UNIVERSITY OF GOTHENBURG

Evolution of Version Control Systems

Comparing CENTRALIZED against DISTRIBUTED Version Control models

Carl Fredrik Malmsten

Bachelor of Applied Information Technology Thesis

Report No. 2010:017
ISSN: 1651-4769

University of Gothenburg
Department of Applied Information Technology
Gothenburg, Sweden, May 2010

Evolution of Version Control Systems

Comparing Centralized against Distributed Version Control models

Carl Fredrik Malmsten
Department of Applied Information Technology
IT-University of Gothenburg
malmstec@ituniv.se

Supervisor
Bill Sullivan
Department of Applied Information Technology
IT-University of Gothenburg
bill.sullivan@ituniv.se

Abstract

A lot more projects are using a Distributed Version Control System (DVCS) for handling of source code and documentation nowadays. The reasons are many, but what's lacking is more support from companies to utilize the power of this model. In this paper I take a look at open source hosting sites, blogs and articles to find a reason as to why this model is becoming so increasingly more popular in contrast to its competitor, the Centralized Version Control System (CVCS). In doing so I try to cast light on the question whether it's possible for the distributed model to win ground against the centralized model and in that way become the future standard of version control systems (VCSs). What I found was that the distributed model is a better choice in contrast to the centralized model when it comes to working and is increasing in popularity, and that by converging these two models it would enable for more productive work in projects altogether.

***Index Terms*—Evolution, Software Configuration Management tools, Version Control Systems, Centralized & Distributed repositories, Open Source, Distributed Work**

I. Introduction

THE WORLD IS BECOMING MORE DISTRIBUTED. Distance or time zones are less of a problem and work can be conducted from another location thanks to the Internet and increased capacity for network transfers. It is now possible for people to travel easier and by that be flexible in how they work. But in order for this way of working to really be successful the tools and systems which are used need to also support distributed work [13].

SCM (Software Configuration Management) is an important area of software engineering and the VCSs which are used increase in complexity to support new functions and work being done [4]. They have evolved from being something rather unheard of four decades ago, to something which is used by nearly all developers around the world today. The improvements within software makes companies and open source developers spend a lot of time and effort on different tools and systems in order to support their needs when it comes to software development [2]. What we see today is that the DVCS is becoming more popular in contrast to the CVCS [19], however the question is why this is the case.

The goal of this research is to evaluate the two different version control models and see which model is working better, whether it being the centralized model where work is done towards a central repository or the distributed model which relies more on a peer-to-peer layout. In addition to this I will find out whether a DVCS can evolve and become a main model of choice in contrast to a CVCS, in short to see where the trend is heading. By saying this my research question is split into a two parts, both of them residing in the same area. The first question is:

Question 1: *What trend are we seeing in the usage of the distributed and the centralized model?*

and the second question being:

Question 2: *How would development become more productive by utilizing a distributed model?*

In this paper I've looked at different sites, i.e. blogs and articles written online where different version control systems have been evaluated, which utilize the two different models. By looking at how people see the different models it can show how work is preferred and what lessons were learned by using them. This data is later compared to each other to see which model has an advantage in the sense of working. In addition to this data I look at open source hosting sites, and in doing so I'll get an indication as to where we are now and in what direction we are heading.

The distributed model can change the mentality in how people can work on tasks and features within

programming and by exploring this field it will be possible to select a version control model based on how work is to be conducted. Evolution of version control systems continues and by changing model can be of interest for everyone within the software engineering industry.

In the next section I cover some basic concepts and early history of SCM and version control which is required for the reader in order to understand some of the concepts of this paper. After this I bring up research done so far and what is being done in the future by going through literature in this selected topic. In section III, "*Research Method*", I state the research method which I used to uncover my conclusion and which lays the base for the data collection. In section IV, "*Data collection*", I present the data which I have found during my investigation. In section V, "*Analysis*", I analyze the data to see where evolution has taken us so far and where we are going. Finally in section VI, "*Conclusion*", I present my conclusion and also give example on future research based on my findings.

II. Theoretical Background

By laying a strong foundation in the form of theoretical ideas and proposals I will be able to present an interesting analysis in the later section.

II-A. What is Software Configuration Management?

Software Configuration Management is about tracking and controlling items which are said to be of importance and can change in software. In software engineering it usually means to track changes done over time to several files or content of files within software, also called version control (usually done on source code) [8]. A need for it was found when there were several developers working together in projects and a standardized way of keeping track of the changes were needed. To find the cause to why something went wrong when a file was changed is a reason and to be able to go back in file revisions was of importance to either reproduce what had been done or to remove what was done [26]. The idea is to keep track of files to see if there are two (or more) developers trying to change the file at the same time and manages action in this event. If there would be no control they would overwrite each other's changes [26].

Right now, two models exist in how it is possible to work, a centralized and a distributed model.

II-A1. Centralized Model

The centralized model of SCM involves that the project has one repository where all developers are working towards. People are working in the same sandbox; if a file is changed everyone else who has access to that project can see the change. If someone sets up a new branch for i.e. testing it will be possible for everyone to see it. It means that if something changes everyone will see it, especially if something went wrong as that could break test suites which the project has [26].

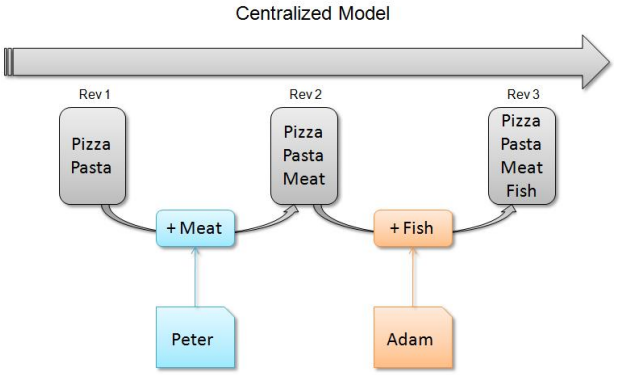

Fig. 1. Showing how a centralized model works.

As figure 1 explains and shows the centralized model is rather straight forward. Peter checks out the latest version of the repository, after which he adds the line "Meat", he then commits that file and "Meat" is added to the file within the repository. Adam in his turn checks out the file and adds "Fish" to the file and commits his change to the repository.

It started with RCS[1] which was first released in 1982, at that time this tool could only keep track of single files, it could not handle entire projects which consisted of several files and keeping track of these which for many was a problem. From this tool there were several successors, CVS[2] (1986), being the more famous one, which was a main tool of choice for many to keep track of source code or documentation for a long time. In the year 2000 many switched to use SVN (Subversion) which basically replaced CVS as the VCS to use within open source [19].

Right now there is a large base of different kinds of tools which utilizes a centralized repository to keep track of changes. A few of the bigged ones are ClearCase[3], Perforce[4] and SVN[5].

II-A2. Distributed Model

In the distributed model every developer has their branch of the project (a copy) and no one can see what the other developers are doing as it is a local copy of the project. There is no central location where the developers are working to (even if this is a possibility still). If a developer creates a branch of his project no one else will see that branch. In order for other developers to see what is going on the developer has to tell them where they can get a copy of the project that he is currently working on. Once that is done they will be able to merge his branch with their own branch and they will see [25].

[1]http://en.wikipedia.org/wiki/Revision_Control_System
[2]http://en.wikipedia.org/wiki/Concurrent_Versions_System
[3]http://en.wikipedia.org/wiki/IBM_Rational_ClearCase
[4]http://en.wikipedia.org/wiki/Perforce
[5]http://en.wikipedia.org/wiki/Subversion_(software)

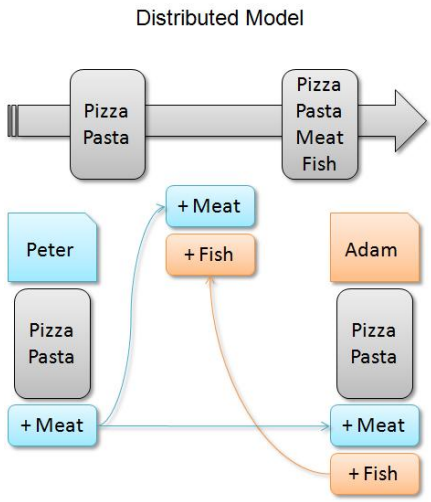

Fig. 2. Showing how a distributed model works.

As figure 2 above shows there is some difference in how work can be conducted. Peter adds "Meat" to the file and publishes his work. Adam and the main developer get the change Peter created and adds that information to their own copy of the project. Later Peter publishes his work when he adds "Fish" in the file and the main developer gets that change into their repository and merges it. It is a bit harder to comprehend how work can be conducted with a distributed model as there can be more repositories spread out during development.

One of the first DVCS was BitKeeper (BK)[6]. Development started in 1997 on this tool and is still ongoing. Other tools following up on this model are for instance Git[7], GNU arch[8] and Mercurial[9].

II-B. Related Work and Literature

One can look at this field of study from two different ways, from a technical aspect (i.e. how information is stored or different merging algorithms) or a social aspect (how the model in itself works). While both go hand in hand when it comes to VCSs I have nevertheless decided to focus on just the social aspect in my research The reason being that by focusing on the social aspects of comparing different models I focus on the differences in how work is done. I also in that way will be able to see what model has a higher probability to evolve further and be the model of choice.

II-B1. Working from a Distance

Working from different locations is done all the time. It was different before due to communication being problematic for corporations in the way that network connections weren't as widely developed. However, research in the area has been performed. Sproull and Kiesler [23] give an example of how seven teams raced to finish a deadline. A few of those teams used the network for communication (mail, distribution lists, bulletin boards) since they were not at the same location working. This resulted in a higher contribution from the team members compared to those teams which had the opportunity to meet more often.

But working distributed also has its side-effects in the way that one misses out on a lot of chatter. By being at the office and hearing other people talk or simply getting coffee can be a boost to understand "what's going on". Dourish and Bly [6] present a smaller solution where cameras are placed around the office, creating a sort of "presence" for the distributed worker; it is called "distributed awareness". By doing so, they showed that those workers who were working distributed were able to maintain their working relationships easier by utilizing that system.

Even though systems, such as the one mentioned above, can create a sort of awareness of what's going on in the team it can't beat the real thing. Orlikowski [15] mentions how important it is to meet people you are working with. In her interview with one employee at Kappa, a software company, he mentioned a working model which was called "face-to-face" where people should know their colleagues. However by using this model it created problems, firstly it was that the project manager had to travel around which is tiring in the long run and secondly the cost for this kind of travel. An external auditor at the company did not see how all this traveling and "no real work" was done, since "communication with employees" in that way was not considered to be work in his eyes. By keeping a face-to-face interaction you can ensure that you know the colleagues, knowing their credibility's and commitment to specific issues and in the end know how to collaborate together to get things done in a global distributed and complex product development environment [15].

II-B2. Open Source

Linus Torvalds, creator of Linux, starts a new way of working once he started the Linux kernel project[10]. In it he brings in more and more people to help him out, as in co-developing on the Linux kernel. In doing so he is able to find bugs faster and inspires others to do the same. This model is not similar compared to how projects were run normally, especially since people in the project were scattered around the world. Despite that fact, the project survived and it became a huge success [14].

Raymond [17] decided to try this model to see why Linux became so popular and successful. He identifies two models; a cathedral model where source code is hidden between releases and only available to an exclusive group of people (the normal model), and a bazaar model where the public can see the source code being developed. He tries the latter model with

[6]http://en.wikipedia.org/wiki/BitKeeper
[7]http://en.wikipedia.org/wiki/Git_(software)
[8]http://en.wikipedia.org/wiki/GNU_arch
[9]http://en.wikipedia.org/wiki/Mercurial_(software)
[10]http://en.wikipedia.org/wiki/Linux

his own Fetchmail project[11] and notices a positive effect almost instantly. However there is a problem with a bazaar model, that if there are many developers working with a distributed model you will create a "*bazaar of cathedrals*". This meaning that in the event that an individual developer is unable to submit an important solution for an unspecified amount of time (due to various reasons) it could make it hard for the development to continue as it should. This can severely hamper the development especially if the functionality which is being worked on was of importance for the project [10, 17].

Krafft [11] goes further in his research and states that in the Debian project[12] people are reluctant to use new innovations. He says that there is no real standardized way to what approaches should be used within the packaging and packaging maintenance as some approaches are used while others discarded quickly. In open source projects people are unwilling to change too much as they do things on their spare time [11]. This can produce problems for the development of these projects and also be a reason to the bazaar of cathedral problem stated above. Another reason for people not changing is that all version control tools aren't easy to learn. The learning curve is high and understanding how these tools work takes time [11].

This is one of the reasons as to why the Linux kernel project officially didn't use a VCS for a very long time, this caused problems. Chu-Carroll et al. [3] validates that problems existed in the Linux kernel project, which in the end lead to the adoption of the BitKeeper (BK)[13] VCS in order to avoid unnecessary problems (such as merging branches of the kernel) (see also [14, 17]). There were at the time also other areas within the project where CVS was used, however not fully adopted due to Linus Torvald's "*hate for it with a passion*"[14]. The issue for many was that BK was a commercial tool and one had to buy a license to be able to use it fully. This looked bad in many people's eyes, especially those who shared Richard Stallman's ideology for free software[15] [21]. However other problems ascended from the use of BK. People were allowed to use BK for free as long as no one copied or tried to reverse engineer it. But after a reverse engineer attempt to create a client which would show the meta-data and compare past versions[16][17] it made the creator of the BK tool to not let some of the leading Linux developers use it for free. This initiated the creation of a new VCS, Git.

II-B3. The Different Models

There are big differences between the models and how the foundation is built; this has big implications on how work is done with the different systems. Estublier [7] states that there are many VCSs which relies on a central repository but with that you get problems which can reduce efficiency and availability and that by using a distributed model where each workspace "*rely on 'its' own local store*" would make them more efficient[7, pp.283].

By being able to do local commits it is expected that more commits would be made. Bird et al. [1] hypothesized that by changing a centralized model to a distributed model there would be more commits and the commits in themselves would be fewer lines of code. Even though that was the case, it was not that big of a difference as apparently only 2 lines less per commit was noticed.

Since it's possible to commit locally with the distributed model developers may not know what artifacts are being worked on in parallel. This can produce conflicts once the work is either uploaded to a central repository or when merges are done between different branches. But once these problems surface there are other tools available to minimize the work needed to resolve these conflicts (i.e. diff tools for comparing different versions of the files/content and the possibility to auto-merge these) [16, 20].

Conflicting files is however not the biggest problem. DVCSs can be seen as more difficult to grasp compared to CVCSs. "*With CVC changes flow up and down (and publicly) via a central repository. In contrast, DVC facilitates a style of collaboration in which work output can flow sideways (and privately) between collaborators, with no repository being inherently more important or central.*"[18, pp. 26]. This meaning that it is harder to grasp the concepts of how a DVCS works.

But Lundell et al. [13] states that companies can receive a lot of benefits by using a DVCS. Most examples where a distributed model has been in use have been success stories but only as long as the team keep good communication channels between the different departments. Being distributed invokes more communication between departments to ensure that problems such as these are removed [13].

However, in order for these systems to evolve people who use them need to have a better understanding of the systems they already use. By supporting this need they can demand better supportive implementations [5] and the systems will eventually evolve. Dart [5] also states that for future VCS to be fully utilized, management in companies need to realize that the systems are complex and in turn that means that it will be expensive to implement solutions which are better than their current ones.

III. Research Method

This research aims to look into the VCS market to find a possible answer as to where the VCS development is going by comparing two different models, the centralized model which uses a server for storing information and the distributed which does not. By looking at open source hosting sites and examples of

[11] http://en.wikipedia.org/wiki/Fetchmail
[12] http://en.wikipedia.org/wiki/Debian
[13] http://www.bitkeeper.com/
[14] http://www.youtube.com/watch?v=4XpnKHJAok8
[15] http://en.wikipedia.org/wiki/Free_software
[16] http://en.wikipedia.org/wiki/BitKeeper#Pricing_change
[17] http://www.linux.com/archive/articles/44147

usage from blogs etc. and see the results of these it will be possible to give an analysis of the compared result.

III-A. Method of choice

The pragmatism philosophy[18] is chosen as I am comparing two different models to each other based on statistics and facts from literature which have been selected. I believe, like the pragmatism philosophy states, that if something works it has to be true, and that is my goal with this research. In other words it is a "common sense" philosophy since "*actions are assessed in light of practical consequences*" [22, pp. 197].

Furthermore, this research utilizes the Content Analysis[19] method.

Content analysis is defined as "*analysis of the manifest and latent content of a body of communicated material (as a book or film) through classification, tabulation, and evaluation of its key symbols and themes in order to ascertain its meaning of probable effect.*" [12, pp.xvii].

There are two types, a quantitative content analysis and a qualitative content analysis. The first model focuses on stripping the text of words, counting word frequencies, space measurements (column centimeters in i.e. newspapers), time counts (radio/TV times) and keyword frequencies [24]. The second one is to categorize and classify the text [24], which is what will be done in this paper.

The content analysis method is used when text is to be reviewed and summarized. Graneheim and Lundman [9] explain how a text can be summarized into several smaller parts by picking elements from it. First the whole text is presented where after the next step is to make it smaller in two parts. This is called "condensed meaning unit" where the text is similar to the original text but written like a summary. The second step is again to make it smaller but where the text is written in a different way with the same meaning. Then one can classify the whole meaning in a sub-theme what the text is about, written in a few words, and lastly a classification of the whole theme of the whole text.

III-B. Data Collected

By looking at many of the open source hosting sites where projects are stored I were able to retrieve data on how many sites supports a certain model and by that fact I would know what model is more popular and could witness a trend within open source projects.

By reading blogs and articles it was possible for me to gather information related to any of these models and see what kind of praise and critique they receive. However, with that in mind it is important to separate users which are just "hyping" their own favorite tool and users which actually state their opinion based on facts. With this data I was able to produce a table of information, listing pros and cons with the different models and by that produce an analysis of the models in general. Collection of the quotes from the sites which are listed in the data collection section happened before 10th of May 2010.

[18]http://en.wikipedia.org/wiki/Pragmatism
[19]http://en.wikipedia.org/wiki/Content_analysis

III-C. Process of collecting data

By reviewing and taking notes of arguments others have made will provide me with this necessary foundation of knowledge that I seek. Databases used for obtaining literature were: Association for Computing Machinery[20], Google Scholar[21], GUNDA[22], Libris[23] among others. Keywords used to obtain literature were i.e. "distributed model", "centralized model", "software configuration management", "version control systems".

With the literature reviewed I was able to continue to the more statistical analysis where I went through various open source hosting sites in order to get information on what model is the more popular.

Facts were after that drawn from different blog posts and articles (from now on referred to as "sites"). The selection of these sites has been merely by searching on Google with keywords shown above but by adding keyword i.e. "blog" in the search. The search was also conducted at http://www.google.com and not a scientific search engine. I based my selections on that the blogs had something of importance to say. Basically meaning that there was a serious undertone and valid arguments for whether one model was better than the other one.

After the collection of data had been made I related this to the literature to answer my research questions.

IV. Data collection

In this section and subsections I present data which have been collected during the research. The data has been separated into different subsections in accordance to its relevance.

IV-A. Open Source Projects

Open source projects have a tendency to push the boundaries when it comes to new technology within software engineering [27] and for that reason there exist many different open source hosting sites where new projects can be stored. These sites compete with each other to support the most projects and thus they try to maintain as many different VCSs as possible.

There are several different hosting sites available and all of them support different solutions for revision control (see table I for a review of the hosting sites and II for the amount of different systems used).

[20]http://portal.acm.org/
[21]http://scholar.google.com
[22]http://www.ub.gu.se/gunda/
[23]http://libris.kb.se/

TABLE I
SUMMARY OF THE LARGER OPEN SOURCE HOSTING SITES

	Number of		Version Control Systems	
Name	**Users**	**Projects**	**Centralized**	**Distributed**
Assembla[a]	170,000	60,000+	SVN	Git Mercurial
BitBucket[b]	35,000	19,100	-	Mercurial
CodePlex[c]	151,782	9,274	SVN Microsoft TFS	Mercurial
GitHub[d]	228,000	747,000	-	Git
GNU Savannah[e]	60,869	3,023	CVS SVN	Bazaar Arch Git Mercurial
Google Code[f]	N/A	250,000+	SVN	Mercurial
Launchpad[g]	1,061,601	17,140	CVS (import only) SVN (import only)	Bazaar Git (import only) Mercurial (import only)
Project Kenai[h]	46,000+	10,000+	SVN	Git Mercurial
SourceForge[i]	2,600,000+	161,992	CVS SVN	Bazaar Git Mercurial
Tigris.org[j]	137,324	1,547	CVS SVN	-

[a] http://www.assembla.com/, [b] http://bitbucket.org/, [c] http://www.codeplex.com/, [d] http://github.com/, [e] http://savannah.gnu.org/, [f] http://code.google.com/, [g] https://launchpad.net/, [h] http://projectkenai.com/, [i] http://sourceforge.net/, [j] http://www.tigris.org/,

TABLE II
TOTAL COUNT OF VCS SUPPORTED AT OPEN SOURCE HOSTING SITES

	Centralized Models			Distributed Models			
	CVS	Microsoft TFS	SVN	Arch	Bazaar	Git	Mercurial
Total Count:	4	1	8	1	3	6	8

The selection of these sites was done based on number of users each site had and number of projects. If the addition of these surpassed the count of 50,000 it was added to the table[24].

Within these 10 hosting sites there are in total three different centralized systems and four different distributed systems. In total the count for centralized systems are 13 (SVN count being 8) while the distributed model counts for 18 (Mercurial count being 8).

By just looking at the different systems it is possible to see an increase in the distributed model. Table I along with table II shows all the different systems available on the major open source sites and there is a significant larger amount of DVCSs compared to CVCSs. In reality it's just CVS and SVN which are used in a larger amount witnessed from a centralized model's perspective. While CVS is being outdated, SVN is increasing in popularity, this mainly as it's seen as the descendant from CVS. A noticeable view of this can be seen in the amount of sites supporting the different systems. Four sites having support for CVS while eight having support for SVN. While saying that, it's crucial to know that at those four sites where support exists for CVS, there also exists support for SVN.

In contrast to the centralized model, the distributed models are rising in number of sites which supports this model. There is also a larger flavor of DVCSs to select from, the reason for this is that many of the systems which are in use were started almost at the same time, Git and Mercurial being those systems in particular. Development of these systems were started almost instantly after licenses from the DVCS known as BitKeeper, was dropped in the Linux kernel project, making them the top two systems.

IV-B. BLOG/ARTICLE POSTING

A lot of public attention on these sites are given to different VCSs. Data obtained from these sources are coming directly from users of the different systems (see table III for address's).

While I investigate the different models people have a tendency to talk about the different systems instead. So they may say the names such as Subversion and Git instead of using the terms "centralized" and "distributed". Even so it presents the different models because even if the two distributed systems Git and Mercurial have different functionalities they still have the same foundation in the term of distribution. Functionalities per system (as in the difference

[24]Based on the following article http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities

TABLE IV
PROS AND CONS FROM DIFFERENT SITES

	Centralized model		Distributed model	
	Pros	**Cons**	**Pros**	**Cons**
Site 1	**1.1**: Simple centralized repository. **1.2**: Can handle binary files by utilizing a locking function on the files.	**1.3**: Operations are slow due to functions are done over network connection. **1.4**: Can't do local work. **1.5**: Branches are hard.	**1.6**: Fast to do work (no network connection needed). **1.7**: Can do local work. **1.8**: Branches are easy to make.	**1.9**: Too easy to make branches (can become a forest of branches). **1.10**: Flexible to work but can be dangerous if not used well. **1.11**: Most teams still use DVCS as a CVCS.
Site 2		**2.1**: Branches and tags convention too visible and hard to merge. **2.2**: No offline commits.	**2.3**: Commit work to local repository and push changes to others when you feel ready.	
Site 3	**3.1**: Forced to review each other's work.	**3.2**: Can only commit if you have privileges. **3.3**: Merging is done harder.	**3.4**: Every checkout is a copy of the repository. **3.5**: Can work without a network connection. **3.6**: Working offline is fast. **3.7**: Possibility to clean up your local commits if a mistake is made. **3.8**: Can work in small steps. No need to commit everything at once. **3.9**: Branches are easy.	**3.10**: Anti-social behavior. Have to put in extra work in order to share.
Site 4	**4.1**: One repository to keep track of. **4.2**: Repository doesn't allow conflicts.	**4.3**: No local history.	**4.4**: Good branching. **4.5**: Fast, work is done locally.	
Site 5		**5.1**: Can't commit to repository unless user has access.	**5.2**: No central repository. All working copies is a clone of the repository itself. **5.3**: Every copy is a branch and commits are only done in that branch. **5.4**: Can work without a network connection. **5.5**: Not dependent on external server for work **5.6**: No need to install or maintain a server. **5.7**: No privileged access required for doing work.	
Site 6	**6.1**: Branches are easily seen.			**6.2**: Too many repositories to keep track of in projects. **6.3**: Work becomes hidden since developers don't need to push their changes until they are done.

between i.e. Git and Mercurial) will not be presented as pros or cons.

In total, six different sites have been selected where each site presents pros and cons against the different models, many of these sites do however present the same information. The text from these sites can be found in appendix A, these texts are taken direcly from the sites and presented as quotes within either pros/cons for the centralized/distributed model. At the end of this section, two tables are presented, the first summarizes what has been said on these sites (see table IV), the second places each of the pros and cons in a related category (see table V). The pros and cons have also been numbered as I will refer to these later in the analysis section.

V. ANALYSIS

In this section I analyze and discuss the data which I have collected with the literature that I have gathered. The structure follow the same as in the data collection section, discussing open source projects

TABLE III
SITES AND THEIR URL ADDRESS

Site	Address
Site 1	http://java.dzone.com/articles/version-control-tools
Site 2	http://www.dribin.org/dave/blog/archives/2007/12/28/dvcs/
Site 3	http://www.wincent.com/a/about/wincent/weblog/archives/2007/10/why_distributed.php
Site 4	http://www.dehora.net/journal/2008/04/06/what-a-dvcs-gets-you-maybe/
Site 5	http://pointbeing.net/weblog/2009/09/git-for-subversion-users.html
Site 6	http://blog.ianbicking.org/distributed-vs-centralized-scm.html

first where after discussion of the different sites and its pros and cons are brought up. Finally a section which summarize and points out other important areas within this field is presented.

V-A. Open Source Projects

By looking at the open source hosting table above we see a larger base of systems supporting the distributed model compared to the centralized model, so why is the distributed model becoming so much more popular now on open source hosting sites? One large issue when it comes to open source projects is the notion of commit-access. In a CVCS people need access to commit and do work, this has implications on security and means that one need to create user groups of who gets to commit and where they get to commit. However, in a DVCS the term commit-access doesn't really exist. This since there is no real central location to where people do commits, work is committed locally instead. After which the other developers can get the work of their developers by simply pulling it from them and later merging. This also has implications in how work is done, the model changes from traditional working methods, working towards a central repository to working towards no central location. However a central location can still be created in many of the DVCS which exist, and then one do get the problem of who gets commit-access.

Seeing as open source projects can be rather large in size of developers it is also important to make it possible for those who do not have access to help out. With a CVCS where an outsider can't commit they have to do all the work at once. Later when they are done with their bug-fix or feature improvement which they have been working on for the last week and want to send it somehow to the main developers it creates a problem. By that they would have to send in all the source code files and the main developer would have to manually check and verify the differences. In contrast, with a DVCS, where work can be committed locally, one can work for a week, do local commits, do improvements and later ask the main developer to get their changes from a location. Once they do, it's a matter of merging the changes into the code automatically, doing work this way saves a lot of time and also helps the outsider to contribute.

But open source projects also have ramifications on how close to each other developers work. In most projects the developers are situated at different locations and meet rarely. And even if the best is to work close to each other there are ways to improve the collaboration to overcome distance, i.e. "distributed awareness" [6] but also by meeting from time to time as they do in distributed development within companies [15]. But even so, projects conducted as tests to witness the effects distribution had on a team was even reported to be a better way of working [23]. This was noticed during the Linux kernel project, and later Raymond [17] also saw this when working distributed that it has its benefits. So by letting more people help out in projects it will be possible to find more bugs and develop more features in projects and by doing so the programs which are developed will be better. This is something which is made easier with DVCSs as they increase the usability for a developer which does not necessarily have access to the central repository but that would like to help out anyway. That's also a reason as to why so many hosting sites support DVCSs as they see an increase of popularity from these systems.

V-B. Blog/Article Postings

All pros and cons from the sites which were found can be divided into four categories based on their themes, these are:

- *Location*
- *Development*
- *Network*
- *Security*

Table V summarizes all the pros and cons and puts them in accordance to their theme. These categories form points of discussion, thus this section has been divided into four subsections, each focusing on the separate theme.

V-B1. Location

The centralized model is very straight forward in its working; this is also noticed by the pros listed for the CVCS. A (**1.1** & **4.1**) "simple centralized repository" is mentioned as a pro, focusing on the fact that it takes less effort to find where to commit changes or get the latest changes. This is very different compared to how DVCSs works or at least can work. For a DVCS it's listed as a pro and con (**5.2 & 5.5**) that there is no central repository. It's seen as a benefit in the way that there are backups everywhere. If one looses ones work it's easy to obtain a copy of it. This is a benefit in the event that the central server, hosting the only valid copy of the full repository, gets corrupted; it creates a big problem for the developers. While in a DVCS everyone has a copy of the project which makes it easy to get a backup of the project if this would happen. In contrast it is seen as a con that there are too many places to keep track of as it's a lot easier to

TABLE V
PROS AND CONS FROM FROM ALL SITES SUMMARIZED INTO THEMES

	Centralized model		Distributed model	
	Pros	**Cons**	**Pros**	**Cons**
Location	**1.1**: Simple centralized repository. **4.1**: One repository to keep track of. **4.2**: Repository doesn't allow conflicts.		**5.2**: No central repository. All working copies is a clone of the repository itself. **5.5**: Not dependent on external server for work **5.6**: No need to install or maintain a server.	**6.2**: Too many repositories to keep track of in projects.
Development	**1.2**: Can handle binary files by utilizing a locking function on the files. **3.1**: Forced to review each other's work. **6.1**: Branches are easily seen.	**1.5**: Branches are hard. **2.1**: Branches and tags convention too visible and hard to merge. **3.3**: Merging is done harder. **4.3**: No local history.	**1.8**: Branches are easy to make. **2.3**: Commit work to local repository and push changes to others when you feel ready. **3.4**: Every checkout is a copy of the repository. **3.7**: Possibility to clean up your local commits if a mistake is made. **3.8**: Can work in small steps. No need to commit everything at once. **3.9**: Branches are easy. **4.4**: Good branching. **5.3**: Every copy is a branch and commits are only done in that branch.	**1.9**: Too easy to make branches (can become a forest of branches). **1.10**: Flexible to work but can be dangerous if not used well. **1.11**: Most teams still use DVCS as a CVCS. **3.10**: Anti-social behavior. Have to put in extra work in order to share. **6.3**: Work becomes hidden since developers don't need to push their changes until they are done.
Network		**1.3**: Operations are slow due to functions are done over network connection. **1.4**: Can't do local work. **2.2**: No offline commits.	**1.6**: Fast to do work (no network connection needed). **1.7**: Can do local work. **3.5**: Can work without a network connection. **3.6**: Working offline is fast. **4.5**: Fast, work is done locally. **5.4**: Can work without a network connection.	
Security		**3.2**: Can only commit if you have privileges. **5.1**: Can't commit to repository unless user has access.	**5.7**: No privileged access required for doing work.	

just have one central location to receive the updates in the project instead of having 15 different "central locations" (**6.2**).

An aspect brought up for the centralized model which works well is that the repository doesn't allow conflicts (**4.2**), this is true and also valid for a DVCS that it doesn't, even if not listed as a pro. One always has to resolve conflicts before work can be committed, and is not generally a point of truth for just one model. However conflicts can surface easier in a distributed model due to parallel work being conducted [16, 20] and once the work is merged it is possible that one could mess up the local repository more and in that way it is negative for the distributed model.

To not having to setup a server and not having to maintain it during a project is one large and important pro for the distributed development (**5.6**). Setting up rules i.e. for how and when one can commit and make sure that the server is up and running, takes time from the development. One should focus on what's really important and not having to think about these smaller issues. In the event that the central repository goes down it will basically put development to a halt if a project uses the centralized model, while in a distributed model it doesn't even affect or change anything.

V-B2. DEVELOPMENT

An aspect, which is both seen as a pro and a con, is the visibility of branches in centralized models (**6.1 & 2.1**). Reason being that if one can see them easy it helps that you know they exist, however it also adds unnecessary information to the repository as

the branches are global in a centralized model. Even though, branches are pointless unless one can merge them, which is usually harder to do with a CVCS (**1.5 & 3.3**).

The distributed model is seen as a better alternative when it comes to branches, this mainly since every checkout is a branch in itself (**1.8, 3.4, 3.9, 4.4 & 5.3**). But also due to the fact that branches are seen as something which is natural within the distributed model, which means a lot of focus is put on merging. In contrast to the centralized model, merging works very well and is done very often when developers pull code from one another. However, since it's very easy to create branches it can also make work quite messy which can be seen as quite negative and also makes it dangerous as one could get lost (**1.9 & 1.10**).

One feature which the distributed model lacks is to lock files from being edited by anyone else, a feature which the centralized models are able to utilize with ease (**1.2**). It's not viable in a distributed model since there are or can be so many different repositories which make it impossible to lock specific files. By having a central server where everything goes, developers are also forced to review each other's work (**3.1**), compared to how it would be in a distributed model. There are or can be repositories everywhere which makes it harder to review each other's code. Code reviews can happen once merges are actually done between the different repositories.

This makes us dive into another similar problem with the distributed model. By being distributed it makes people work "offline" more, which in turn can create problems in that people "hide in their caves" and doesn't make much noise until their work is published (**3.10 & 6.3**). This relates to a problem which Raymond [17] noticed and that Krafft [11] later built upon, the "bazaar of cathedrals". It can be negative for the entire project, especially if the function which is being worked on is of importance. It would be beneficial if developer-advertising would be done, stating who's working on what, which is usually done by open source projects (i.e. bug tracking tools or mailing lists), but not always done well enough. It is very important to keep good communication while working distributed [13], and only then is it possible to succeed fully with a DVCS. Although relating to this the possibility to keep a local history in distributed projects creates a somewhat solution to when the work is submitted. Since it's possible to commit locally it is possible for other developers once they can review the work to see how they came to their solution. In that way one can follow the flow in a more exact detail even if they are not communicating that much during the development (**3.7 & 3.8**). At the same time the function which was developed on is more certainly working by the time it is pushed to other developers (**2.3**). In contrast the centralized model doesn't support this feature as commits have to be done to the central repository, which is a drawback (**4.3**).

Most distributed models are used as a centralized model. This is seen as a con (**1.11**) as in that way the model is not taken full advantage of. In most DVCSs it is possible to work as if one would use a CVCS; while it isn't optimized for that sort of usage it is still more viable since one can do local commits. The same thing can be seen on the other side. SVN, while not being able to commit locally, also have received extensions (such as SVK[25]) which enables the possibility to do local work. However while not many of the two different models do each other's work good as of yet, this is most likely where the two models are heading.

V-B3. Network

An aspect which is seen very negatively upon, at least after having tried a distributed model, is when work has to be done over the network. Cons for the centralized model are all in regards that work can't be done locally and everything has to be performed over a network (**1.4 & 2.2**). While Internet connections are getting more widely spread out over our world it can still take time to do work over the network. Maybe 10 seconds isn't such a long time to do commits or to retrieve the latest changes from one revision to another (**1.3**), but compared to the distributed model, where everything is saved on the computer (**5.4**) and having an ability to do commits and retrieve logs between revisions in less than a second[26] (**1.6, 1.7, 3.6 & 4.5**), it becomes a rather big difference. It also affects the usability as was mentioned in point **4.5** that your mind can "wander away" which could break the flow in the work.

V-B4. Security

Security of who gets to commit, especially in open source projects, was discussed in the previous section more extensively and seen as one large reason as to why more distributed models were seen within open source hosting sites. The centralized model states that this commit-access is necessary in order to keep a project structured, and it does so to a certain aspect. The problem is shown when it's realized that the group of people who gets to commit is either too small or too big, and this has implications on how fast a project moves. By being able to commit is a big aid for developers if they want to help out in open source projects. This is something that the distributed model supports, although if it's only locally. If a single developer wants to do a simple commit it is beneficial for them that they can do that without having to worry about not being able to commit to the central repository, since they don't have access (**3.2, 5.1 & 5.7**). If the project later wants to get that developers code or not is another issue.

V-C. Prolonged Analysis

An indication of what can be expected in the future from VCSs can be seen in the last paragraph in the

[25]http://en.wikipedia.org/wiki/SVK

[26]This depends of course on what system is used and size of project and is not always the case for all projects.

Development section above, that the two models are converging. It is now possible to work as the other model is doing, even if more work is needed to set that up. For a centralized model it means that one has to add an extension to the initial system (this at least in SVN's case), and for a distributed model it means to somehow find somewhere to set up a server. The distributed model is, so far, further ahead in this race, since its local working capabilities which it has, are harder to implement (and to grasp the concept of [18]) than it is to set up a server for collaboration and fix commit-access for developers. But an important factor is missing in the distributed model. Being able to lock files, meaning that no one else can work on those files, is a function worth having. This comes in handy once a developer need to change a binary file for instance, since if a binary file gets a conflict it's hard to resolve. And while not said to be a good way to handle files such as these, as it prevents other people to do work on those individual files, it's at least one way of doing so.

Being able to work distributed in today's world is quite common, and to do work offline is valuable as people travel on i.e. airplanes or may find themselves unable to connect to the central server which makes them to work less efficient. Valuable time would be lost, as instead of working one would curse at the absence of a connection to the central repository for viewing logs etc.

Open source projects are mentioned a lot next to the distributed model, mainly since the most attention is put there and it makes people wonder if there even is a market for companies to utilize the power of that system. It's easy to look past, but the first DVCS was a commercial system (BitKeeper), and there are many companies using this system if only on a small scale. Companies have a tendency to wanting centralized systems as that gives them more control of how work is conducted. However, by looking at working methods, such as the Waterfall model, we see how it was the main model of use for a long time before the agile models started to pop-up. The agile models can be seen as more unstructured as one does work with tasks parallel (much like the distributed model) and eventually that model has won a lot of ground as it's now used frequently in projects. So all the distributed model needs is time to get accepted and once it has been accepted it will be used more in companies for sure. It's usually like that with "new" technology; time is needed for its establishment and later acceptance. However, by using a distributed model, where work can be done locally, it could significantly speed the process up in the event that a new experimental function is required. Seeing as most companies have commit procedures, indicating that everything has to be tested before commits are made, one can by a distributed model commit locally and do all tests required before the whole function is pushed to the main repository. This means that once companies realizes the benefits of the DVCSs they will eventually want to utilize those benefits also as it is possible to save a lot of time in projects with that model. In that way work becomes more efficient, as Estublier [7] previously has stated.

VI. Conclusion

I have presented several reasons in this paper as to why a distributed model would make development more productive, the main reason being that work can be done locally without the need of a connection to the main repository. By utilizing a distributed model work can be done everywhere and by that fact it will be easier for developers around the world to conduct their work and also to collaborate in projects. It also enables backups to be made easier and creates in that way a more robust system which is safer against corruption, since every developer has a copy of the entire history of the repository and there are or can be many repositories. And this would ensure that development would not come to a halt in the event of (a) corruption of main repository or (b) that the server crashes.

Needless to say there are of course issues with the distributed model, issues which if not addressed could severely set back a project. The notion of bazaar of cathedrals and big clusters of branches to get lost in are not to be forgotten. However these issues are related to learning how to work with the model rather than holes in the model itself. Once one do learn how to work with the model, problems like these are removed. But it could prevent developers from actively want to change model, as it may prove to be more difficult to understand and in turn make companies and their managers less likely to accept a change from one VCS to another if it proves to be more complicated to use. But Dart [5] stated that management need to realize that it is expensive to implement solutions, better than the current ones. So even if it is hard to implement something like this due to being harder to comprehend, it would pay off in the long run.

Companies and managers desire control in projects and with a distributed model less control is given to them, so although the distributed model in itself is a better choice for developers it takes away the control from others. There are restrictions and all companies can't support a distributed model due to corporation secrets and a distributed model could prevent them from sleeping well at night. So even if the trend shows us that it's heading towards a distributed model the model can't be applied everywhere. Some companies even want to reduce developers access by only allowing them access to the source code by just letting them work by remote accessing the repository, thus giving companies complete control[27].

Many projects which use a distributed model still use it as a CVCS, meaning that they have a central location to where they are working. A reason for this can be that it's more familiar to use it like that first for the developers and after a while go over to a more actual distributed model, but then good communication

[27]Article in swedish: http://www.idg.se/2.1085/1.312963/accenture-chef-centralisering-ersatter-standardisering

is required to be established between the developers. Another reason is that it "feels" right for developers to have one central location to go to where everything is "up to date", and for that it seems like the two models are converging into one, a global VCS, which has both models benefits. Taking the strengths from both camps, and it seems like the trend is heading in that direction.

For future research it would be suggested to evaluate work conducted with different solutions for a converged model which utilize both sides' strengths and in that way push evolution forward even more.

References

[1] C. Bird, P.C. Rigby, E.T. Barr, D.J. Hamilton, D.M. German, and P. Devanbu. The promises and perils of mining git. In Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, pages 1–10. IEEE Computer Society, 2009.

[2] Tilmann Bruckhaus. Tim: a tool insertion method. In CASCON '94: Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research, page 7. IBM Press, 1994.

[3] M.C. Chu-Carroll, D. Shields, and J. Wright. Version control: A case study in the challenges and opportunities for open source software development. In ICSE 2002, 2002.

[4] Ian Clatworthy. Distributed version control systems - why and how. 2007.

[5] S.A. Dart. The past, present, and future of configuration management, 1992.

[6] P. Dourish and S. Bly. Portholes: supporting awareness in a distributed work group. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 541–547. ACM, 1992.

[7] J. Estublier. Software configuration management: a roadmap. In Proceedings of the conference on The future of Software engineering, pages 279–289. ACM New York, NY, USA, 2000.

[8] J. Estublier, D. Leblang, G. Clemm, R. Conradi, A. van der Hoek, W. Tichy, and D. Wiborg-Weber. Impact of the research community on the field of software configuration management. 2002.

[9] UH Graneheim and B. Lundman. Qualitative content analysis in nursing research: concepts, procedures and measures to achieve trustworthiness. Nurse education today, 24(2):105–112, 2004.

[10] M.F. Krafft. Workflow in distributed volunteer projects — Intuitive approaches to modern Debian package development. Phd proposal, Department of Computer Science and Information Systems University of Limerick, Ireland, October 2005.

[11] M.F. Krafft. Method diffusion in large open source projects. Phd research proposal, The Irish Software Engineering Research Centre, CSIS, University of Limerick, Ireland, August 2006.

[12] K. Krippendorff. Content analysis: An introduction to its methodology. Sage Publications, Inc, 2004.

[13] B. Lundell, B. Lings, P.J. Ågerfalk, and B. Fitzgerald. The distributed open source software development model: Observations on communication, coordination and control. In Proceedings of the 14 th European Conference on Information Systems (ECIS 2006), Gothenburg, Sweden. Citeseer, 2006.

[14] G. Moody. Rebel code: How Linus Torvalds, Linux and the open source movement are outmastering Microsoft. Allen Lane, 2001.

[15] W.J. Orlikowski. Knowing in practice: Enacting a collective capability in distributed organizing. Organization Science, pages 249–273, 2002.

[16] B. O'Sullivan. Making sense of revision-control systems. Communications of the ACM, 52(9):56–62, 2009.

[17] E.S. Raymond. The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2001.

[18] P.C. Rigby, E.T. Barr, C. Bird, D.M. German, and P. Devanbu. Collaboration and governance with distributed version control. 2010.

[19] Nayan B. Ruparelia. The history of version control. SIGSOFT Softw. Eng. Notes, 35(1):5–9, 2010. ISSN 0163-5948. doi: http://doi.acm.org.ezproxy.ub.gu.se/10.1145/1668862.1668876.

[20] A. Sarma and A. van der Hoek. A conflict detected earlier is a conflict resolved easier. Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering, pages 82–86, 2004.

[21] M. Shaikh and T. Cornford. Version management tools: Cvs to bk in the linux kernel. In 3rd Workshop on Open Source Software Engineering, pages 127–131, 2002.

[22] P.M. Shields. Pragmatism as philosophy of science: A tool for public administration. Research in Public Administration, 4(1):195–225, 1998.

[23] L. Sproull and S. Kiesler. Connections: New ways of working in the networked organization. The MIT Press, 1992.

[24] Wikipedia. Content analysis — wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/w/index.php?title=Content_analysis&oldid=361259368. [Online; accessed 12-May-2010].

[25] Wikipedia. Distributed revision control — wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/w/index.php?title=Distributed_revision_control&oldid=361573339. [Online; accessed 27-May-2010].

[26] Wikipedia. Revision control — wikipedia, the free encyclopedia, 2010. URL http://en.wikipedia.org/w/index.php?title=Revision_control&oldid=363889540. [Online; accessed 27-May-2010].

[27] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida. Collaboration with lean media: how open-source software succeeds. In Proceedings of the 2000 ACM conference on Computer supported cooperative work, pages 329–338. ACM, 2000.

APPENDIX

Below follows the quotes which were collected from the different sites and the location as to where one can further read for each individual site.

Site 1[28] Pros to the centralized model:

1.1 "Subversion encourages a simple central repository model, discouraging large scale branching."

1.2 "Where the artifacts you're collaborating on are binary and cannot be merged by the VCS - for example word documents or presentation decks. In this case you need to revert to pessimistic locking with single-writer checkouts - and that requires a centralized system."

Cons to the centralized model and pros to the distributed model:

1.3 & 1.6 "Because distributed systems always give you a local disk copy of the whole repository, this means that repository operations are always fast as they don't involve network calls to central servers. This is a palpable difference if you are looking at logs, diffing to old revisions, and anything else that involves the full repository. If this is noticeable on my home network, it is a huge issue if your repository is on another continent - as we find with our distributed projects."

1.4 & 1.7 "If you travel away from your network connection to the repository, a distributed system will still allow you to work with the repository. You can commit checkpoints of your work, browse history, and compare revisions on an airplane without a network connection."

1.5 & 1.8 "DVCS encourages quick branching for experimentation. You can do branches in Subversion, but the fact that they are visible to all discourages people from opening up a branch for experimental work. Similarly a DVCS encourages check-pointing of work: committing incomplete changes, that may not even compile or pass tests, to your local repository. Again you could do this on a developer branch in Subversion, but the fact that such branches are in the shared space makes people less likely to do so."

Cons to the distributed model:

1.9 "This last point also leads to the argument against a DVCS, that it encourages wanton branching, that feels good early on but can easily lead you to merge doom. In particular the FeatureBranch approach is a popular one that I don't encourage. As with similar comments earlier I must point out that reckless branching isn't something that's particular to one tool. I've often heard people in ClearCase environments complain of the same issue. But DVCSs encourage branching, and that's the major reason why I indicate that team needs more skill to use a DVCS well."

1.10 "A distributed system opens up lots of flexibility in work-flow, but that flexibility can be dangerous if you don't have the maturity to use it well"

1.11 "And although DVCSs give you lots of flexibility in how you arrange your work-flows, most people I know still base their work patterns on the notion of a shared mainline repository that's used with Continuous Integration. Although modern VCS have almost magic tools to merge different people's changes, these merges are still just merging text. Continuous Integration is still necessary to get semantic consistency. So as a result even a team using DVCS usually still has the notion of the central master repository."

Site 2[29] Cons to the centralized model:

2.1 The whole trunk/tags/branches convention. "I've never used it, mainly because I haven't felt the need. Also, because it's hard. Repeated merges from a branch are painful"

2.2 No offline commits

Pros for distributed models:

2.3 "Commit to your local repository, and push out the changes when you have an Internet connection or when you're confident the feature is stable enough."

Site 3[30] Pros for the centralized model:

3.1 "With a centralized system, people are forced to collaborate and review each other's work"

Cons against the centralized model:

3.2 "With Subversion you can only commit if you have commit privileges."

3.3 "Without proper merging the ability to branch is next to useless; even Subversion can branch like a champ, it's just its merging which sucks."

Pros for the distributed model:

3.4 "Every "checkout" is actually a full copy of the entire remote repository (all its branches, all its history). After a while you just get used to the idea that you can rapidly look back at any previous tag (previous releases for example) and look at any of the branches that are currently under development (unlike Subversion where the typical workflow is to check out only the tip of the "trunk"). This also means that every checkout is a full backup of everything in the history of a project. And once you've done this initial

[28] http://java.dzone.com/articles/version-control-tools

[29] http://www.dribin.org/dave/blog/archives/2007/12/28/dvcs/

[30] http://www.wincent.com/a/about/wincent/weblog/archives/2007/10/why_distributed.php

"checkout" (called a "clone" in Git terminology for obvious reasons) you can do all this stuff (look at previous releases, switch branches, explore the history) without any network access."

3.5 "I already touched on this above, but the fact that your local "checkout" is a full-fledged repository means that you can do basically everything without a network connection: commit changes, create branches, perform diffs against any other point in the project history, merge, and so forth. You later make your changes available to the outside world when you are ready."

3.6 "Working offline is fast."

3.7 "Working offline provides you with an additional "staging area" (your local, private repository): if you commit something by mistake you can fix it up before anyone else sees it; this in turn means that you can make your history cleaner, keep the "noise" down, and can make your development easier to understand for others (and for yourself when you come back to look at it six months down the track)."

3.8 "With Git you can do anything "the maintainer" can do. You can develop your code in small steps, committing along the way, reverting changes if necessary, ensure its correctness, and then when it's ready for publication prepare a patch series that shows the logical steps you took; this will be much easier to understand than the monolithic "all-at-once" patch that you'd have to send if you were working with Subversion."

3.9 "Because Git is so good at branching and merging, it's dead easy to maintain separate "experimental" and "maintenance" branches. But even more so, branching and merging is so easy that you find yourself making feature branches."

Cons against the distributed model:

3.10 "DVCS ... encourages anti-social behavior ... In a nutshell: in a decentralized system, the default behavior is for each developer to privately fork the project. They have to put in some extra effort to share code and organize themselves into some sort of collaborative structure."

Site 4[31] Pros for the centralized model:

4.1 "Centralised VCS also results in a bias towards the server as the single point of truth..."

4.2 "...your local sandbox can get messed up via conflicts, but a centralised model doesn't ever allow you to check in conflicted files. If the local merge after update fails you have to cleanup conflicts manually."

Cons against the centralized model:

4.3 "This points to a limitation in centralised version control systems - the developer local history of changes is not preserved. It is as though you have a maintenance/dev branch where every time you commit to the branch, the checkin is routed to the code line where the branch was taken from. That means no branch history is kept, ever. The information is thrown away. And if your version of the file prior to the merge is never versioned, that in turn that means any post-facto work or cleanup of mistakes has to been dealt with manually. You can't go back through the history."

Pros for the distributed model:

4.4 "Better branching, thus control over code"

4.5 "Speed, thus ease of use ... Once you start using a DVCS for local work, going back to a centralised model feels slow, as in your mind wanders and breaks flow, which is the worst kind of slow."

Site 5[32] Cons against the centralized model:

5.1 "If you take an SVN checkout of, say, the Linux kernel source code, you're free to play with it, and make any changes you like, but you can't commit those changes."

Pros for the distributed model:

5.2 "there is no central repository for a project which is in Git. Another way to look at it is that every single checkout/working copy - known as a "clone" - of the project is actually the repository itself, with its full history and everything you need to manage it."

5.3 "Another unfamiliar aspect is that every working copy is effectively a branch, and commits to it are isolated from any kind of notional trunk, or indeed anybody else's clone, until such time as you manually merge two or more clones."

5.4 "If you travel a lot, or otherwise regularly find yourself in situations where you don't have easy, reliable internet access, you can continue to work and make commits without network access."

5.5 "Thus, a developer is not directly dependent on an external server, which greatly reduces the risk of slow response times and server outages preventing the developer from working and committing changes."

5.6 "While Git is designed primarily for collaborative development, dispensing with the requirement for a central repository makes Git an appealing option for individual developers or small teams who don't have the time or inclination to install and maintain a version control repository and server."

5.7 "A user doesn't require privileged access to the repository in order to keep a full history of their own changes."

[31] http://www.dehora.net/journal/2008/04/06/what-a-dvcs-gets-you-maybe/

[32] http://pointbeing.net/weblog/2009/09/git-for-subversion-users.html

Site 6[33] Pros for the centralized model:

6.2 “Centralized systems allow you to list the files and branches and whatnot. Subversion made an important improvement on CVS by making the branching and tagging very transparent, where it was somewhat invisible and mysterious in CVS. That makes a real and practical difference in the usability of branching. Distributed systems are a step back in this respect.”

Cons against the distributed model:

6.2 “But frankly the "server-less" systems they set up are usually much more complex in practice than a single well-maintained server. Now that Subversion has fixed many of its server problems (with fsfs among other things), server maintenance is really not a problem. And we share the work around; there are far fewer servers than developers, and that works fine.”

6.3 “But more practically, I think distributed systems enable private work in a way that is bad for the community. I think the private workflow so touted by distributed systems is a total non-feature, even an anti-feature. Open source development should happen in the open; that’s what people usually want to do, and that’s what we should encourage at every opportunity.”

[33] http://blog.ianbicking.org/distributed-vs-centralized-scm.html