Software Release Management

Thomas Lampoltshammer
Bernhard Zechmeister

University of Salzburg
Department of Computer Sciences

thomas_josef.lampoltshammer@stud.sbg.ac.at
bernhard.zechmeister@stud.sbg.ac.at

Abstract

Software release management is a field of management often underestimated regarding its complexity by software vendors. Being used properly it helps the vendor to work with a standardized framework which provides defined processes for software releases and their maintenance. However, this framework will not function properly if used on false assumptions taken by the vendor. This paper intents to give an introduction to the general topic of software release management. It will cover the best practice framework ITIL followed by a detailed discussion of misconceptions in the field software release management. In addition to that, two concrete tools, namely Subversion and Git are introduced presenting two different approaches of versioning in software release management. The paper finally concludes with a comparison of these tools.

Keywords: Software Release Management, Subversion, Git, ITIL

1 Introduction

Due to the ongoing technological development, software products cannot be considered as static constructs. They are rather exposed to a continuous development process that consists of programming, testing and releasing in a cyclic re-occurrence. This dynamic characteristic that is usually represented by a large number of software products leads to high effort that has to be invested in the maintenance of previously released versions. As a result of high maintenance and service costs software release management is a discipline that has recently gained in importance.

This chapter will deal with the term software release management and its concepts in general followed by a more practical point of view with the help of ITIL standards version 2 and version 3.

1.1 Concepts of Software Release Management

Software release management in general contains a variety of other fields that have to be covered in the lifetime of a software product. The source code has to be managed in a form that allows collaboration of team members and provides a historical documentation of all the changes made during development. Furthermore a issue tracking system has to be introduced to provide an overview and a classification of changes that have to be applied, are in progress or have been completed. Mechanisms have to be developed to follow the needed quality requirements and to schedule the releases in an adequate cycle.

1.1.1 Version Control

A Version Control System (VCS)[1] provides the functionality to record, which changes are made by which author and at what time. It also enables developers to restore older states to revert unwanted changes or access previous states of development. The VCS allows multiple developers to work on the same project. It either provides a centralized or decentralized contribution strategy. The difference between those models will be discussed as a main topic later in this document.

[1] `http://en.wikipedia.org/wiki/Revision_control`

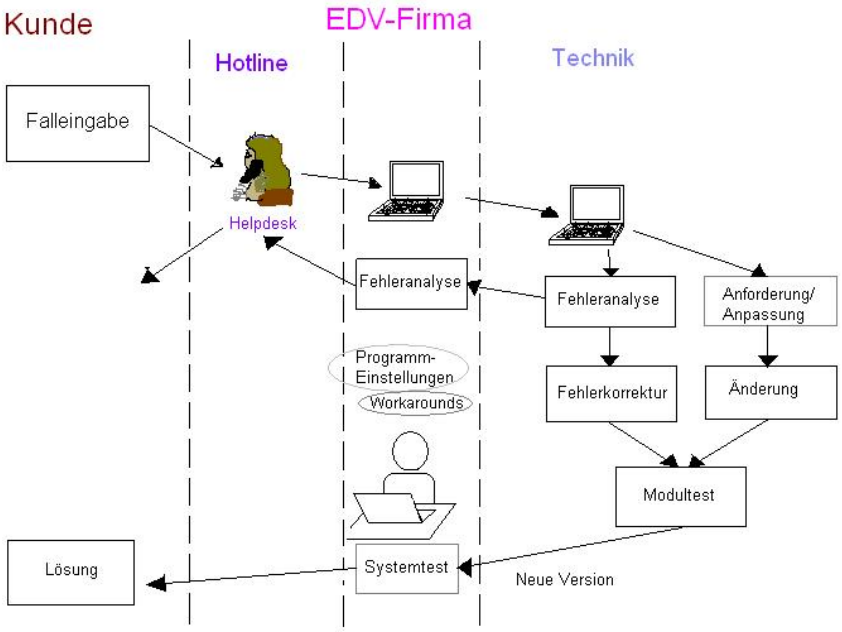

Figure 1: Issue processing in IT-Support [14]

1.1.2 Issue Tracking

An issue tracking system[2] provides a structured overview over bugs, features and requests and their processing state. Issues are initially created if a problem or request is identified the fist time. It is also used to manage responsibility and to monitor and guarantee a proper progress. Issue tracking systems can also become a knowledge base that helps finding solutions for re-occurring problems. Figure 1 shows the procession of an issue starting with a call at the hotline until the acceptance of the solution by the customer.

1.1.3 Quality Requirements

There are specific quality aspects that have to be concerned when managing a software release. It is important to evaluate the original targets that were planned for a specific release. The resulting software development state has to be compared to these targets to determine whether they have been reached. Specific tests plans provide strategies to support these evaluations [15].

[2] `http://en.wikipedia.org/wiki/Issue_tracking_system`

1.1.4 Release Process

The release process starts with the identification of the requirement of the release until the development is finished and the new version is published. Depending on the size and complexity of the software the process requires multiple other routines and phases to be passed during its cycle. It also covers the analysis of potential risks that can occur due to a planned release. The next section will provide more detailed information about software release management according to the IT Infrastructure Library (ITIL)[3].

1.2 Information Technology Infrastructure Library

The ITIL is an open, non-proprietary best-practice quasi-standard, which was originally started by the British government in the mid-80's. It is meant to support the Chief Information Offices (CIO) in strategic and operative decisions. Therefore, it provides a process model of all areas located in the IT service management. It is based on a pool of documents regarding subsectors of IT management. These documents include paramount examples of successful implementations of various organizations. The idea is to learn from others' failures and in consequence of that help to implement a high standard regarding IT service management.

The framework is not meant to be as a panacea or a tool as such. It will not solve any underlying problems regarding the lack of skills or free manpower inside an organization. It should also be noticed, that the examples inside the document pool are not intended to be copied and applied one to one. They should be seen as an inspiration and possible starting point to adapt the included concepts into the own organization.

ITIL can be classified into two basic main parts, as there are the *Service Delivery Set* and the *Service Support Set*. The first is providing strategic support for existing IT environments. Elements therein are service level management, finance management, availability management, capacity management and continuity management. The second main part is focusing on operational support as there are accident management, problem management, configuration management, change management and release manage-

[3] `http://www.itil-officialsite.com/home/home.aspx`

ment. The last one will be discussed in detail in the following part of this section [9, 3].

Release management in ITIL is responsible for handling the release of authorized hardware and software. Due to this paper's focus on software release management the authors will focus on the software component only. All versions of a release are created, authorized and managed in a central way, which enables the company or organization to provide structured and organized rollouts of software products. In this process not only the creation of new releases is included - the cooperation with distributors is treated as well. A tight coupling regarding the change management can be observed due to the fact that only authorized changes may be used as new releases. All release related information is stored in the Configuration Management Database (CMDB). This database is accessible for all processes related to the release. The history of all versions of a software release is stored in the Definitive Software Library (DSL). The whole structure of the release management can be seen in figure 2.

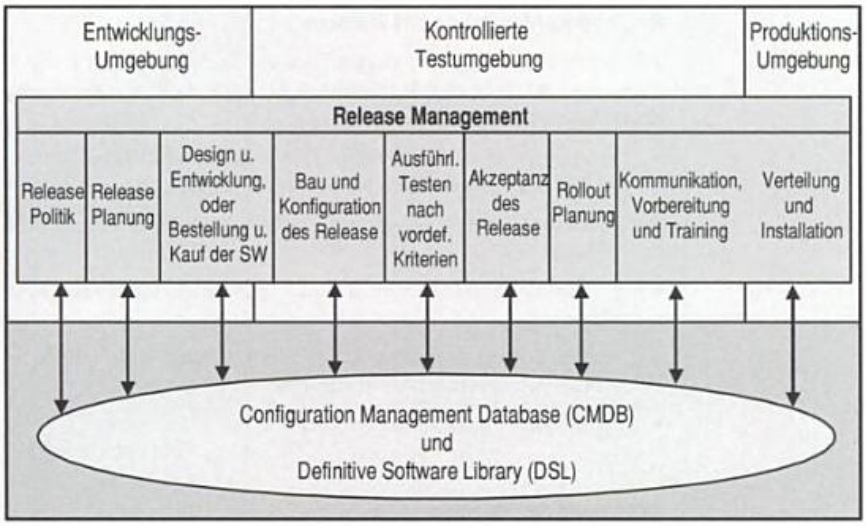

Figure 2: Adapted from [13]

As it is shown in figure 2, the tasks included in the release management process are not only applied to the productive environment, also the development and testing environment are taken into consideration. Therefore release management in ITIL has several key tasks to accomplish. One of them is the planned and controlled delivery of software. In order to do this in an

optimal way, changes to software components have to be planned, designed, configured, tested and bundled to release packages. In addition to that the coordination and evaluation of user acceptance tests play an important role regarding the successful delivery of releases.

If a release turns out not be be ready for delivery, there are two options available:

Fallback - this option represents the recovery of the former releases. This means, that all changes, which were accumulated in one release are going to be reverted.

Backout - this option is a step by step procedure. The single changes within a release are reverted one by one. As a result a backout defines a new release version every time a step has been revoked.

In some cases it is not possible to accomplish a recovery due to the fact, that the point-of-no-return was already reached. In that case emergency procedures have to be defined, so that the damage to the customer and to the vendor as well can be reduced to a minimum.

As it was demonstrated in this section, the release management included in the ITIL provides a solid process framework for handling software releases. However, this framework can only function at its best, if the vendor is aware of his customers needs. If this is not the case and the decisions are based on wrong estimations the best framework cannot save the vendor from grave problems.

Due to this fact, the next chapter is going to discuss known misconceptions of software vendors and the related misinterpretation of cost and value functions of vendor and customer alike.

2 Problems and Misconceptions

As already mentioned in the chapter before, the best framework cannot function properly if it is based on false assumptions. Such assumptions are not uncommon at all and happen even to large communities as it could be observed in [17]. Hence, the importance of being aware of possible traps related to software release management becomes even more evidently. For that reason this chapter will introduce functions of costs and value, both for vendor and customer and how these functions are important for the decision of releasing a new software version or purchasing a new release respectively.

2.1 Cost and Value Functions in Release Management

Weather a customer is going to update a product or not is based on a number of different factors. The most important factor is based on the assumption that the update brings a greater value to the customer than the existing product version is already providing. The value of the update can be expressed via the function $Cval$ (1). In addition to that, the costs regarding the update process have also to be taken into consideration. This costs are shown in the function $Ccost$ (2). Only if the benefits of the update exceed the costs for it, the customer is going to change to the new product, see equation (3).

$$Cval(update) = value(newFeatures) + value(removalOfWorkarounds) \quad (1)$$

$$Cost(update) = \begin{cases} \text{cost(downtime)} + \text{cost(training)} + \text{cost(updateEffort)} + \\ \text{cost(lowFunctionality)} + \text{cost(paymentToVendor)} \end{cases} \quad (2)$$

$$Cval(update) > Ccost(update) \quad (3)$$

However, decisions based on value functions are not only a point of decision for customers, they are also important to the vendor himself. These functions

are more complex than the customer's ones due to a higher number of factors that have to be considered.

$$Vvalue(newUpdatePackage) = \begin{cases} \text{newCustomers * priceNewRelease} + \\ \text{oldCustomers * priceOfUpdate} + \\ \text{costReduction(support)} \end{cases} \quad (4)$$

$$Vcost(newUpdatePackage) = \begin{cases} \text{cost(development)} + \\ \text{cost(updateCurrentCustomers)} + \\ \text{cost(deliveryToCustomers)} + \\ \text{cost(packageCreation)} + \\ \text{cost(increasedSupport)} + \\ \text{cost(marketing)} \end{cases} \quad (5)$$

$$Vval(update) > Vcost(update) \quad (6)$$

A vendor releasing a new update is taking financial benefits from two groups simultaneously. There are the new customers who will be addressed via this update as well as the existing customers, who are going to upgrade. The combined factors can be seen in $Vval$ (4). On the other hand there are a high amount of additional costs related to new updates. They have to be developed, promoted, distributed and supported as well. All these costs are combined in the function $Vcost$ (5). Only if $Vval$ is higher than $Vcost$ the vendor is going to release a new version of his product, see function (6) [5].

Having these factors in mind, the next section will discuss some misconceptions often assumed by vendors related to their release management.

2.2 Misconceptions in Release Management

Customers want to stay up-to-date - Vendors' customers have only one reason to buy their products. It should fulfill a certain requirement needed in their companies. If a new software version is released and adds no features desired by the customers they are not going to buy it. For example, a customer purchased a FTP software for exchanging files. There is no reason for him to buy the successor version of the software,

because it now also features a VPN client as well as long as he does not need such a client and the old release version works satisfactorily for him. In this case it can be seen, that *Ccost* outweighs *Cval*.

Customers must stay up-to-date - This misconception is based on the same issue as above mentioned. But here the point of time where the issue is detected plays an important role. In the first example, the issue occurs when a new software version is released. In the situation here the vendor of the software product is realizing the problem when already different versions are deployed in the market and all have to be supported. Due to the customers are still satisfied with the version they purchased years ago and did not apply any updates so far. This can lead to big problems if the vendor not only added new features but also security fixes which are not available without the new release and so only a few customers are now protected. Again, *Ccost* outweighs *Cval*.

Fixes can be postponed to the next major release - This is a typical and common mistake due to the plan of saving costs and time to implement the fix into several releases. If the next major release is coming soon and the customers want to update it can work out but if not the fixes have to be back ported to ensure satisfactory of the customers. This is a classic case where *Ccost* appeared to be less than *Cval*, thus it turned out to be vice versa.

Workarounds must be avoided at all costs - As long as the statement *Ccost* outweighs *Cval* is valid, workarounds are a acceptable solution in comparison to high costs for new software releases and their deployments. A prominent example can be found by looking on Microsoft's Internet Explorer which featured errors regarding the correct display of style sheets. However, it was prominent that this issue could be solved by applying a hack to the source code and hence Microsoft delayed the fix to the release of Internet Explorer 7. Thus, Microsoft could avoid the additional expenses regarding the deployment of a fix to a huge number of customers.

Customers always want new features - Here the company assumes that all customers always want new features if they come along freely. For example, a console-based application is going to be equipped with a

graphical user interface. Regarding the customers, which were used to the old system and could operate it nearly blindly, this new interface was disturbing and slowing them down. So for them *Ccost* outweighs *Cval* and the vendor had to redraw the new version.

A quiet customer is a happy customer - In an informal survey conducted by a software vendor it turned out, that it does not mean for the customer to be more satisfied with a product if he is not contacting support or helpdesk regularly, especially in the early adoption phase of the product. In many cases, those who complained and discussed with the vendor at the beginning were more satisfied at the ending. In some cases, the quiet customer had already turned to another product even if the contract with the original vendor was still valid. The vendor has missed his chance to intervene and convince the customer that despite additional calls, visits and so on the *Cval* of the product still succeeds its *Ccost*

Customers read release notes - If a company or organization is big enough to afford a system manager, who is eager in checking release notes to find valuable updates and fixes for the company this is a nice to have condition but not a standard. Normally, customers are flooded with release notes or are even getting none and have to look up the fixes themselves on the vendor's website. It can not be assumed, that the customer will take care of themselves. Therefore it would be a proactive solution to filter release updates and fixes according to the customer's products and just sending the filtered and for the customer useful list of fixes and updates [5].

This chapter discussed several common misconceptions of software vendors regarding their customers. It was shown that this has a falsifying effect regarding the interpretation of the associated cost and value functions of both vendor and customer. Combining the findings from this misconceptions and the solid ITIL framework it becomes possible for the vendor to plan, release and manage his software releases in a structured, organized and reliable way. The following chapter will now introduce two version control tools that can be utilized in order to support the vendor during the development phase of the software release.

3 Tools

This chapter explores possible tools that can help in the process of software release management. The main focus is set on the tools Subversion and Git, which represent two different approaches, a centralized and a distributed model of collaboration. Their origin is going to be discussed as well as their methods, advantages and disadvantages.

3.1 Subversion

The first tool that is examined is Subversion, a free open-source VCS that is developed as a project of the Apache Software Foundation. The Subversion project has the following vision [11]:

> *Subversion exists to be universally recognized and adopted as an open-source, centralized version control system characterized by its reliability as a safe haven for valuable data; the simplicity of its model and usage; and its ability to support the needs of a wide variety of users and projects, from individuals to large-scale enterprise operations.*

3.1.1 History of Subversion

The Subversion project was started in 2000 by CollabNet, Inc.[4] as a replacement for Concurrent Version System (CVS) which had obvious known limitations. Nevertheless CVS had been used in most open source projects because it was the best tool available under a free license at this time. Because of the frustration many developers encountered with CVS the Subversion project was growing very quickly and after 14 months of development in 2001 Subversion became *self hosting* [2].

In 2010 the Subversion project reached the desire to be integrated in the Apache Software foundation, where it was fully adopted as a top-level-project [2].

[4] `http://www.collab.net/`

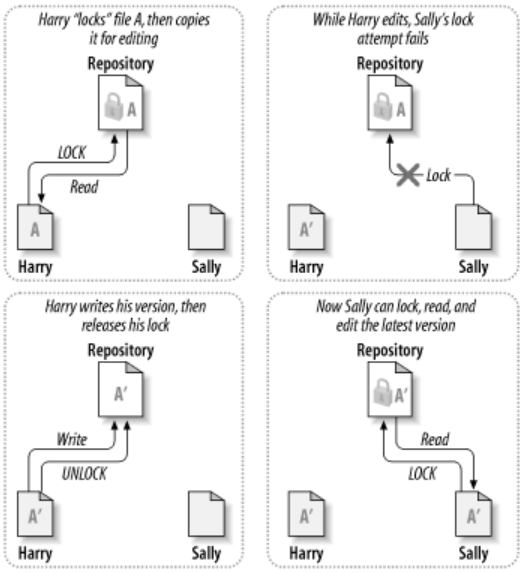

Figure 3: The lock-modify-unlock solution [2]

3.1.2 Versioning Models

Sharing files for collaborative work causes the typical problem where one person overwrites the changes of another person by copying a file to a central storage. One possible solution would be the *Lock-Modify-Unlock* strategy where simultaneous editing of the same file is mutually excluded. It is obvious that this solution is not satisfying the requirements of software development teams where parallel development and maintenance of multiple features takes place at the same time and also in the same files. Figure 3 shows the workflow in the lock-modify-unlock solution [2].

The *Copy-Modify-Merge* model allows users to work concurrently on projects and even files in their own offline working copies. The working copies represent a local copy of a specific revision of the central repository. The modifications of the single users are merged together by synchronizing them with the repository. The individual changes can be merged automatically as long as there are no conflicting modifications. If there are for example two different modifications in the same line of program source code this would result in a conflict. These conflicts have to be solved manually by the users. They have to be able to make the right choice for a modification or combining them to work properly. Usually conflicts do not occur very often in a well organized development environment and if they do they are easily and quickly

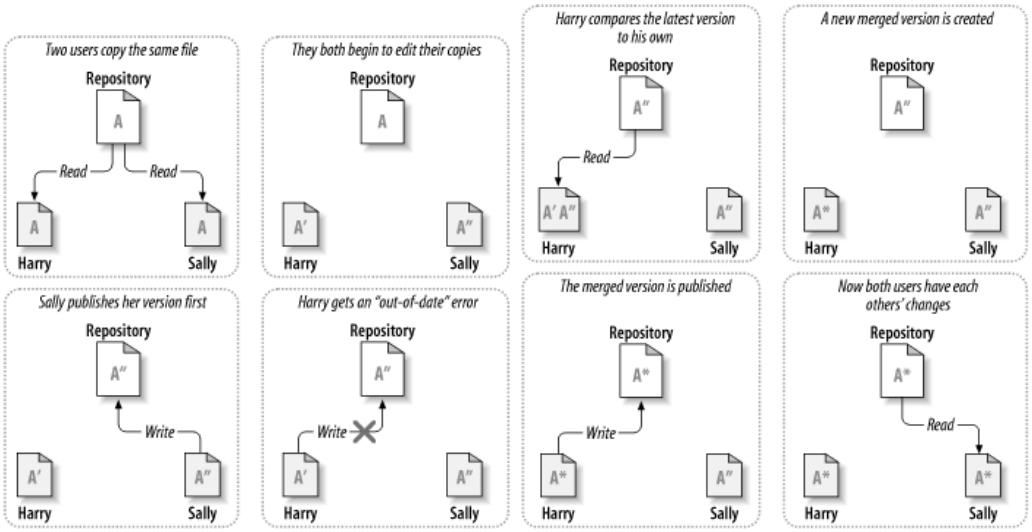

Figure 4: The copy-modify-merge solution [2]

solvable. On the other hand in some cases, especially when there is a lack in communication between team members, it could happen that resolving conflicts becomes a very difficult job. Figure 4 illustrates the collaboration in the copy-modify-merge model [2].

Every commit of changes to the central repository is handled as an atomic operation by subversion which increments the global revision number. Therefore each revision number lower than the actual head revision number represents a historical state of the whole repository and its whole content at the specific point of time in the past. Figure 5 visualizes the historical development of a repository starting at revision 0. Below each revision number the file system tree is shown in the state it had been after the commit with that specific revision number [2].

3.1.3 Cheap Copies

When files or directories are copied in Subversion, the existing data is not duplicated in the repository. The new file or directory refers to the original source it was copied from. This can be achieved by simply remembering the location and revision number of the original which points to a fixed state that will not change anymore. Starting at the revision where the copy operation takes place, the two copies will start their independent progress

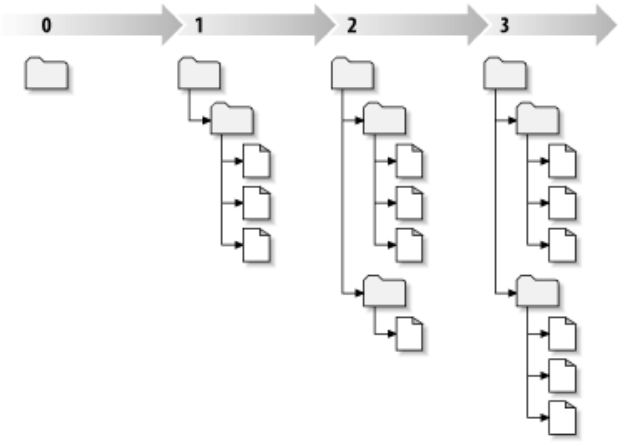

Figure 5: Global revisions in the repository [2]

and history development. Because the copy operation will not consume any memory (except the reference information) copies made in Subversion are called *cheap copies* [2].

The concept of copying is the base technique that is used for creating tags and branches in Subversion.

3.1.4 Tagging and Branching

The possibility of creating tags addresses the need of making snapshots of specific states in the lifetime of a project. For example a possible reason for creating a tag could be the release of a new version. The tag will just be a cheap copy of the project state at a specific revision, which is copied to the *tags* subdirectory of the project and given a meaningful name. Per definition commits should not be made to tags, however Subversion would allow it as they are just directories in the repository file system [2].

In software projects development does not only take place in one single line. As shown in Figure 6 in some cases it is necessary to make maintenance modifications to previously released versions which will result in patch or bug-fix releases. These requirements can be achieved by creating branches, which represent lines of development that are independent of the main line of development.

The main line is usually located in the *trunk* subdirectory of a project. A separate subdirectory *branches* will comprise all other lines that exist. There

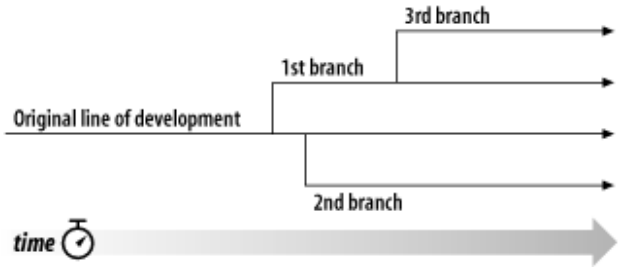

Figure 6: Branches in Subversion [2]

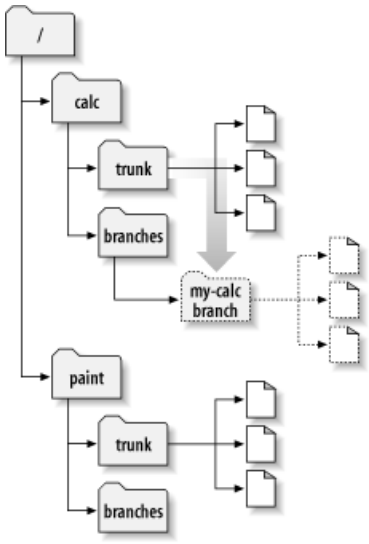

Figure 7: Creating a feature branch in Subversion [2]

can be different purposes branches are used for. One of them is creating release branches. As mentioned before these can be made to prepare releases and maintain previously released versions. Another possibility to use branches is to develop new features in dedicated feature branches independently and without disturbing the main development in the trunk. Figure 7 illustrates the creation of a branch *my-calc branch* whose origin is the *trunk* of the project *calc* [2, 12].

When working with branches it is often necessary to port changes across different branches. Therefore Subversion has the built-in merge command. Single changes can be merged from one branch (or the trunk) to another to obtain procedures such as reintegration of a feature branch or application of chosen bug-fixes to a release branch. Figure 8 shows the history of a Subver-

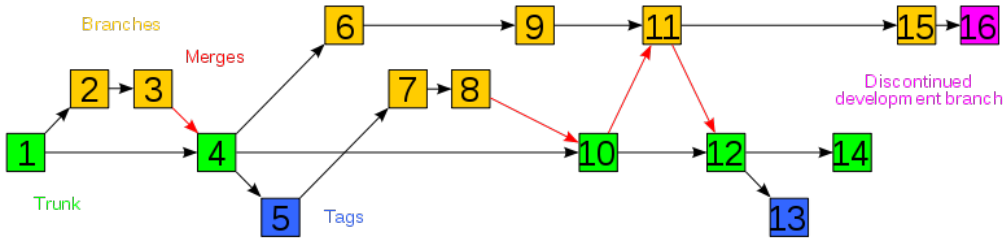

Figure 8: Revision graph of project history in Subversion [16]

sion project over a set of revisions with copy operations to create branches or tags and merge operation to combine different lines of development [2].

3.1.5 Managing Releases

The following section describes the release procedure that is made by Subversion itself. In [12] the guidelines for making Subversion releases are defined. It is widely based on the Apache Portable Runtime (APR) release guidelines. Both of them are open source projects whereas the guidelines are based on a community-driven development model.

As there are different purposes that releases of new versions are scheduled for, releases are classified in three different types. These types are also reflected in the release numbers which have the MAJOR.MINOR.PATCH format convention. Version numbers are increased by following these rules:

Patch release - Patch releases are designated for bug fixes and small changes without changes to the Application Programming Interface (API). No new features are introduced in patch releases. Only existing features are maintained. They are fully for- and backward compatible to the other releases in the same minor version line.

Minor release - In a minor release new features can be introduced and new API functionality may be added. Existing API functions must not be changed. Therefore code that was designed to work with a previous version in the same major version still has to work when upgrading to a new minor version. If new functionality is already in use, downgrading to an older minor release will not provide full compatibility anymore.

Major release - Major releases contain the biggest possible change sets. Only in major releases it is allowed to change or remove existing API functions.

Before a minor or major version is released it has to pass a stabilization process which starts with the creation of a release branch for the planned release as a copy of the trunk. For a period of four weeks only conservative bug fixes are made. If changes that have the potential to destabilize the release the soaking period is re-started. Core changes and changes where there is a disagreement if it could have destabilizing consequences require voting by committers. Changes need three positive votes and may not have any veto to be accepted for porting to the release branch.

3.1.6 Known Problems and Limitations

As mentioned before Subversion is a centralized VCS. In a central repository it is important to control the rights of the developers. A special permission is granted to the committers that gives them write access to the repository. It is obvious that committers have to have good qualifications and must have the community's trust. The disadvantage of this system is that not everyone is allowed to make commits to the repository. Potentially good developers who do not have write access to the repository cannot use all the functionalities that are offered by the VCS which makes it difficult for new members to join the development team [7].

Another aspect of the centralized version control system is that the central repository represents a *Single Point of Failure (SPOF)*. If the central server is not available the developers are not able to commit their changes, view the history or synchronize them amongst others. This situation will already occur when a developer has to work offline without having access to the repository [10].

Merging changes that are conflicting with other changes can lead to problems because there is only a single point of integration. Because solving of conflicts has to be done manually it might happen that developers have to wait for others to complete their outstanding merge operation by resolving all the conflicts manually. Because merge operations can become complicated very quickly, developers often abstain from creating branches, even if an isolated development line would be reasonable [10].

Distributed VCS address these problems by having solutions for them. In distributed VCS every developer has a full clone of the entire repository included in the working copy. This provides local access to all the history of the projects and allows all operations to be made offline such as branching, tagging or committing code modifications. Actually each working copy is already a branch. Therefore each synchronization between repositories is a commonly used merge operation that can be handled by the system and the users [10].

3.2 Git

The second tool, which is going to be examined, is the free, open source and distributed VCS called *Git*[5]. It is developed by Junio Hamano and Linus Torvalds [1]:

> *Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server.*

3.2.1 History of Git

Git was born out of necessity. Before the year 2002, the Linux community around Linus Torvalds used patches and archive files in order to store and distribute updates of the Linux kernel. In order to improve the version control, in 2002 the project moved to the proprietary, distributed VCS BitKeeper[6]. This step was not welcomed by all community members due to the fact that it was against the principles of free software development not to use closed source tools. In order to be able to access the repository, a special BitKeeper client had to be used. This client was free of charge for the kernel developers but was the only allowed way to access the repository. Andrew Tridgell, author of Samba[7], was not satisfied with this situation and therefore developed through reverse-engineering of the BitKeeper protocols his own client software. By doing so, he provoked the author of BitKeeper Larry McVoy,

[5] `http://git-scm.com/`
[6] `http://bitkeeper.com/`
[7] `http://www.samba.org/`

which in return revoked all free licenses of the Linux community. Due to the urgent need of a new VCS, Linus Torvalds decided to develop his own VCS from the scratch and *git* was born[8].

3.2.2 Fundamentals of Git

The heart of a Git system is its repository. It can be described as a database holding together all information needed in order to manage the revisions and history of a software project handled by Git. Therefore, the repository holds a complete copy of the whole project content as long as it remains alive. The special property in comparison to other VCS is the fact, that it also holds a complete copy of the repository itself. Git also includes a set of configuration settings related to each repository. Such settings include values like user's name, email address and so on. These configurations are not duplicated during a clone procedure due to the fact that Git manages these settings in a per-site, per-user and per-repository way.

Each repository can be split up into two main data structures as there are the *object store* and the *index*. These data is stored in the root of the working directory hidden under the identity of *.git*. The *object store* is included into each cloning process in order to provide capabilities of a fully distributed VCS, while the *index* is described as a transitory information and hence is private to a repository. If needed, the *index* can be created or modified on demand [8].

As it can be seen in figure 9 the *object store* consists in its whole out of four types of objects:

Blobs - Every single version of each file is embodied as a *blob*. The acronym stands for 'binary large object' which can be interpreted as variable or file that can include any kind of data. The data's structure is normally ignored by the program. Due to the fact, that a *blob* does not own any metadata about its name or the file included, it can be described as unintelligible.

Trees - *Trees* are used in order to illustrate one level of directory information. It is responsible for tracking *blob* identifiers, pathnames and

[8] `http://www.infoworld.com/t/platforms/linus-torvalds-bitkeeper-blunder-905`

some metadata related to all files in that particular directory. It is also capable of recursive references regarding subtree objects. Hence, it is possible to represent a complete hierarchy of files or subdirectories.

Commits - This object holds metadata related to any changes submitted to the repository. This includes information like author, committer, date of commit and logging information. Each commit object is referencing to a *tree* object that includes as a whole snapshot, the current state of the repository at the time the commit operation was performed.

Tags - This type of object assigns a human-readable description to an object.

The information stored inside the *object store* are over time subject to changes due to editing, commits and so on. In order to preserve disk space and network bandwidth, the objects are compressed into *pack files*, also included inside the *object store*.

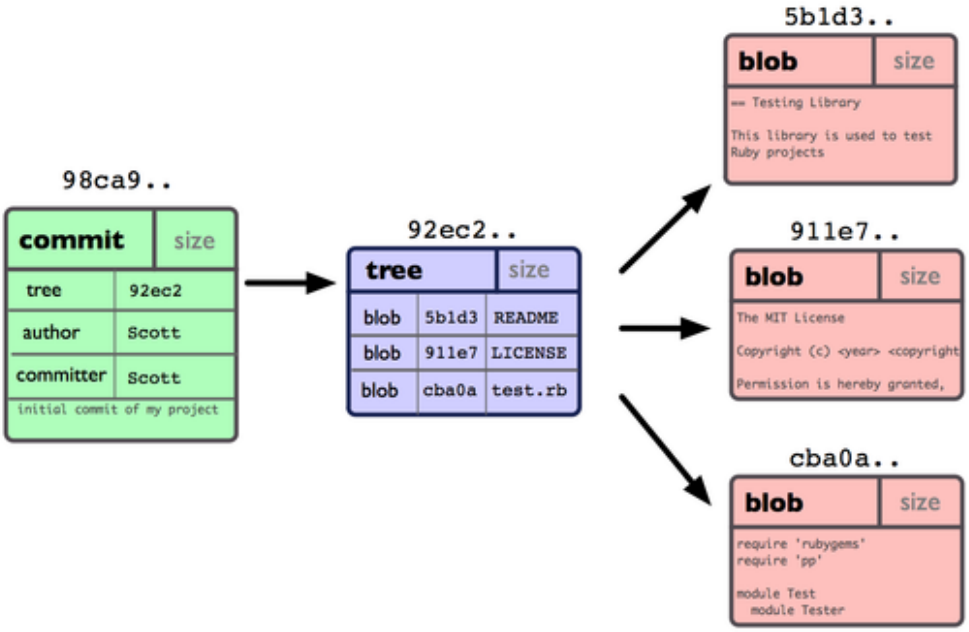

Figure 9: Git Objects [1]

The *index* is represented as a temporary and dynamic binary file, which depicts the whole repository structure at a certain moment in time. A key

feature of Git is based on the fact, that it is possible to alter contents inside the index in well-defined ways. The index also provides functionalities regarding incremental development phases and their commits. It is possible to stage changes within the index. These changes usually modify files in a certain way. The index keeps track of these modifications along the development process until the final commit. Therefore it is also possible to revert changes before a commit is down. So the index can be seen as guideline through the single steps in a repository from the untouched file to the final commit, see figure 10.

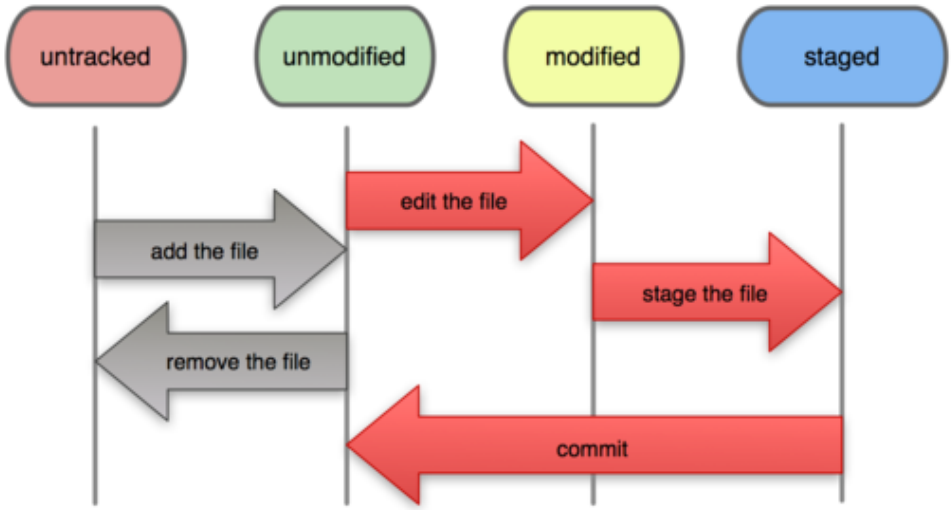

Figure 10: Git Structure [1]

3.2.3 Protocols used in Git

In order to perform network operations Git can have recourse to four different networking protocols. These protocols are *Local*, *Secure Shell (SSH)*, *Git* and *Hypertext Transfer Protocol (HTTP)*. All of the named protocols will be discussed in the upcoming section.

Local Protocol - This protocol is used if the remote repository is located in another directory on the same disk. This is normally the case if all members of a development team do have access to a shared file system like a Network File System (NFS) mount or maybe even work on the

same physical machine. If the whole development team is working in such a situation, setting up a repository can be accomplished without greater obstacles. The basic settings are equal to setting up a shared directory by setting the permissions for every member. However, if the group members are not at the same physical location, each of them has to remotely mount the shared directory, which in return introduces network latency.

SSH - This variation is likely to be the most used protocol for Git. If the repository is located on a server, this server is presumably also providing access via SSH. In addition to that, SSH is the only network-based protocol, which provides read and write access by standard. Compared to the both remaining protocols *Git* and HTTP this means a noticeable advantage due to them being read-only in general. Furthermore, SSH is secure due to its encryption and also provides access to authorized users only.

Git - This protocol is a special demon, which comes along with the Git installation. It provides similar capabilities as SSH but without any authentication. As a result a repository may be either accessible for everyone or no one. The Git protocol is the fastest protocol available regarding read-only access. The lack of write access on the other hand can be a downside and may only be compensated by providing additional access via SSH. Furthermore, the Git protocol uses port 9418, which is not a standard port and therefore requires additional manipulations on the server's firewall settings.

HTTP - This protocol provides pure simplicity regarding the access to the repository, which has just to be put into the http-directory of the specific web server where the project should be hosted on. Due the fact, that a HTTP server can handle thousand access requests at the same time, it will become difficult to overload such a server. Additional security options are also available via Hypertext Transfer Protocol Secure (HTTPS). The HTTP port is usually not blocked at corporate firewalls. The downside comes in terms of speed. The cloning process for example is slower than compared to the other protocol options.

3.2.4 Branching in Git

Branching in Git is handled by the help of moveable pointers to the designated start of the new branch. By default, the first branch (original development line) is titled *master*. This pointer moves along while new commits are submitted into the default branch - always pointing to the latest commit. Creating a new branch includes the creation of another pointer, which can be moved along the commits. This newly created pointer points to the current commit that is being worked with at the moment. In order to identify the current branch, Git uses another special pointer called *HEAD*.

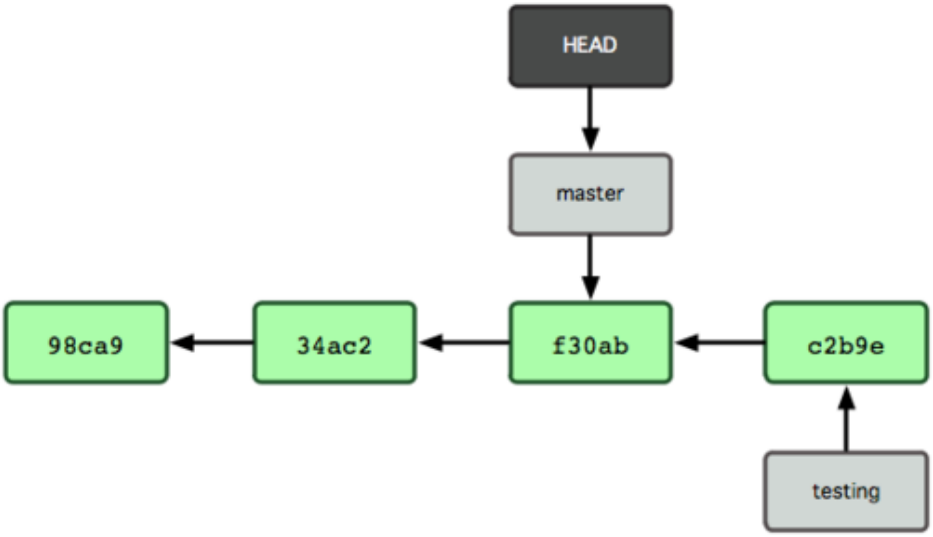

Figure 11: Branching in Git [1]

As it can be seen in figure 11, there exist two branches named *test* and *master*. The current active branch is *master*, shown by the *HEAD* pointer. The interesting thing to notice is, that by a commit to the active branch, the other branch pointer does not move along and still points to its latest commit. By switching between the branches, all files are reverted to the point in time, the branching pointer and therefore the commit points to.

Figure 12 shows how the development line has been split up by commits in both branches *test* and *master*. This makes it possible to develop from one common point in time into two different directions. Both branches are completely separated from each other. Git makes high usage out of branches

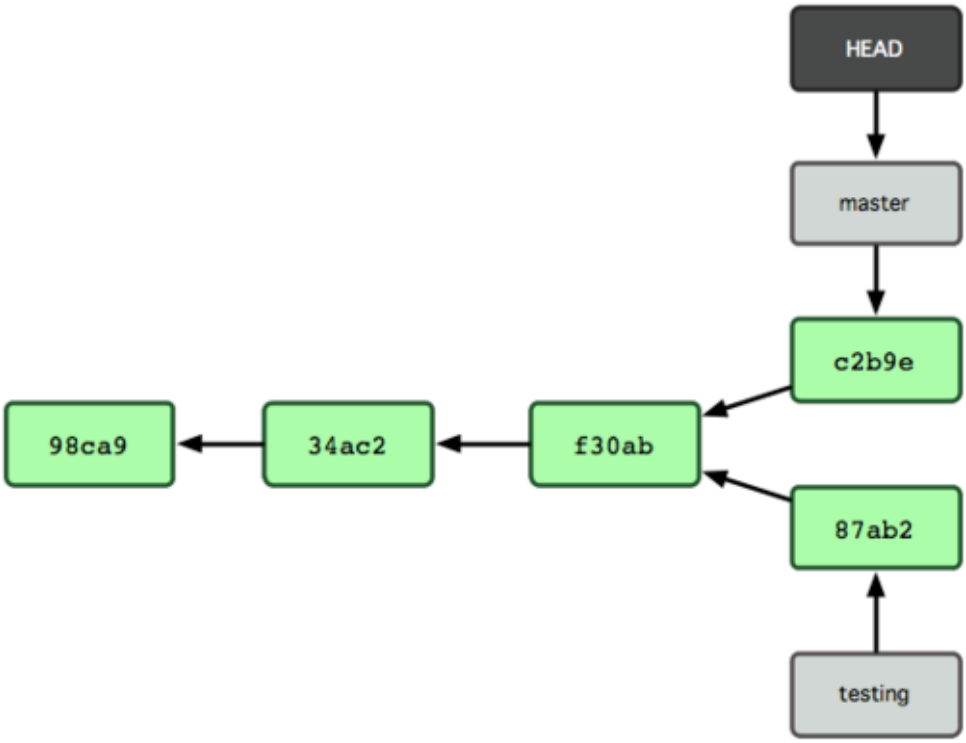

Figure 12: Branching in Git [1]

due to the fact that they are cheap in terms of storage. A branch in Git is represented by a file containing a 40 character SHA-1 checksum of the commit it refers to, which makes it easy to create and delete branches at any time.

The next section will describe how two branches can be merged into one new branch and development line.

3.2.5 Merging in Git

If a split of the original development line, as showed in the section before, should be combined into one single branch again, Git provides so-called merging functionalities. An example of such a merging procedure can be seen in figure 13.

In this case, the commit that is currently worked on is not an direct ancestor

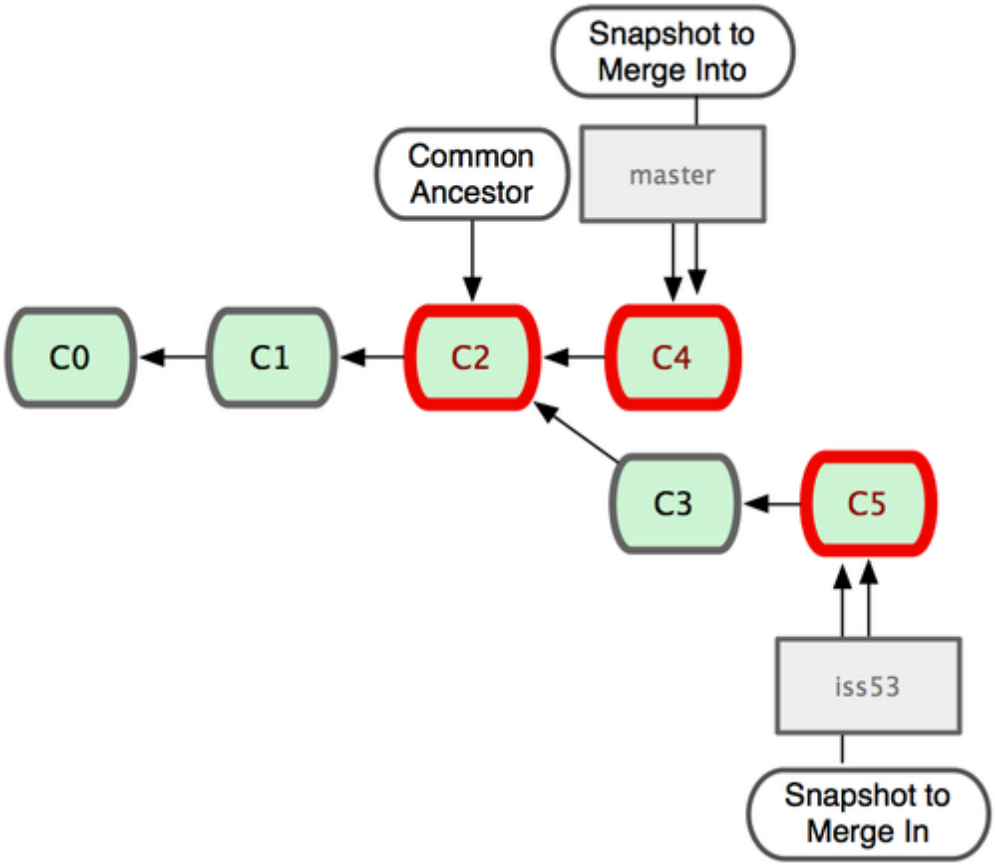

Figure 13: Merging in Git [1]

to the original branch and therefore this problem can not be resolved in one single step. In order to overcome this issues, a three-step merge is conducted by the usage of the two snapshots and a common ancestor. The best fitting common ancestor is determined by Git automatically. The branch pointer is not just forwarded, hence a new snapshot is created with a commit referring to it. This habit is called *merge commit* and is a special operation due to the fact that it depends on more than one ancestor as it is shown in figure 14.

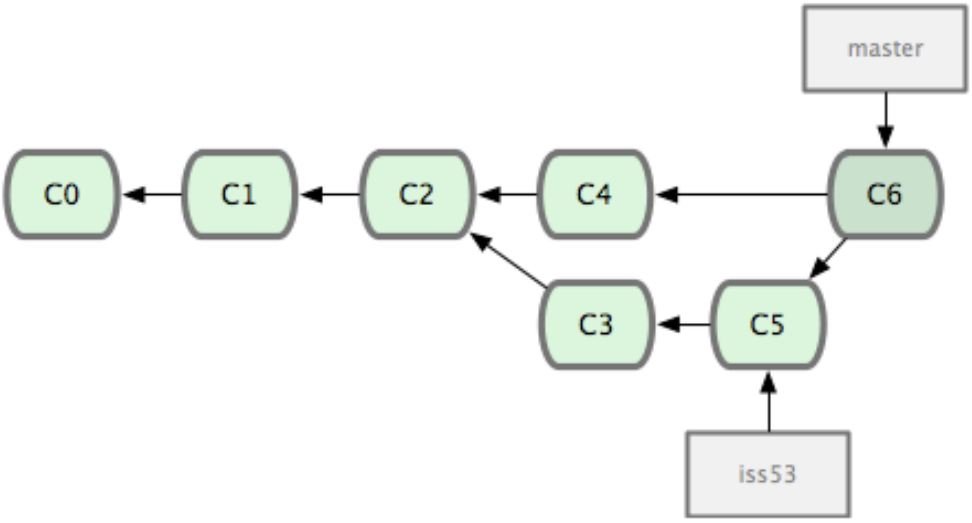

Figure 14: Merging in Git [1]

3.3 Comparison

This section provides a comparison of the advantages and disadvantages of the above described centralized (Subversion) and distributed (Git) VCS [4].

One of the differences between both systems that has a major impact is that considering Subversion, the central repository is storing all history related information, while the clients do not. Compared to Git this is different due to the clients also hold the complete history. This is representing a great benefit regarding data security in terms of loss of data. However, it also introduces issues concerning data privacy due to the fact that it is harder to secure all clients than just one central repository.

Another disparity between Git and Subversion is the compatibility regarding the operating system they can be used on. While Subversion plays along fine regarding all major operating systems, Git has still serious issues if installed on a Windows machine. This is a minus considering the fact, that today Windows holds about 90% market share[9].

In addition to that, Git still lacks in providing a decent graphical user interface. While Subversion offers beside its classic console a huge variety of

[9]http://www.appletell.com/apple/comment/operating-system-and-browser-market-shares-for-february-2011/

graphical user interfaces, Git still runs best at the console, which is one reason why it is hardly adapted by some user groups like designers.

However, if it comes to terms of speed of performance and space that is occupied by the system then Git is the option to choose. Due to the circumstances of the entire repository being available locally the following operations can be conducted without suffering any network latency:

- Viewing file history
- Performing a diff operation
- Obtaining another version of a file
- The merging of branches
- The commit of changes

Hence, the only operations that need a working network connection are related to *push* and *fetch* operations.

Regarding the used space Git outperforms Subversion as well. This becomes obvious on the example of the whole Mozilla project repository, which requires in Subversion disk space over 12GB while at the same time a repository in Git only uses 420MB[10].

All in together it can be stated, that none of the both tools is really the best one. Depending on the kind of project that has to be dealt with either the one or the other tool will fit best. However, in the authors' opinion Git is developing fast and while time passing by it will become a serious alternative to Subversion. Its speed and strength in terms of performance and being highly distributive are strong advantages. The only thing leaving a negative mark is due to the lack of a fully comfortable graphical user interface, though this will merely be a matter of time [6].

[10]http://news.softpedia.com/news/Git-Gaines-Advantage-Over-Subversion-136737.shtml

4 Summary

In this paper we discussed the importance of an existing and correct handled software release management framework inside a company or organization. The process of software release management can be complex and therefore the usage of supporting best practice frameworks like the ITIL is obligatory. Furthermore, software vendors have to be aware of common problems and misconceptions regarding the release of new software products. We have described such common misconceptions and showed with the help of cost and value function why these issues occur. However, not only theoretical methodologies are important - tools like version control systems have a huge impact factor as well. We have introduced the centralized version control system Subversion as well as its distributed counterpart Git. Both tools have been described regarding their features and were compared by their capabilities and disadvantages. As a result it can be stated, that only a combination of both ways, the organizational as well as the technical support of the software release management process will lead to a success. What exact tool fits best, still depends on the development structure inside the company as well as on the project itself.

References

[1] Chacon, S.: *Pro Git (Expert's Voice in Software Development)*. Apress, New York, 2009.

[2] Collins-Sussman, B., Fitzpatrick, B.W., and Pilato, C.M.: *Version Control with Subversion*. Creative Commons, Stanford, 2008.

[3] Elsässer, W.: *ITIL einführen und umsetzen: Leitfaden für effizientes IT-Management durch Prozessorientierung*. Hanser Fachbuchverlag, München, 2nd edition, 2006.

[4] Git Wiki Homepage: *Git SVN Comparision*. `https://git.wiki.kernel.org/index.php/GitSvnComparison` (22.03.2011).

[5] Jansen, S. and Brinkkemper, S.: *Ten misconceptions about product software release management explained using update cost/value functions*. In *Software Product Management, 2006. IWSPM '06. International Workshop on*, pp. 44 –50, Sep. 2006.

[6] Kunene, G.: *Subversion vs. Git: Choosing the Right Open Source Version Control System*. `http://www.developer.com/open/article.php/3902371/Subversion-vs-Git-Choosing-the-Right-Open-Source-\Version-Control-System.htm` (26.05.2011).

[7] Linus Torvalds: *Git - Source code contol the way it meant to be!* `http://www.youtube.com/watch?v=4XpnKHJAok8` (01.05.2011).

[8] Loeliger, J.: *Version control with Git*. O'Reilly, Sebastopol/CA, 2009.

[9] Olbrich, A.: *ITIL kompakt und verständlich: Effizientes IT Service Management - Den Standard für IT-Prozesse kennenlernen, verstehen und erfolgreich in der Praxis umsetzen*. Vieweg+Teubner, Wiesbaden, 4th edition, 2008.

[10] Roden, G.: *Verteilte Versionsverwaltungen*. dotnetpro, 6 2011.

[11] The Apache Software Foundation: *Apache Subversion*. `http://subversion.apache.org/` (28.04.2011).

[12] The Apache Software Foundation: *Making Subversion Releases*. http://subversion.apache.org/docs/community-guide/releasing.html (22.03.2011).

[13] Victor, F. and Günther, H.: *Optimiertes IT-Management mit ITIL: so steigern Sie die Leistung Ihrer IT-Organisation : Einführung, Vorgehen, Beispiele : mit 54 Abbildungen*. Vieweg+Teubner, Wiesbaden, 2005.

[14] Wikipedia, Die freie Enzyklopaedie: *Issue Tracking System*. http://de.wikipedia.org/wiki/Issue-Tracking-System (23.05.2011).

[15] Wikipedia, Die freie Enzyklopaedie: *Release Management*. http://de.wikipedia.org/wiki/Release_Management (23.05.2011).

[16] Wikipedia, the free encyclopedia: *Apache Subversion*. http://en.wikipedia.org/wiki/Apache_Subversion (11.05.2011).

[17] Wright, H. and Perry, D.: *Subversion 1.5: A case study in open source release mismanagement*. In *Emerging Trends in Free/Libre/Open Source Software Research and Development, 2009. FLOSS '09. ICSE Workshop on*, pp. 13 –18, May 2009.

Abbreviations

API Application Programming Interface

APR Apache Portable Runtime

CIO Chief Information Offices

CMDB Configuration Management Database

CVS Concurrent Version System

DSL Definitive Software Library

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

ITIL IT Infrastructure Library

NFS Network File System

SPOF Single Point of Failure

SSH Secure Shell

VCS Version Control System