systems handle repositories and by categorizing the version control systems into centralized and distributed version control systems. Third goal was to provide unbiased technical comparison of the four most popular open source version control software. To achieve this, comparable technical details were collected into tables for easy comparison and the main differentiators explained more carefully. Fourth goal was to provide a way to map a suitable open source version control software for a software project. This was achieved by defining the major characteristics of different software projects. Then the needs of each characteristic was paired with the technical features of the version control system and the version control software that best supports these specific features. Project characteristics and the level of version control software support was then paired and collected into table that can be used rate version control software by a project definition. Second table was collected from version control technical details that are not covered by the needs of projects major characteristics, but has to be considered case by case. The open source version control software suitable for specified project can then be derived by using these two tables. Considering all defined goals were met, this thesis can be seen successful. Although, for finding the optimal open source version control software for a software project, additional research might be needed.

From the technical comparison of the four most popular open source version control software, it can be seen, that CVS is no more up to bar with the others. In hindsight, one could argue that CVS could have been left out and replaced with more modern version control software. On the other hand, including CVS in comparison that clearly states its shortcomings compared with the other options might encourage CVS users to migrate to use something more modern and powerful.

For future work, interesting aspect would be to find out why Git is more popular than Mercurial, while the version control software technical comparison and the compatibility comparison presented in this thesis suggest that Mercurial is more optimal choice for most projects. Is the popularity of Git only because of its position as the version control software used in Linux kernel development or is it there technical superiority that favors Git? Future work could also extend the comparison to include more version control software or closer examination of software projects to find out more characteristics and needs that software projects impose on version control software. This could be done by examining actual software projects.

7. REFERENCES

[1] LinusTalk200705Transcript. URL: https://git.wiki.kernel.org/index.php/LinusTalk200705Transcript. Accessed 19.3.2015

[2] O'Sullivan B (2009) Making sense of revision-control systems. Communications of the ACM 52(9): 56-62. DOI: 10.1145/1562164.1562183

[3] De Alwis B & Sillito J (2009) Why are software projects moving from centralized to decentralized version control systems? Proc. 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE 2009. Vancouver, Canada, 36-39

[4] Spinellis D (2005) Version Control Systems. IEEE Software 22(5): 108-109.

[5] Chacon S & Straub B (2014) Pro Git (Second Edition). Apress.

[6] Louridas P (2006) Version Control. IEEE Software vol. 23(1): 104-107.

[7] Better SCM Initiative (2012) Version Control System Comparison. URL: http://better-scm.shlomifish.org/comparison/comparison.html. Accessed 10.3.2015

[8] Rodriguez-Bustos C & Aponte J (2012) How Distributed Version Control Systems Impact Open Source Software Projects. Proc. 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). Zurich, Switzerland, 36-39