Fachhochschul-Bachelorstudiengang
SOFTWARE ENGINEERING
A-4232 Hagenberg, Austria

Analysis and Comparison of Distributed Version Control Systems

Bachelorarbeit
Teil 1

zur Erlangung des akademischen Grades
Bachelor of Science in Engineering

Eingereicht von

Daniel Knittl-Frank

Begutachter: DI Dr. Stefan Wagner

Hagenberg, Juni 2010

software engineering
IKM Fakultät Hagenberg

Contents

Abstract

Version and configuration control systems have played an essential role in software development for the last decades. Starting with the new millennium a new trend surfaced, distributed version control: dedicated central servers for few authorized users lose importance in favor of a more dynamic workflow, in which every developer works with a copy of a project's complete history and changes are synchronized ad-hoc between developers.

Popular open-source distributed systems will be analyzed and compared – benchmarks provide a quick glance at performance of the examined systems. Furthermore differences to traditional, centralized systems, such as Subversion, will be discussed.

Kurzfassung

Bereits seit mehreren Jahrzehnten wird die Softwareentwicklung durch Versions- und Konfigurationsverwaltungsprogramme unterstützt. Seit der Jahrtausendwende zeichnet sich verteilte Versionsverwaltung als ein neuer Trend ab. Jeder Entwickler hat eine vollständige Kopie des Repositorys und Änderungen an diesen werden mit anderen synchronisiert.

Im Zuge dieser Arbeit werden die gängigsten verteilten Systeme miteinander verglichen und analysiert. Benchmarks in diversen Kategorien sollen zeigen, wo sich welches System bevorzugt einsetzen lässt. Weiters werden die Unterschiede zu traditionellen, zentralisierten Systemen wie Subversion untersucht.

Chapter 1

Introduction

1.1 Motivation

When developing software it is essential to be able to keep track of changes made to files and store different versions of a project as it evolves over time. Version control systems provide developers with many features to ease project development. Each version stores the author who made the change, creation time and can be provided with a message describing the change.

While RCS[1] could be perfectly used by system administrators to version control configuration files of an operating system, it lacked proper support for network operation. CVS[2] in its beginnings wrapped RCS commands in scripts which added networking support and enabled multiple users to work together. Subversion extended CVS to use transactions while committing and moved from per-file version numbers to project-global numbering.

Distributed version control systems (DVCS) are the next logical step in the evolution of version control systems. DVCS shifted from a server-client architecture to a peer-to-peer like system, making it very easy for developers to work on complex features without distracting others and to fork existing projects, while still retaining easy synchronization and merging between copies.

Not having to be connected to a single dedicated central server all the time cuts down network latency and enables developers to perform almost every operation locally and to work with the repository anytime and anywhere. Network access is only needed when synchronizing with other repository copies.

[1] Revision Control System

[2] Concurrent Versions System

1.2 Goals

The goal of this thesis is to provide both, an overview of distributed version control systems and an in-depth analysis of three open source projects. Implementation differences and their effects on runtime and disk usage are examined and highlighted.

Possible parallels and differences with centralized version control systems are pointed out and their implications are explained.

The reader will be introduced to different workflow models which are necessary when working with a distributed version control system.

Benchmarks of the latest available versions will support the performance conclusion drawn from the analysis of each system.

1.3 Content

In the next chapter the author will introduce the reader to version control in general, explain important terms used across different version control systems and give an overview of different workflows and branching models. Different patch formats are presented and merge strategies are discussed.

Several older and popular centralized systems, including CVS and Subversion, are highlighted in chapter 3 (Centralized Version Control) to give a historic background and to show existing solutions which are still widely used in corporate environments or open source projects.

Chapter 4 (Distributed Version Control) is dedicated to distributed version control systems. Differences to the central server-client architecture are highlighted and new workflows are presented. Three open-source systems are analysed in detail and their internal data structures, key features and user interface are described.

After these theoretical chapters the author will benchmark distributed systems in chapter 5 (Comparison) to show advantages and disadvantages in performance for common, daily used operations. Scalability of commands is important, the benchmarks will measure runtime depending on a number of factors and their effects on the result.

Chapter 2

Version control in general

According to [Vesperman, 2003, Section 1.1 What Is a Versioning System] version control, also known as *revision control*, “is the process of recording and retrieving changes in a project”, where as [Pilato et al., 2004] describe version control as a time machine which enables users to recover old versions through recorded changes made to files and directories.

[Vesperman, 2003, Section 1.1 What Is a Versioning System] lists the benefits and key features of a version control system:

- Any stored revision of a file can be retrieved to be viewed or changed.
- The differences between any two revisions can be displayed.
- Patches can be created automatically.
- Multiple developers can work simultaneously on the same project or file without loss of data.
- The project can be branched to allow simultaneous development along varied tracks. These branches can be merged back into the main line of development.
- Distributed development is supported across large or small networks.

The user of such a system stores his files in a central place, the repository. In the simplest form the repository stores every single version of the file, very much like saving the file under a different name each time with a time stamp or version number as part of the filename. This central place can be on a server, on the same computer, or even in the same directory as the file being edited.

A user has to *checkout* a file to start working on a local copy of it. After a user has edited a file, he informs the version control system about it. This operation is called a *checkin* or *commit*. The version control system copies a snapshot of the current file and adds it as a new version to its repository.

Usually multiple users want to work on the same project or same file simultaneously, so version control systems must provide a mechanism to prevent users from accidentally overwriting changes or blocking others.

2.1 Workflows

Two common methods how version control systems handle changes by different users on the same file are described in [Pilato et al., 2004, The Problem of File Sharing], namely*Lock-Modify-Unlock* and *Copy-Modify-Merge*.

- **Lock-Modify-Unlock:** Whenever a person wants to edit a file, the file has to be locked beforehand. After the changes are applied and the updated file is checked into the repository again, the lock on the file is released.

  In practice this model poses problems on the project's workflow. Work on the same file by different users is serialized and it is only possible for a single user to work on a file, however changes often occur in multiple files at a time.

  If a user locks a file and forgets to unlock it, other users are prevented from locking and changing that file. Tools often require an administrator to break the lock – which in turn causes problems for the user who locked the file in the first place.

  A *Lock-Modify-Unlock* workflow is desired for files which are hard or impossible to merge, such as binary files. It is not available in distributed version control systems, because there is no single global repository which could manage locks of files.

- **Copy-Modify-Merge:** Most of the time users want to modify the same set of files, but different parts in each file. With the *Copy-Modify-Merge* workflow every user has a personal copy of the project files, called a *working copy*. When a user tries to upload a file which is out-of-date (i. e. someone else modified the same file and uploaded it to the server) the user is forced to *merge* his changes with the other changes.

  In practice these merges can be done automatically by the version control system itself, but if the changes overlap, user interaction is required. *Conflicts* are marked in the file in question and the user has to decide, if he wants to keep his changes, keep the modified version from the server or rewrite the changes altogether.

  Version control systems can apply these changes using a three-way-merge, using the initially checked out version (base/original) as merge base, while repository changes (upstream) and local modifications are the left and right side of the merge.

Distributed version control systems use a model which is best described as '*`Copy-Modify-Commit-Merge`*'; users commit locally and then merge these commits with upstream.

2.2 Foundations

This section describes the foundations shared between different version control systems and explains terms commonly used. The terms are used by centralized version control systems as well as in distributed version control.

2.2.1 Glossary

- **Commit:** A commit, revision, version or changeset means a defined state of a project or file. Many systems use their own proper terms. Commit is also often used as a verb.
- **Checkout:** To load a particular version of a project or file to your hard disk, it needs to be checked out. In Subversion this term is also used to describe the working directory.
- **Repository:** The database where the history of a project is stored is called repository.
- **Branch:** A branch is a diverging parallel line of development. Many systems use a special name to denote the default branch in which development happens.
- **Clone:** To obtain a copy of a repository, it is cloned (only used in context of distributed version control systems). A clone is an implicit branch.
- **Merge:** To bring two branches back together, they need to be merged. Merging incorporates all changes in one branch into the other branch. This often creates a new commit (the merge commit).
- **Working directory:** The working directory – also working tree, working copy, sandbox, checkout – is the place where actual development happens.
- **Parent:** The commit on which a new commit is based. Merge commits have multiple parents.
- **Diff:** The difference between to files or versions of a project.
- **Patch:** A file describing the difference between files.

2.2.2 Repository

The *repository* or *database* stores the complete history of a project. In centralized version control the repository is hosted on a server and every developer interacts with the same repository. In distributed version control every developer keeps a separate copy of the repository.

Storage models

There are different ways to store versions of a file, which will be explained in this section.

- **Snapshot based:** In snapshot based storage each version of a file is stored as a complete snapshot, independent of older versions of it. To save space files are usually compressed using simple run length encoding. Access time for a single version of a file is asymptotically constant, in other words does not depend on history length, number of branches, repository size, etc.

  Git and Bazaar use a storage format based on the snapshot model.

- **Delta based:** Delta based storage only stores the difference between two versions of a file. Based on the idea that files do not change significantly between two subsequent versions, this cuts down the space needed for storage. The time to construct a version of a file depends on the number of changes to that file.

  Forward deltas store changes that must be applied to an earlier version to get the next version. *Reverse deltas* store a full copy of the most recent version and build older versions by applying changes in reverse order. This often improves performance, because the last version is accessed more often than older versions.

  Skip deltas calculate the difference to other versions than the previous, to either produce smaller deltas between files or to minimize the number of changes needed to reconstruct a file. Skip deltas in Subversion provide access times to any version of a file, which grow logarithmically with the number of revisions of that file.

  Mercurial, CVS and Subversion store history as deltas.

- **Weaves:** All versions of a file are stored interleaved in a single file, in which metadata is attached to each block (usually lines) describing the revisions which this block is part of. Any version can therefore be extracted with one sequential read over the whole storage file. Extracting a version of a file gets slower as the number of unique lines in the history of the file grows. Weave storage requires rewriting the storage file every time a new version is added.

Older versions of Bazaar used a weave based storage format.

2.2.3 Working Directory

The *working directory* or *working copy*, which is distinct to each developer, is the private copy of the files from the repository. Additional metadata is often stored in the working directory, e. g. to track the latest checked out version of files or to prepare commits. Most distributed version control systems store the developer's copy of the repository inside the working directory.

2.2.4 Diff and Patch

In software development it is essential to tell the differences between two versions of files. The differences are expressed in the *diff* format, which shows all lines which have changed from one version to another.

The program `diff` reports differences between two files, expressed as a minimal list of line changes to bring one file into agreement with the other [Hunt and McIlroy, 1976]. A group of contiguous differing lines is called a *hunk*[1].

The difference (and representation in *diff* format) between two files is computed by the `diff` algorithm. The input to diff are two sequences (usually lines), from which the *longest common subsequence* is determined, this sequence exists in both sequences. Combined with the first sequence it can be used to generate a sequence of additions and deletions to convert the first sequence into the second.

Files generated with `diff` can be applied with the command `patch`.

Contextual Diff

Diffs in context format start with a header, specifying the filenames which were used to generate this diff. Hunks in context format diffs are delimited by lines of asterisks.

Hunk headers start with asterisks or dashes and show the position and length of the hunk in both, the original and new file. New lines are prefixed by "`+`", removed lines by "`-`" and changed lines are prefixed by "`!`". Context lines are prefixed by a space. Another space character is inserted between this prefix and the actual line content.

Context lines in the output allow the application of the patch, even if line numbers changed slightly by looking for matching context lines[2]. This process is often called fuzzing.

An example patch file in contextual diff format can be seen in Listing 2.1.

[1] http://www.gnu.org/software/diffutils/manual/#Hunks

[2] http://www.gnu.org/software/diffutils/manual/#Imperfect

Listing 2.1: Patch file in contextual diff format

```
--- file.old  2010-01-03 13:52:54.000000000 +0100
+++ file.new  2010-01-03 13:53:18.000000000 +0100
***************
*** 1,3 ****
--- 1,4 ----
+ a
  1
  2
  3
***************
*** 5,12 ****
  5
  6
  7
! 8
! 9
  10
  11
  12
--- 6,14 ----
  5
  6
  7
! b
! c
! d
  10
  11
  12
***************
*** 14,17 ****
  14
  15
  16
- 17
--- 16,18 ----
```

Unified Diff

The unified format is similar to the context format, but generates usually less output for the same set of differences. Context format will print context lines for both files when lines are changed, where as unified diff merges the new and original hunks into a single hunk of added and removed lines, so context lines only need to be printed once.

Hunk headers in unified diff format are surrounded by two at signs (`@@`). Many diff implementations proceed this header with the name of the section the change occurred in (e. g. the function name in a C source file), although POSIX[3] does not define anything to come after the header line. Such section

[3] Portable Operating System Interface

comments are especially useful when a programmer wants to have a quick glance at what changed in the file.

Removed lines start with “-”, added lines start with “+”, changed lines are represented by a pair of add-remove lines. Unchanged context lines are prefixed with a single space character. This format or its variations are used in almost every major version control system (Subversion, Bazaar, Mercurial, Git).

The same difference shown in Listing 2.1 expressed as a unified diff can be seen in Listing 2.2 below:

Listing 2.2: Patch file in unified diff format

```
--- file.old  2010-01-03 13:52:54.000000000 +0100
+++ file.new  2010-01-03 13:53:18.000000000 +0100
@@ -1,3 +1,4 @@
+a
 1
 2
 3
@@ -5,8 +6,9 @@
 5
 6
 7
-8
-9
+b
+c
+d
 10
 11
 12
@@ -14,4 +16,3 @@
 14
 15
 16
-17
```

2.2.5 Merge

Merging is the process of incorporating changes from different sources into the same file or project. By calculating the difference between two files and their *common ancestor* (commit where development diverged) it is possible to apply all changes to this ancestor, resulting in a file containing both versions. This is called a *three-way merge*.

A three-way merge is described by the following mathematical statements [Baudiš, 2009, 6.2 Three-way Merge]:

$$\begin{aligned} d_x &= \Delta(b, x) \\ d_y &= \Delta(b, y) \\ d &= d_x \cup d_y \\ m &= \Delta^{-1}(b, d) \end{aligned}$$

To begin with, the difference (Δ) between the common ancestor (base, b) and both files is calculated, which is then combined in the second step. The resulting difference (d) is then applied to the base, resulting in the new version m with the changes from both files.

Merging in version control systems does not only work on file level, but also on a project level: entire directory trees can be merged, but not all systems support correct merging of renamed files. A special case is a *fast-forward merge*, which can be used when one of the two differences is empty. In such a case it is then unnecessary to create a new merged version, because the would be equal to the version with changes.

Several other merge strategies exist, which primarily differ in the way they select the common ancestor in case there are multiple candidates. This usually happens after criss-cross merges, where the two revisions are merged several times and conflicts are resolved differently each time. Git provides a recursive merge algorithm which can perform a three-way merge between the candidates recursively, to prevent mis-merges.

Bazaar uses the LCA[4] merge which builds sets (new or killed) of lines from all ancestors. If a line is contained in both sets a conflict is reported. Recursive merge and LCA merge behave like a normal three-way merge, when there is only a single common ancestor for a file.

A merge without conflicts can still result in broken code. A common problem scenario which cannot be resolved automatically by version control systems is renaming of variables or functions.

Listing 2.3: A semantic merge conflict

```
/* original version */
int a = 5;
a = doSomething(a);

anotherMethod();
/* more code ... */

/* developer A: changes variable name */
int x = 5; /* change to `x`, it better describes the purpose */
x = doSomething(x);

anotherMethod();
```

[4] least common ancestor

```
/* more code ... */

/* developer B: new code, using variable a */
int a = 5;
a = doSomething(a);

anotherMethod();
/* more code ... */

int x = a << 2; /* store a*4 in new variable x */

/* merged version */
int x = 5; /* change to `x`, it better describes the purpose */
x = doSomething(x);

anotherMethod();
/* more code ... */

int x = a << 2; /* store a*4 in new variable x */
```

Both versions before the merge in Listing 2.3 compile and run without problems, but the merged version will not be compilable, because of the undeclared variable ‘a’ and the double declaration of the variable ‘x’. Such problems can only be avoided by good communication between members of the project team.

It is best to always check and test automatic merge resolutions of version control systems; most projects have a policy that every commit has to be compilable. Thorough unit tests help to discover errors introduced by merges.

Listing 2.4: A typical conflict with conflict markers

```
<<<<<<<
int x = 2; // fast
=======
int x = 4; // accurate
>>>>>>>
```

When a version control system detects textual conflicts during merge or patch operations, i. e. two changes modify the same set of lines, it usually inserts both versions of the changed line into the final merged file.

Conflict hunks start with ‘<<<<<<<’ and end with ‘>>>>>>>’, both versions of a changed set of lines are printed between these two markers and are separated by ‘=======’. The *diff3* format prints the original unchanged line between ‘|||||||’ and ‘=======’. A simple merge conflict with conflict markers is shown in Listing 2.4.

Version control systems often put additional information after conflict markers to help developers to resolve the conflict, e. g. version numbers or identifiers for each side of the merge hunk.

2.2.6 Mainline, Branches and Tags

A *branch* is "a forked line of development in your project, with the line that has been forked off called the branch, and the main line the trunk" [Vesperman, 2003, 4.3 Branching]. Depending on the type of version control system it is possible to branch a group of files, directories or the whole project – this applies to tags as well.

[Vesperman, 2003, 4.3.1 Uses for Branches] lists five common use cases for branches:

- Variations on a theme, such as stored configurations for similar servers
- Bugfix management
- Experimental work, such as experimental code or a new web page design
- Major changes, such as complete code rewrites
- Release candidates for testing

[Vesperman, 2003, 4.4.1 Branching Philosophies] describes different branching philosophies and styles for CVS, but they apply to all version control systems which support branching and merging. There are two philosophies, *basically stable* and *basically unstable*. Basically stable only allows code in the trunk, which is verified to work, whereas development happens exclusively in branches. Release candidates can be quickly created by taking code from the stable and slowly changing trunk.

With the unstable philosophy development mainly happens in trunk. When planning a release, a new branch is started on which the code base is prepared for the release. Most of the time the trunk contains buggy code which might not even compile.

In addition to these branching philosophies, several branching styles exist. While it makes sense to only use one philosophy per project and enforce its usage, different branching styles can be used in a single project.

Branches can be categorized by their duration and direction of merging (cf. [Vesperman, 2003, 4.4 Branching Strategies] [Pilato et al., 2004, Chapter 4, Section 4]):

- **Long branch, merging to branch:** This style should be used when code in the branch should not affect code in the trunk, but the branch needs changes from trunk. It is mainly used to refactor code or develop bigger features which are incompatible with existing code; newer version control systems usually speak of a *feature branch* or *topic branch*. Typically it is used in combination with the philosophy of *basically stable*.

- **Long branch, merging to trunk:** If a branch should not be affected by development in trunk, but the trunk should receive changes from the branch, this model can be used. It is mainly used to make corrections in the branch, which should be applied to the trunk as well; newer version control systems often call it a *maintenance branch* or *release branch.* It can be used with stable and unstable trunk policies.

- **Long branch, merging both trunk and branch:** Long lived branches which depend on new code from trunk, but whose changes can be reintegrated back to trunk early should use this strategy. This can also be a feature branch, where a feature is added step by step.

- **Short branches:** Short lived branches are used to make simple changes or to add small features. Long branches merging in both directions can be simulated using a series of short branches; instead of merging the trunk to the branch, a new branch is created off trunk. When both, trunk and branch, change significantly, this might be the right method.

- **Nested branches:** Branches can not only be created from the trunk, but also from other branches. The parent branch then acts as a virtual trunk – every discussed branching style can be used for nested branches.

Chapter 3

Centralized Version Control

Centralized version control builds upon the idea of a client-server architecture, where a single central server stores a repository which can be accessed by clients. Only privileged users can update the repository on the server, but anonymous clients may be granted read privileges.

Branches can be created only on the server and are visible to everyone by default. The server assigns revision identifiers and numbers to each commit. These revision numbers are authoritative and globally unique.

3.1 Tools

This section will present two very popular centralized version control systems, namely CVS and Subversion. The legacy systems RCS[1], SCCS[2] and CSSC[3] will not be discussed, as they are mostly irrelevant nowadays and do not work for multiple users.

3.1.1 Concurrent Versions System (CVS)

The project CVS was started to extend RCS – one of the first version control systems – to work over the network and to overcome some of its shortcomings. RCS could only control one file at a time and had no notion of projects. RCS stored the history of a file in the same directory as the versioned file under the same name with '`,v`' appended (the history of the file '`file.txt`' was stored in '`file.txt,v`').

In its beginning CVS was a collection of wrappers calling the RCS commands, which is still visible in its file format (cf. [Vesperman, 2003, 6. Repository Management]). Revisions are recorded per file and are committed

[1] Revision Control System

[2] Source Code Control System

[3] Compatibly Stupid Source Control, the GNU implementation of SCCS

independently and as a consequence of this, commits are non-atomic. History graphs could be different between files and are basically independent.

The history of revisions of project files is stored in RCS format and CVS uses the same dotted decimal revision numbering scheme as RCS, where a revision is represented by a prefix and revision identifier (e. g. '`2.6.2.1`'). New revisions increment the revision identifier (number after last period) and new branches add a new fragment (the branch number) before the revision identifier [Vesperman, 2003, 4.3.7 Branch revision numbers]. CVS inherits the inability of RCS to properly record file renames and to move or delete directories.

A particular state of the whole project has to be marked using tags or named branches, since revision numbers are independent of each other and commit times are unreliable due to the non-atomicity of commits. This conceptionally groups files and revisions. A file can be in multiple branches and tags at the same time.

The latest stable version of CVS (1.11.23) was released on May 8th, 2008.

3.1.2 Subversion

> *Why does this project exist?*
>
> To take over the CVS user base. Specifically, we're writing a new version control system that is very similar to CVS, but fixes many things that are broken.
>
> — Subversion FAQ

Subversion was started in May 2000 by CollabNet, Inc.[4] as a replacement for CVS, because CVS had some obvious design flaws inherited from RCS (non-atomic commits, per-file revision numbers). Subversion uses an user interface very similar to CVS, so users proficient in CVS should have minimal problems learning Subversion. In August 2001 Subversion became self hosting, the team migrated from a CVS repository to a Subversion repository.

Subversion stores older revisions using *skip deltas*, an approach similar to skip lists. Deltas are not generated against the immediate predecessor revision, but against an earlier revision which is determined by an algorithm. Subversion uses bit-arithmetic to produce delta lists of length $\text{ld}(revision)$[5].

Due to the non-atomicity of operations in CVS, users could check out files from a repository while another developer was committing and therefore might only get part of the new changes. Also a commit operation could fail, after some files were already written to the repository. This was usually caused by an out-of-date sandbox. The developer then had to update his sandbox, resolve potential conflicts and commit again. All this time the

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

[5] http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas

repository was in an untested and probably inconsistent state, visible to every other user.

Commits in Subversion are transactional and do not exhibit this problem. When a commit fails, the repository is rolled back to a known good state and the developer can fix conflicts without worrying about others. Atomic commits enable Subversion to use repository-global revision numbers.

Subversion does not differentiate between branches, tags and the project trunk (mainline). Each of them is represented by standard directories, thus allowing developers great freedom in arranging their repository. The recommended repository layout is to have a '`trunk`' folder for mainline development, a '`branches`' folder for branches and a '`tags`' folder to tag certain revisions [Pilato et al., 2004, Repository layout].

Because Subversion allows sparse checkouts (checkout of a subdirectory of the repository), it is also common practice to host multiple projects in a single repository. In this case projects are hosted in separates directories each containing the trunk, branches and tags structure. First and second level of the directory might be switched in such a case though, so the trunk, tags and branches directories contain the project sub-folders[6].

Branches and tags in Subversion are therefore mutable. While this is desirable for branches, tags should never change. The only thing that differentiates the trunk from branches and from tags in Subversion is convention and agreement between developers.

Subversion is still under active development and Subversion 1.6.15 was released on October 24th, 2010.

[6] The best example is the Apache Software Foundation repository, browsable at http://svn.apache.org/viewvc/

Chapter 4

Distributed Version Control

4.1 Differences to Centralized Version Control Systems

Network traffic and bandwidth are cheap these days, but disk space was and is much cheaper. Additionally there will be always times when there is no network or Internet around, but usually developers have access to their hard-drive. This is one of the many reasons, why modern version control systems took the step from using a single central server to distributing source code across hosts.

Having the complete repository on a local hard disk enables developers to experiment with the code base without influencing the work of others. A disadvantage of this model is that deleted files have to be copied every time a repository is cloned, which might incur a performance loss on projects with long history. With source code files this usually isn't a problem, but with a lot of binary files which compress badly, the size of a repository might grow too fast.

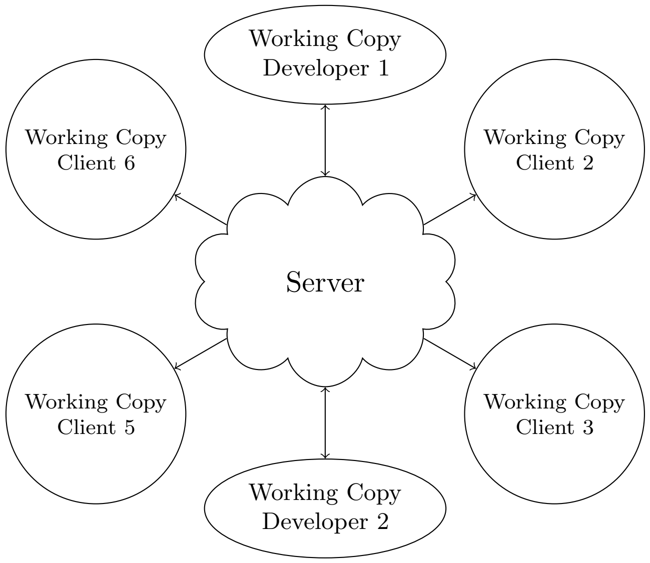
In centralized version control users make changes in their working copy and interact directly with a single main repository. To store and retrieve new changes 'commit' and 'update' is used. Committing changes in a distributed version control system only works on the local copy of the repository, thus new commands to synchronize distinct repositories are needed.

Most systems adapt a push/pull workflow, where developers selectively pull changes from other developers or remote repositories, and push changes to remote copies.

Whereas conflicts in centralized version control occur directly on update and could destroy a developers work during update, users using the distributed approach can delay updating/merging their working copy until they are ready to do so.

While branching in centralized version control systems was very easy (a branch in Subversion is just a copy of the trunk or any other directory, created

Figure 4.1: Centralized development model



with the `svn copy` command), merging often led to problems. Subversion only recently gained the ability to automatically merge branches without specifying revisions (merge tracking properties). Prior to this developers had to remember the start and end points of branches.

Because every developer effectively owns a branch of a project, branching and merging is possibly the most common task when using distributed version control systems collaboratively.

When renaming a file in a branch and changing it in the trunk or another branch, this would result in a tree conflict [Pilato et al., 2004, Keeping a reintegrated branch alive]. To properly merge branches in Subversion great care must be taken when working with a branch.

Distributed systems internally maintain a DAG[1] of history, which allows them to identify exactly which changes have to be merged and which changes have already been merged. Changes can follow renames of files and branches can be merged to and from the trunk many times without inherent problems.

Copying (branching) is easy, but integrating and synchronizing changes (merging) between different copies is difficult and the relevant aspect.

[1] Directed acyclic graph

Figure 4.2: Distributed development model

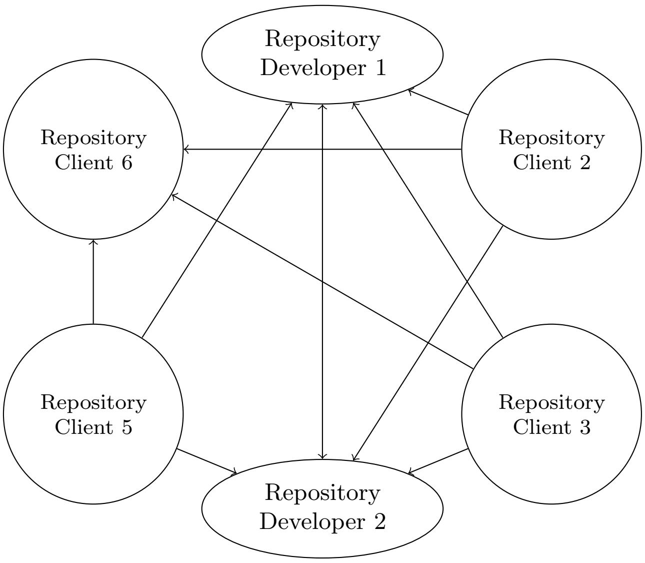
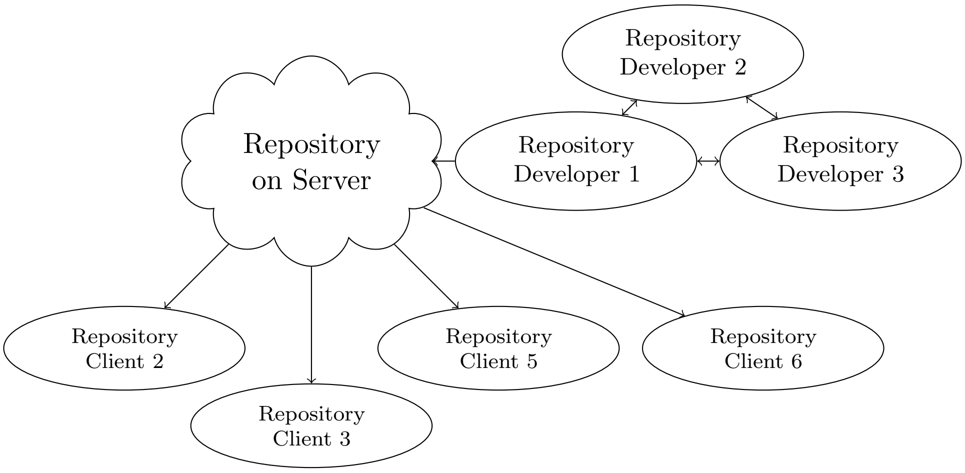


Figure 4.3: Distributed development model in combination with server



4.2 Concepts of Distributed Version Control Systems

As mentioned in the previous section, a significant key feature of distributed version control systems is their distributed nature: Each repository of the same project is treated equally, eliminating the need for a main repository server, thus reducing the number of network operations. This allows developers to implement bigger features without disturbing or breaking the work of other developers.

4.2.1 Revision identifiers

[Baudiš, 2009] mentions three important properties of revision identifiers:

- **Uniqueness:** A revision number is guaranteed to be unique in a repository.
- **Simplicity:** Easy to remember and to use (e. g. incrementing numbers give information about the order in which commits happened).
- **Stability:** Identifiers should not change between repositories or over time.

While centralized version control system can satisfy all three requirements, distributed systems cannot satisfy all three directly. A central authoritative server can assign revision numbers which are unique per project, stable and easy to use. Subversion produces simple integers to represent revisions.

Distributed systems can use hash functions to produce unique and stable revision identifiers, but those are usually not easy to remember and use. Mercurial and Bazaar offer local incrementing revision numbers which are easy to use, but not stable, i. e. they change between different repositories or after a merge operation.

4.2.2 Integrity

It must be possible to verify the content and history of a repository. One-way cryptographic hash functions are very good at achieving this. Hash functions turn arbitrary messages (M) into a fixed-length hash value ($h = H(M)$) [Schneier, 1996, Chapter 18.7 Secure Hash Algorithm (SHA)]. One-way hash functions have additional characteristics, which make them perfect candidates for verifying files and repositories. Systems which use hashes to identify certain objects also require collision resistance from the hash function.

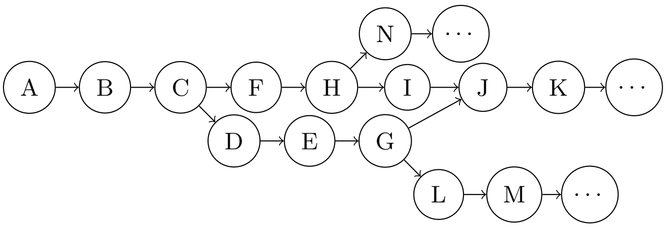
A hash function must meet the following requirements:

- Given M, it is easy to compute h
- Given h, it is hard to compute M such that $H(M) = h$
- Given M, it is hard to find another message, M', such that $H(M) = H(M')$
- It is hard to find two random messages, M and M', such that $H(M) = H(M')$ (Collision resistance)

Most distributed version control systems use SHA1[2] as their hashing

[2] Secure hashing algorithm

Figure 4.4: Directed acyclic graph representing history of a project



function. Hashes generated by SHA1 are of length 160 bit.

Listing 4.1: SHA1 example

```
SHA1('distributed version control') =
  '992ecca98df8f64297e65763e5be119b311971c2'
```

4.2.3 History representation

History in distributed version control systems can be represented as a DAG[3], where each commit points to its ancestor or ancestors. A directed graph (or digraph) consists of a non-empty finite set of elements called *vertices* and a finite set of ordered pairs of distinct vertices called *arcs*. A digraph is acyclic if it has no cycle [Bang-Jensen and Gutin, 2008, 2.1 Acyclic Digraphs].

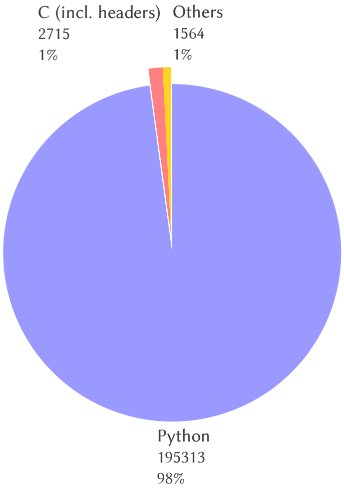
$N^{+}_{D}(v) = \{u \in V - v : vu \in A\}$ is called the *out-neighbourhood* and $N^{-}_{D}(v) = \{w \in V - v : wv \in A\}$ is called the *in-neighbourhood*. Commits with an *out-neighbourhood* greater than one are merges in DVCS terminology, where as commits with an *in-neighbourhood* greater than one are branching points in history. A commit can be a merge commit and a branching point at the same time. For initial commits in a project the *out-neighbourhood* is zero, and branch tips have an *in-neighbourhood* of zero. Such a DAG can be seen in Figure 4.4.

4.3 Tools

This section explains the object and storage model of Bazaar, Git and Mercurial. Listings are made against the official upstream version of each project and can be run at any later time to get the same results.

[3] directed acyclic graph, acyclic digraph

Figure 4.5: Lines of code (Bazaar)



4.3.1 Bazaar

> The number one thing I want from a distributed version control system is robust renaming. [. . .] Distributed version control is all about empowering your community, and the people who might join your community.
>
> — Mark Shuttleworth

Development for Bazaar was started by Canonical in an effort to improve one of the early distributed systems available for version control, GNU Arch[4]. Started as a branch to make working with Arch more comfortable, Canonical soon saw that Bazaar evolved differently than Arch and had different goals – Bazaar was born.

Bazaar hides implementation details of underlying structures and abstracts access to low level objects through a stable API. Upgrading a data format thus has little effect on higher levels; in fact the internal repository format was already changed several times in the past.

Bazaar is the only one of the analyzed systems which is able to track empty directories.

[4] http://wiki.bazaar.canonical.com/HistoryOfBazaar, http://www.gnu.org/software/gnu-arch/

Branches in Bazaar are created by cloning a repository (a repository is called *branch* in Bazaar); each Bazaar repository may only have one branch head.

As a workaround Bazaar provides shared repositories[5] to store multiple branches in a single repository. Working copies can also be detached from branches, which allows to mimic a workflow similar to centralized systems, where commands work directly on a central server. Such working copies can be converted back to distributed independent branches.

Projects which use Bazaar for version control include Ubuntu, MySQL, Inkscape, Bugzilla[6].

Repository Structure

Text

Text objects in Bazaar store line-based text files (e. g. source code). The exact storage on disk is unspecified by Bazaar and depends on the storage format chosen for a branch. Earlier versions of Bazaar stored files in *weave* format, newer versions use *format 2a* which uses a groupcompress algorithm to achieve better performance and scalability.

Text objects are accessed through Bazaar's API with the tuple `(file_id, rev_id)`. When Bazaar is instructed the first time to track history for a file, it creates a unique, permanent identifier for the file. File ids have the format "`file name - timestamp (YYYYMMDDhhmmss) - 16 char random string - number`".

When importing projects from other version control systems, Bazaar creates the file_id from information specific to that version control system. A file_id for a file imported from Subversion looks like "`1864@b9310e46-f624-0410-8ea1-cfbb3a30dc96: %2Flinux%2FX%2Fsymbols%2Fde`". Referencing files through their unique and permanent file_ids allows Bazaar to merge across file renames.

Revision

Revisions provide additional metadata to a state of the project and contain the author, a timestamp, the commit message and a list of parents. Revisions imported from other systems usually contain metadata such as original revision numbers and committer names.

Parents do not have to exist in the same repository (ghost revisions, e. g. 'mbp@sourcefrog.net-20050711064100-c2eb947e0212f487' in bzr.bzr), which allows Bazaar to do lightweight checkouts. This implies that it is not possible to verify that a revision identifier links to an existing revision.

[5] http://wiki.bazaar.canonical.com/SharedRepositoryLayouts

[6] http://bugzilla.org, http://ubuntu.com, http://mysql.com, http://inkscape.org

Revision identifiers are a simple strings of the form "`email - timestamp (YYYYMMDDhhmmss) - 16 char random string`" and are created when committing a new revision.

Listing 4.2: Revision ids starting at r5410

```
$ bzr log --show-ids \
  -r 1..revid:pqm@pqm.ubuntu.com-20100906113342-s41muavhjutdc7xr \
  | grep ^revision-id | cut -d' ' -f2
pqm@pqm.ubuntu.com-20100906113342-s41muavhjutdc7xr
pqm@pqm.ubuntu.com-20100903025310-t8mj1bjq4fsyxk7p
pqm@pqm.ubuntu.com-20100903013246-mydkx60um8b2trfq
pqm@pqm.ubuntu.com-20100903001355-l29hnxtjgnlhpq2f
pqm@pqm.ubuntu.com-20100902225129-83167cameln73xz2
...
```

Bazaar exposes CVS-like revision numbers similarly to its users, where revisions on a branch are prefixed with a branch number. Revision numbers represent a view on the local repository, mainline revisions are simple increasing integers.

Branch revisions are represented in a dotted decimal fashion, the first number being the revision when the branch was created, the second number standing for the branch number (if multiple branches are branched from the same commit or a branch is created from a branch) and the last number is again an incrementing integer which is local to that branch. When referring to revision numbers it is necessary to mention a branch URL as well.

Currently, Bazaar (format 2a) stores revision data in bencoded[7] form.

Listing 4.3: Bencoded data of a revision

```
$ bzr cat-revision pqm@pqm.ubuntu.com-avhjutdc7xr342-s41mua
l16:formati10eel9:committer54:Canonical.com Patch Queue Manager <pqm@pqm.
    ubuntu.com>el8:timezonei3600eel10:propertiesd11:branch-nick6:+
    trunkeel9:timestamp14:1283772822.691el11:revision-id50:pqm@pqm.
    ubuntu.com-20100906113342-s41muavhjutdc7xrel10:parent-idsl50:pqm@pqm.
    ubuntu.com-20100903025310-t8mj1bjq4fsyxk7p52:v.ladeuil+lp@free.fr
    -20100906100836-ufi18ftz3zx901o7eel14:inventory-sha140:76
    eedc47510eefa74efa417deba2971abbf20e8ae17:message93:(vila) Cleanup
    imports in bt.per_wt.test_pull (most of them were useless).
 (Vincent Ladeuil)ee
```

Inventory

An *Inventory* describes the tree state of the project for a given revision. It contains a list of paths and file_ids. An entry additionally covers the following fields: the file_id of the parent directory, revision id of the file, executable flag, SHA1 of the file contents and the size of the file.

[7] BitTorrent encoded

Subsequent inventories can be expressed as deltas against the previous inventory. Such a delta consists of a list of tuples '`(old path, new path, file id, new entry)`'. Different values for old path and new path indicate a rename. Setting either to the special value 'None' results in an addition or removal of the file.

Listing 4.4: Inventory for r5410

```
$ bzr inventory --show-ids \
  -r revid:pqm@pqm.ubuntu.com-20100906113342-s41muavhjutdc7xr
.bzrignore                bzrignore-20050311232317-81f7b71efa2db11a
.rsyncexclude             rsyncexclude-20050408053852-27e0a5928b682...
.testr.conf               testr.conf-20100228092111-z02a13qrv22mn7up-1
BRANCH.TODO               BRANCH.TODO-20060103052123-79ac4969351c03a9
COPYING.txt               gpl.txt-20060725144612-kxut42v3nkatynfv-1
INSTALL                   INSTALL-20051019070340-4b27f2fb240c7943
MANIFEST.in               manifest.in-20100109214549-ime1ec4zij1zkovi-1
Makefile                  Makefile-20050805140406-d96e3498bb61c5bb
NEWS                      NEWS-20050323055033-4e00b5db738777ff
NEWS-template.txt         newstemplate.txt-20100219053124-f4a3zo3uji...
...
bzrlib/__init__.py        __init__.py-20050309040759-33e65acf91bbcd5d
bzrlib/_annotator_py.py   _annotator_py.py-20090617192546-21fnjrg2s2...
bzrlib/_annotator_pyx.pyx _annotator_pyx.pyx-20090623201937-ic33ic1a...
bzrlib/_bencode_pyx.h     _bencode_pyx.h-20090604155331-53bg7d0udmrv...
...
```

Testament

Testaments are used to certify revisions as authentic. Two semantically equal revisions will have the exact same testament. Testaments are generated from a subset of information from a particular revision, which is considered stable even when the underlying storage format of revisions changes.

The following fields are stored in a testament: testament version, revision-id, unix timestamp, timezone identifier, parent revision-ids, commit message of the corresponding commit, a list of inventory entries in canonical form plus hashed contents of files, branch name.

Listing 4.5: Long testament for r5430.1.2

```
$ bzr testament --long \
  -r revid:andrew.bennetts@canonical.com-20100917065959-p7syp0v4z3aep4y8
bazaar-ng testament version 1
revision-id: andrew.bennetts@canonical.com-20100917065959-
    p7syp0v4z3aep4y8
committer: Andrew Bennetts <andrew.bennetts@canonical.com>
timestamp: 1284706799
timezone: 36000
parents:
  andrew.bennetts@canonical.com-20100917043523-c5t63gmvxqxmqh5j
  pqm@pqm.ubuntu.com-20100917064608-qm7k0sez9941oj85
message:
  Merge latest lp:bzr/2.2.
inventory:
  file .bzrignore bzrignore-20050311232317-81f7b71efa2db11a
    cc2b0834b2e5e62257a36d3f4d1f06a67c382142
  ...
  directory apport contribapport-20100131162357-aladkx1ilh730byb-1
  file apport/README readme-20100131163221-0u77xzj0p9auj54u-1 5
    cc4bad4e5254bde9025942f0d9102533593ae35
  file apport/bzr-crashdb.conf bzr.conf-20100131163221-0u77xzj0p9auj54u
    -2 27d1fb56418582428a7d46f633cf193d5080788d
  file apport/source_bzr.py source_bzr.py-20100131163221-0
    u77xzj0p9auj54u-3 10cc21ac457c0b0bce6aacc373f89c160847d131
  file bzr bzr.py-20050313053754-5485f144c7006fa6
    eadc881054d9a8f11667e9cc6073605e6a19f3a8
  file bzr.ico bzr.ico-20060629083000-q18ip0hk7lq55i4y-1 1
    bc526f63a65fae06ab88d238559b1a52e5fbd1f
  directory bzrlib bzrlib-20050309040749-4ac9a0e211602846
  ...
properties:
  branch-nick:
    merge-2.2-into-devel
```

The short form which is used for signing only contains the testament version, revision-id and the hash of the long form. When a testament is signed, the hash function which was used to generate the hashes is prepended to the message.

Listing 4.6: Signature of r5430.1.2

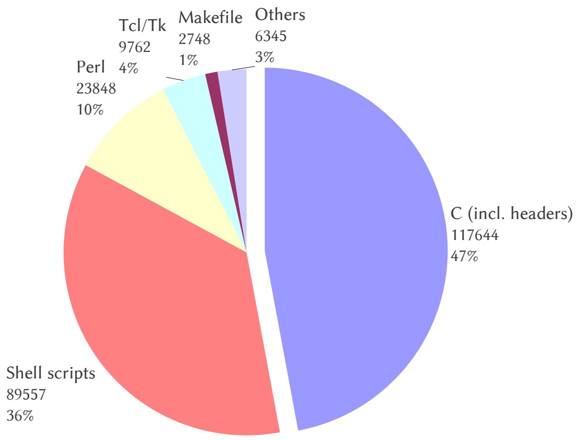
```
$ bzr cat-signature \
  -r revid:andrew.bennetts@canonical.com-20100917065959-p7syp0v4z3aep4y8
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

bazaar-ng testament short form 1
revision-id: andrew.bennetts@canonical.com-20100917065959-
    p7syp0v4z3aep4y8
sha1: 9243b8fc132d437c52615383f473b93ef2e63f3f
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.10 (GNU/Linux)

iEYEARECAAYFAkyTEfAACgkQkHkW2hvmQuWolwCeI4xSqEMIFmHJIPSgt+tNkSih
HwcAnAh7Yrj3UH8Xj24Gpkbis3TqGbQz
=WSEl
-----END PGP SIGNATURE-----
```

A strict form of the testament exists, which includes additional information for each inventory entry, namely the last changed revision and the executable flag.

Figure 4.6: Lines of code (Git)



4.3.2 Git

> [. . .] It's not an SCM, it's a distribution and archival mechanism. I bet you could make a reasonable SCM on top of it, though. Another way of looking at it is to say that it's really a content-addressable filesystem, used to track directory trees.
>
> — Linus Torvalds

Git is a distributed version control system started by Linus Torvalds in early April 2005 to aid kernel development. It was developed as a replacement for BitKeeper[8] which changed its license so kernel developers could no longer use it for free. In July 2005 Torvalds selected Junio C Hamano to become the new project maintainer for Git.

The pursued goals were performance and scalability. Git had to work well for thousands of developers[9] and tens of thousands of files[10]. In the beginning Git was a collection of shell scripts which called a few core commands written in C. Over time many of the shell scripts were replaced by native C code to improve performance of Git.

Git makes heavy usage of the SHA1 hashing algorithm to verify and guarantee project integrity. This makes it almost impossible for an attacker

[8] http://www.bitkeeper.com

[9] 7022 unique authors in Linux kernel 2.6.36-rc3

[10] 33556 source files, 13234910 lines of code, 210153 commits, 449 MB, Linux kernel 2.6.36-rc3, since April 16, 2005

to change information in the repository without others noticing it. Git's diff algorithm is based on LibXDiff[11] since 25th March 2006 which has since then been heavily extended and optimized for Git.

Projects which use Git for version control include the Linux kernel, Android, Wine, X.org[12].

Repository structure

Git's storage model is not based on changes but on snapshots. Each commit will point to a full tree object which in turn points to other tree objects or complete files (blobs). Because blobs and trees can be identified by their hash, already existing objects can be reused and do not have to be stored multiple times [Chacon, 2009, 1.3 Snapshots, Not Differences].

This means a git commit object will always contain the hash of a tree object which is a full snapshot of a project's working tree state, but only changed objects are newly stored with each commit and existing objects are simply re-used. Git can *delta compress* objects in a self-contained *packfile* to preserve space. Integrity is guaranteed through a SHA1 checksum over the compressed contents.

Commit

Commit objects contain the hash of the associated tree object, a list of zero or more parents hashes, the author name and creation time, the committer name and commit time and a commit message.

Listing 4.7: An octopus merge commit

```
$ git cat-file commit 211232bae64bcc60bbf5d1b5e5b2344c22ed767e
tree cdafa88fa4ed7fcc7bb6c64d62e2d7c4d3b65e42
parent fc54a9c30ccad3fde5890d2c0ca2e2acc0848fbc
parent 9e30dd7c0ecc9f10372f31539d0122db97418353
parent c4b83e618f1df7d8ecc9392fa40e5bebccbe6b5a
parent 660265909fc178581ef327076716dfd3550e6e7b
parent b28858bf65d4fd6d8bb070865518ec43817fe7f3
author Junio C Hamano <junkio@cox.net> 1115335014 -0700
committer Junio C Hamano <junkio@cox.net> 1115335014 -0700

Octopus merge of the following five patches.

  Update git-apply-patch-script for symbolic links.
  Make git-prune-script executable again.
  Do not write out new index if nothing has changed.
  diff-cache shows differences for unmerged paths without --cache.
  Update diff engine for symlinks stored in the cache.

Signed-off-by: Junio C Hamano <junkio@cox.net>
```

[11] http://www.xmailserver.org/xdiff-lib.html

[12] http://kernel.org, http://android.com, http://winehq.org, http://x.org

In Git a commit object can contain an arbitrary number of parents. A commit without parents is called a *root commit*. In centralized version control this would be commit number 1. Projects managed with Git can have more than one root commit and a repository can even contain several unrelated projects.

Normal commits in history have a single parent and are created by editing files, adding changes to the index and then calling the commit command. Running the merge command will create a new commit with more than one parent, a *merge commit*. A special form of the merge commit is the so-called *octopus merge*, which has more than two parents. Git has no notion of local revision numbers as other distributed version control systems have. To reference commits users have to use their hash id or branch names. Git provides a special syntax to access ancestors of commits. The most commonly used are shown in the following listing:

Listing 4.8: Specifying parents

```
^n  selects the n-th parent of a merge, if n is omitted 1 is assumed
~n  selects the n-th parent of the commit, following the first parent of merge commits
@{n}  selects the n-th last commit to which the ref pointed (cf. reflog)
^{type} dereferences an object to the specified type. i.e. v1.7.2^{tree} will
  return the tree object referenced by the commit referenced by the v1.7.2 tag
```

These tokens can be combined arbitrarily to specify any commit in the ancestry line. ‘master^^2~4’ will select the fourth parent of the right parent of the merge which happened one commit before master.

Commit objects are the only objects in Git which store ancestry information and allow to walk history – tree and blob objects are referenced either directly or indirectly by a commit object.

Listing 4.9: Commit 05a59a0

```
$ git cat-file commit 05a59a087c29c0b5dd267ec0bd488829427cc3d7
tree 358d4a3faafb532c75bff09a14e80244c587a0aa
parent d0b16c8f878bef5c1268e033a3d1f427498c7008
author Daniel Knittl-Frank <knittl89+git@googlemail.com> 1274795151 +0200
committer Junio C Hamano <gitster@pobox.com> 1275602529 -0700

Show branch information in short output of git status

This patch adds a first line in the output of `git status -s` when given
the option `-b` or `--branch`, showing which branch the user is
currently on, and in case of tracking branches the number of commits on
each branch.

Signed-off-by: Daniel Knittl-Frank <knittl89+git@googlemail.com>
Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

Tree

Tree objects are a snapshot of a particular revision of the project, mapping hashes of `blob`s and sub`tree`s to paths and permissions (only the executable bit is stored). This recursive approach is very similar to file systems, where a directory can contain both files and directories.

Listing 4.10: The tree 358d4a3

```
$ git ls-tree 358d4a3faafb532c75bff09a14e80244c587a0aa
100644 blob 5e98806c6cc246acef5f539ae191710a0c06ad3f  .gitattributes
100644 blob 14e2b6bde9bef55d678da8ba44dc180b039cd3ac  .gitignore
100644 blob a8091eb5dfa430bf1b0537da47a31e7cf88d8622  .mailmap
100644 blob 536e55524db72bd2acf175208aef4f3dfc148d42  COPYING
040000 tree 8f387b2039a59b2915ec6db7be2ccafe311981ba  Documentation
100755 blob e45513dee938dde3a8428a833fb43023b04ca95b  GIT-VERSION-GEN
...    ...  ...                                       ...
100644 blob 636ecdd896c1ceecf79ede6caf4e61c519681053  wt-status.c
100644 blob 4f190454e5b2ad46b33894fb55aa7dd8dd28d604  wt-status.h
...    ...  ...                                       ...
```

Listing 4.11: ‘`Documentation`’ subtree

```
$ git ls-tree 8f387b2039a59b2915ec6db7be2ccafe311981ba
100644 blob ddb030137d54ef3fb0ee01d973ec5cee4bb2b2b3  .gitattributes
100644 blob 1c3a9fead579a9b52037f1bbe245998db8a2f40b  .gitignore
100644 blob b8bf618a30fd32a014e41e1ba9914f5e652bdefd  CodingGuidelines
100644 blob 04f69cf64e5d989bac3cd1c235e6a7e657c6c103  Makefile
100644 blob fea3f9935b7794ce86f04d22c9d68fb9d537167d  RelNotes-1.5.0.1.
    txt
...    ...  ...                                       ...
```

Blob

Blobs[13] represent the actual file data. Each blob gets the SHA1 hash of its contents assigned.

Blobs are created in the ‘.git/objects/’ directory, when a file or changes are added with the `git add` command.

Listing 4.12: Content of blob 4f19045

```
$ git cat-file blob 4f190454e5b2ad46b33894fb55aa7dd8dd28d604
#ifndef STATUS_H
#define STATUS_H

#include <stdio.h>
#include "string-list.h"
#include "color.h"

enum color_wt_status {
  WT_STATUS_HEADER = 0,
  WT_STATUS_UPDATED,
...
```

[13] binary large object

Tag

Tags in git reference any other type (commit, tree, blob) and can be used to give that object a meaning (e. g. ‘v1.7.2’ for a commit or ‘junio-gpg-pub’ for a blob). Tags can be annotated, i. e. equipped with an additional message. Such messages are very useful for release management and often contain release notes or the changelog for that release.

Additionally, annotated tags can be signed with a PGP[14] key to confirm the authenticity of the object it points to [Schneier, 1996, Chapter 20, Public-Key Digital Signature Algorithms]. Due to Git’s internal data model and feedback of object hashes to all commits, this does not only sign the current state of the project, but the complete history – i. e. how this state was reached – of the project.

Listing 4.13: Annotated tag v1.7.2 with signature

```
$ git cat-file tag v1.7.2
object 64fdc08dac6694d1e754580e7acb82dfa4988bb9
type commit
tag v1.7.2
tagger Junio C Hamano <gitster@pobox.com> 1279742158 -0700

Git 1.7.2
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.9 (GNU/Linux)

iEYEABECAAYFAkxHUM8ACgkQwMbZpPMRm5oZcQCggTUlgYmUlHiafN/J83cRLWl9
ZGgAoIUiBUHFM2OyefhglVMRdya1gUWn
=FIkP
-----END PGP SIGNATURE-----
```

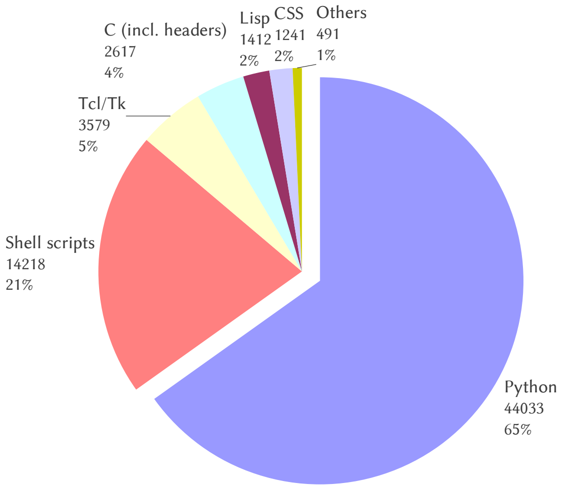
Staging Area

Git introduces another level of abstraction which most other version control systems do not implement or hide from the user. The *index*, also called the *staging area*, is used to prepare and mark changes for commit. The index describes the state of the tree object which will be used when running `git commit`. Content can be added and removed incrementally from it to build up commits.

In case of merge conflicts the index contains information about which line was changed by which parent and which hunks are still unresolved. The user can then incrementally fix conflicts and diff against the index too see which conflicts are still unresolved.

[14] Pretty Good Privacy

Figure 4.7: Lines of code (Mercurial)



4.3.3 Mercurial

> Much thought has gone into what the best asymptotic performance can be for the various things an SCM has to do and the core algorithms and data structures here should scale relatively painlessly to arbitrary numbers of changesets, files, and file revisions.
>
> — Matt Mackall

The first announcement about Mercurial was made late April 2005 by Matt Mackall on the Linux Kernel Mailing List[15]. The project became self-hosting on May 3rd.

Most parts of Mercurial are written in Python[16], which makes it easily portable to any system with support for Python 2.4. At the moment it is possible to install Mercurial on Unix-like systems (Linux, Mac OS X) and Windows. Mercurial's main diff algorithm is similar to the diff algorithm in python's difflib [Mackall, 2006] and is one of the few pieces of Mercurial implemented in C for performance reasons.

Projects which use Mercurial for version control include Mozilla Projects, OpenOffice.org, Python[17].

[15] http://lkml.indiana.edu/hypermail/linux/kernel/0504.2/0670.html

[16] http://python.org/

[17] http://python.org, http://openoffice.org

Revlog

Mercurial stores its history in a structure called a *revlog*. A revlog can take three forms: filelog, manifest and changelog. It uses two files, the index file with the extension '`.i`' and the actual data file with the extension '`.d`'.

A revlog index entry uses 64 bytes and consists of the following fields:

- **nodeid:** The SHA1-hash to identify this revision is calculated by concatenating parent ids – sorted ascending by value – and the data of the revlog entry.
- **first parent revision:** This field is set to the parent revision. The first entry in a revlog has both parents set to -1.
- **second parent revision:** This field is only set if this entry is the result of a merge operation, otherwise it defaults to -1.
- **offset:** The offset of this revision's data in the data file.
- **length:** Compressed and uncompressed length of the revision data in bytes.
- **base revision:** Revision used as base for delta calculation.
- **link rev:** Revision number of the changelog, when this revlog entry was created. For a changelog this field is equal to the revision number of that entry.
- **flags:** Contains the version of the revlog format and additional properties.

Every entry in a revlog is compressed using the deflate compression algorithm, but it is stored uncompressed, if the resulting data would be bigger than the uncompressed data due to additional metadata. This is important for already compressed data (archives, mp3, jpg, and other binary data).

Mercurial stores changes individually per file in a so-called *filelog*. A *manifest* describes the state of the project directory at a certain point in history. A *revision* in Mercurial's repository consists of its SHA1 hash, the author, a pointer to the revision data (the *manifest*) and the two IDs of its immediate parents.

Listing 4.14: Changelog index

```
$ hg debugindex .hg/store/00changelog.i
  rev  offset length base linkrev nodeid       p1           p2
    0       0   305     0       0 9117c6561b0b 000000000000 000000000000
    1     305   152     1       1 273ce12ad8f1 9117c6561b0b 000000000000
    2     457   119     2       2 ecf3fd948051 273ce12ad8f1 000000000000
    3     576   110     3       3 3a6392190075 ecf3fd948051 000000000000
  ...    ...   ...   ...           ...          ...          ...
  100   13434   130   100     100 526722d24ee5 db5eb6a86179 000000000000
  101   13564   144   100     101 6da5cf0c4193 526722d24ee5 000000000000
  102   13708    99   102     102 58039eddbdda 3dde7c87e36d 6da5cf0c4193
  103   13807   132   102     103 33500fe7d56c 47c9a869adee 000000000000
  104   13939   106   104     104 20b3e7aad499 58039eddbdda 33500fe7d56c
  ...    ...   ...   ...           ...          ...          ...
11172 2092830   103 11172   11172 e9226eb3af2a 3d0a9c8d7184 3b3261f6d9ba
11173 2092933   318 11172   11173 5b48d819d5f9 e9226eb3af2a 000000000000
11174 2093251   135 11174   11174 ba78a1bfbfd9 5b48d819d5f9 000000000000
11175 2093386   236 11174   11175 39e7f14a8286 ba78a1bfbfd9 000000000000
```

Changelog

An example entry of the changelog from a mercurial repository is shown below in Listing 4.15. The changeset ID matches with the nodeid from the changelog index in Listing 4.14 above. The first line is the nodeid of the manifest associated with this revision. The changelog also stores the originating branch of each entry.

Listing 4.15: Data of changelog entry 39e7f14a8286

```
$ hg debugdata .hg/store/00changelog.d \
  39e7f14a828666c2ca2e4560b4a41896a7692326
ffd1749b3205815d234c55ec0ed7ee40fca635e6
Matt Mackall <mpm@selenic.com>
1273849269 18000
tests/test-acl.out
tests/test-backout.out
tests/test-bundle-r.out
tests/test-bundle.out
...
```

Manifest

The *manifest* describes the directory structure of the project at a given time, it maps file revisions to actual files and permissions. Its a flat list, storing the mapping between the stored SHA1 of a file and its location in the directory tree in a canonical form. Additionally the executable bit is stored for each file path. The first few lines from a manifest from the official Mercurial repository can be seen in Listing 4.16.

Listing 4.16: Data of manifest ffd1749b3205 (column delimiters added for clarity)

```
$ hg --debug debugdata .hg/store/00manifest.d \
  ffd1749b3205815d234c55ec0ed7ee40fca635e6
.hgignore                 6d2dc16e96ab48b2fcca44f7e9f4b8c3289cb701
.hgsigs                   3b484f8114aecb9571de2b32224c4950a027c403
.hgtags                   d8363fe5eadc8b10c90b4f56bdc08c2fd744feca
CONTRIBUTORS              7c8afb9501740a450c549b4b1f002c803c45193a
COPYING                   5ac863e17c7035f1d11828d848fb2ca450d89794
Makefile                  0962fb2061a3a21e699324ad757cb57bef257d18
README                    8b836c38b9e9a8644e4468c67aeba1d217db335b
contrib/bash_completion   deb2ad4a6d27e8bc1c05faad96f835f4f85e7ae0
contrib/buildrpm          cf3ace555055fd6080a30f670fe7a0b519319990   x
contrib/check-code.py     5dd34662929933e3fb050e343f68addf0c216541   x
...                       ...
```

Filelog

The *filelog* stores the history of each individual file in the repository. Mercurial stores these files in the directory '`.hg/store/data`' which has the same structure as the worktree. Special measures are taken for cross-platform compatibility: Capital letters are prefixed with an underscore to avoid problems on case-insensitive file-systems, directory names cannot end with a period or space thus the last character is replaced by an underscore, illegal characters are encoded as '`~xx`', xx being a two digit hex code. If the resulting filename would exceed the file-systems maximum length for pathnames, the file is stored under a irreversible hash name in '`.hg/store/dh`'.

A filelog index and the corresponding entry for revision 19 can be seen in Listing 4.17 and Listing 4.18

Listing 4.17: Filelog of '`CONTRIBUTORS`'

```
$ hg debugindex .hg/store/data/_c_o_n_t_r_i_b_u_t_o_r_s.i
rev  offset  length  base linkrev nodeid       p1           p2
...     ...     ...   ...     ... ...          ...          ...
  8     967       0     0     878 85cf8acf767b 136e6e9f03fe af9155a6730e
  9     967       0     0     894 be579a35e6f0 85cf8acf767b d65b9e834aad
 10     967       0     0     895 47a0f8bb36da 85cf8acf767b be579a35e6f0
 11     967       0     0     896 6de0bd02887f d65b9e834aad 47a0f8bb36da
 12     967     128     0    1080 ea694a28b9b1 6de0bd02887f 000000000000
 13    1095     129     0    1231 ef9eec058694 ea694a28b9b1 000000000000
 14    1224     142     0    1310 74e4b8afb382 ef9eec058694 000000000000
 15    1366      52     0    1450 ef3c19e5b938 74e4b8afb382 000000000000
 16    1418      54     0    2120 c1bca402be4e ef3c19e5b938 000000000000
 17    1472      52     0    2162 d0701d05450a c1bca402be4e 000000000000
 18    1524      48     0    2947 900bc7f182a5 d0701d05450a 000000000000
 19    1572     141     0    5514 7c8afb950174 900bc7f182a5 000000000000
```

Listing 4.18: Data stored in nodeid 7c8afb950174

```
$ hg debugdata .hg/store/data/_c_o_n_t_r_i_b_u_t_o_r_s.i \
  7c8afb9501740a450c549b4b1f002c803c45193a
[This file is here for historical purposes, all recent contributors
should appear in the changelog directly]
```

```

Andrea Arcangeli <andrea at suse.de>
Thomas Arendsen Hein <thomas at intevation.de>
Goffredo Baroncelli <kreijack at libero.it>
Muli Ben-Yehuda <mulix at mulix.org>
Mikael Berthe <mikael at lilotux.net>
Benoit Boissinot <bboissin at gmail.com>
Brendan Cully <brendan at kublai.com>
...
```

Listing 4.19: Index of '`README`' file

```
$ hg debugindex .hg/store/data/_r_e_a_d_m_e.i
rev  offset  length   base linkrev nodeid       p1           p2
...     ...     ...   ...     ... ...          ...          ...
 21    4483     248      0     633 df204fa43749 b4476c784622 000000000000
 22    4731     108      0     969 8da5d4f33e95 df204fa43749 000000000000
 23    4839       0      0     981 a903ec460ca8 8da5d4f33e95 df204fa43749
 24    4839     120      0    1308 fd3d82992166 a903ec460ca8 000000000000
 25    4959      12      0    2031 08a20b374b3c fd3d82992166 000000000000
 26    4971     372      0    2208 9c4fcdc8c999 08a20b374b3c 000000000000
 27    5343     939     27    2364 71c45066973e 000000000000 000000000000
 28    6282     330     27    2365 6a400d9ae4ca 71c45066973e 000000000000
 29    6612     148     27    2366 2099b1e5861e 6a400d9ae4ca 000000000000
 30    6760     117     27    2367 87ff382bce0b 2099b1e5861e 000000000000
 31    6877     225     27    2368 5a7fc5d81ff8 87ff382bce0b 000000000000
 32    7102    1380     27    3507 1e5e927aa2c3 9c4fcdc8c999 000000000000
 33    8482     277     27    3689 eea77300976c 1e5e927aa2c3 000000000000
 34    8759      79     27    3847 3c643b91e14c eea77300976c 000000000000
 35    8838     188     35    3935 e4907aefc8dd 3c643b91e14c 000000000000
 36    9026      72     35    8936 a1f3120a2019 e4907aefc8dd 000000000000
 37    9098      16     35    9595 57d6f1e057cd a1f3120a2019 000000000000
 38    9114      12     35    9596 8b836c38b9e9 57d6f1e057cd 000000000000
```

This data model allows to read a specific version of a file from a snapshot and few deltas. Another advantage is, that in case of repository corruption it is still possible to reconstruct the contents of files.

If a user requests revision 38 (or id 8b836c38b9e9) of the '`README`' file, whose filelog index is shown in Listing 4.19, Mercurial will read the fully stored snapshot (rev 35, id e4907aefc8dd) from offset 8838 with a length of 188 plus all deltas up to and including offset 9126 ($9114 + 12$) from the datafile of this filelog ('`.hg/store/data/_r_e_a_d_m_e.d`'). This results in one sequential read from the hard drive for each of both the index and data file.

Its storage model can be compared to modern video compression algorithms. Each version (either a frame or a changeset) is stored as a delta to its previous state. After a given threshold a full snapshot of the data is stored, which in video compression relates to a keyframe.

Merges in Mercurial are recorded in all three forms of a revlog: revision, manifest and file.

Other objects

Tags in Mercurial are stored in the '`.hgtags`' file. It lives in the project root like any other tracked file. Each line contains the full 40 char SHA1 of a changeset and after a whitespace the name of the tag.

Commit signatures are stored inside the '`.hgsigs`' file. Each line contains the full SHA1 of the signed changeset, the signature version and the base-64 encoded signature.

Chapter 5

Comparison of Distributed Version Control Systems

5.1 Testing Scenario

Shell scripts were used to create a repository for each system in a sandbox directory and to perform common operations, measuring runtime with the `time` command. Each test was executed 10 to 1000 times and logged to a file. The arithmetic average of the measured values is shown in the diagrams below. Statistical dispersion of the results was negligible. Output (both '`stdout`' and '`stderr`') was redirected to '`/dev/null/`' to eliminate additional costs of displaying the output in the terminal.

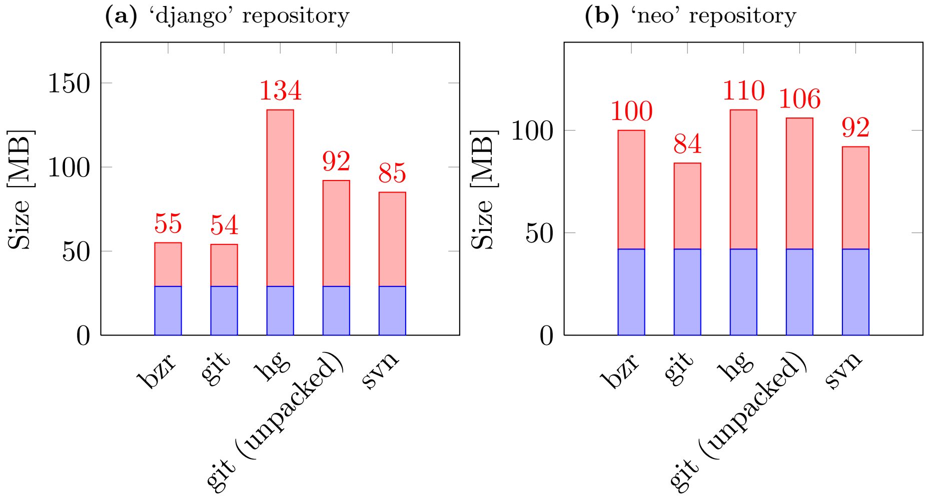
Tested operations include initialization of an empty repository, cloning of projects, adding files, committing files and displaying history logs for the project, a subdirectory and a single file.

5.2 Results

All tests were performed on a 32-bit Ubuntu 10.10 alpha 3 system with 4 GB RAM and a Core2Duo T9300@2.50GHz with the latest release of each version control system.

All figures use the name of each system's binary as labels (Bazaar: bzr, Mercurial: hg, Subversion: svn). Units are written using the SI-Prefixes, but are actually base 2.

Figure 5.1: Repository size for ‘django’ and ‘neo’ repository


(a) ‘django’ repository

(b) ‘neo’ repository

Repository Size

Repository size for each system was measured (using `du -sh` on the repository/branch directory). The size of git repositories was measured before and after running `git gc --aggressive`. `git gc` is used to compress loose objects into a packfile. Bazaar and Mercurial do not provide such a mechanism.

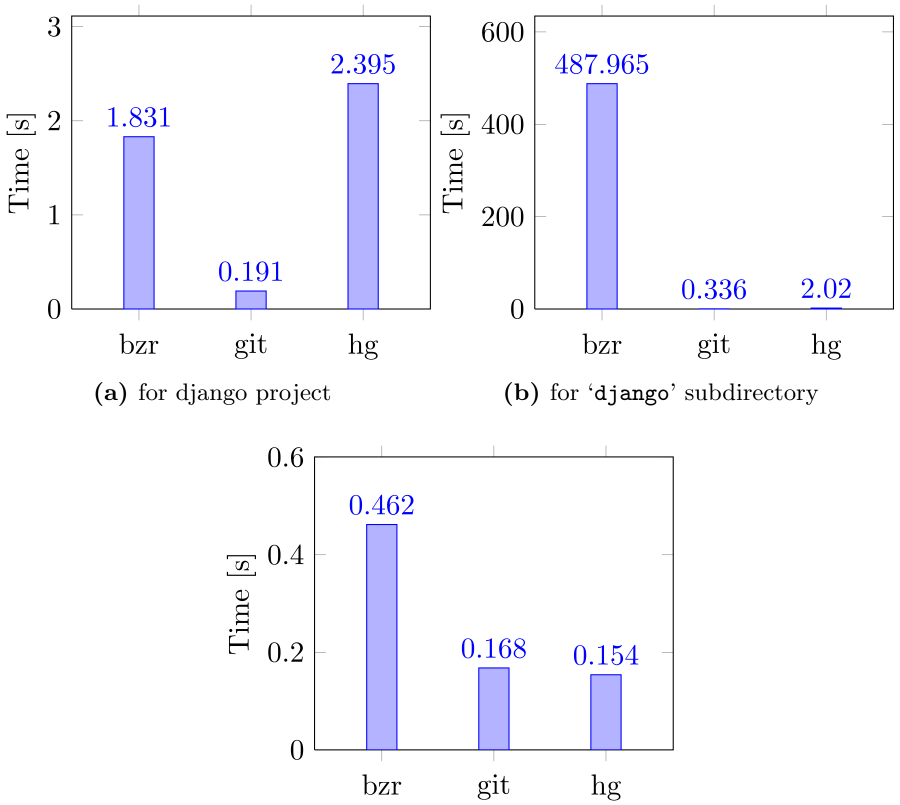
Disk consumption of Subversion was measured for a single checkout including metadata. The size of the checkout is comparable to the repositories of the distributed systems, even though it does not include the complete history of the projects. The working directory accounts for 42 MB in case of the ‘neo’ repository and for 29 MB for the ‘django’ repository. Plots of disk space consumption are shown in Figure 5.1.

Log

Time was measured to display the history log of all revisions for a project, a subdirectory of a project or a single file in a repository. The django project contains 8651 commits, of which 5569 commits touch files inside the ‘`django`’ subdirectory. The ‘`AUTHORS`’ file was modified 461 times in the history of the project.

Very noticeable is the great margin between Bazaar and the other systems when running `log` on a subdirectory. The fact that it takes Bazaar more than six minutes to display the log of a subdirectory is probably due to a flaw in its log implementation. The exact times of the different log operations can be seen in Figure 5.2.

Figure 5.2: Measured times to run `log` command



(a) for django project

(b) for ‘`django`’ subdirectory

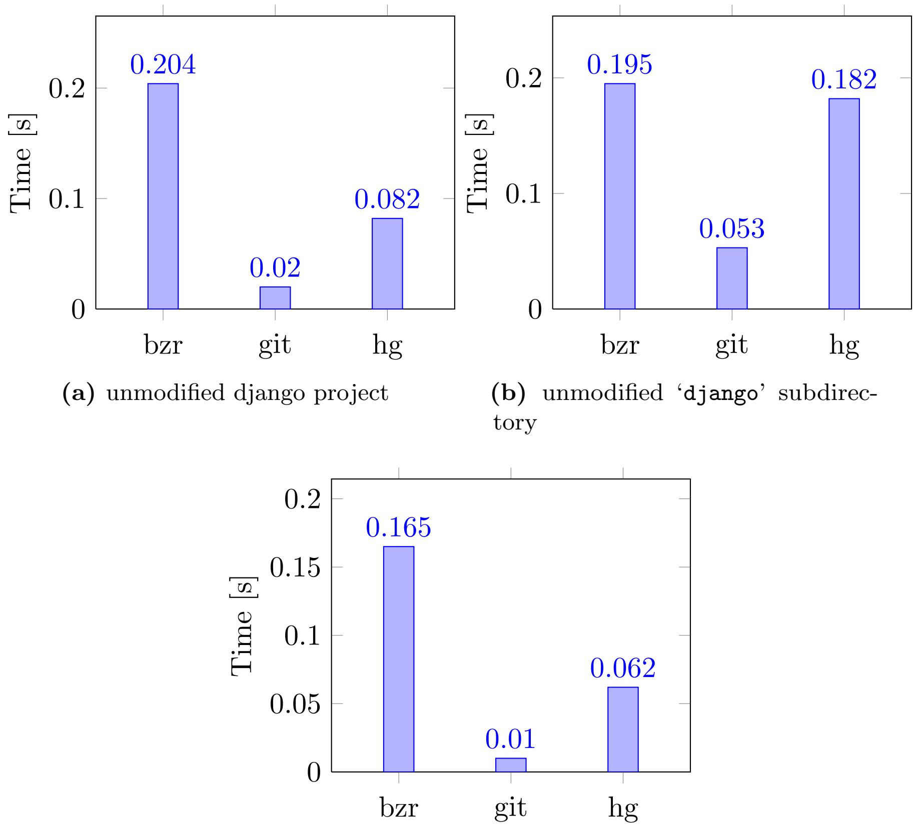
(c) for ‘`AUTHORS`’ file

Branching and Merging

Branching was tested with Git and Mercurial, because Bazaar does not provide the same concept of local branches. Creating branches was almost instant with both tested systems (0.04 seconds with mercurial and not measurable for git, i. e. 0 s)

Time to perform the first merge of a simple feature branch back into mainline (143 commits on mainline, 18 commits on the branch) was benchmarked using the ‘django’ repository. Again, these tests were only performed with Git and Mercurial, because Bazaar does not allow local branches. Git needed 0.065 s to merge the branch and Mercurial performed the same merge in 0.308 s.

Figure 5.3: Measured times to run `status` command



(a) unmodified django project

(b) unmodified ‘`django`’ subdirectory

(c) unmodified ‘`AUTHORS`’ file

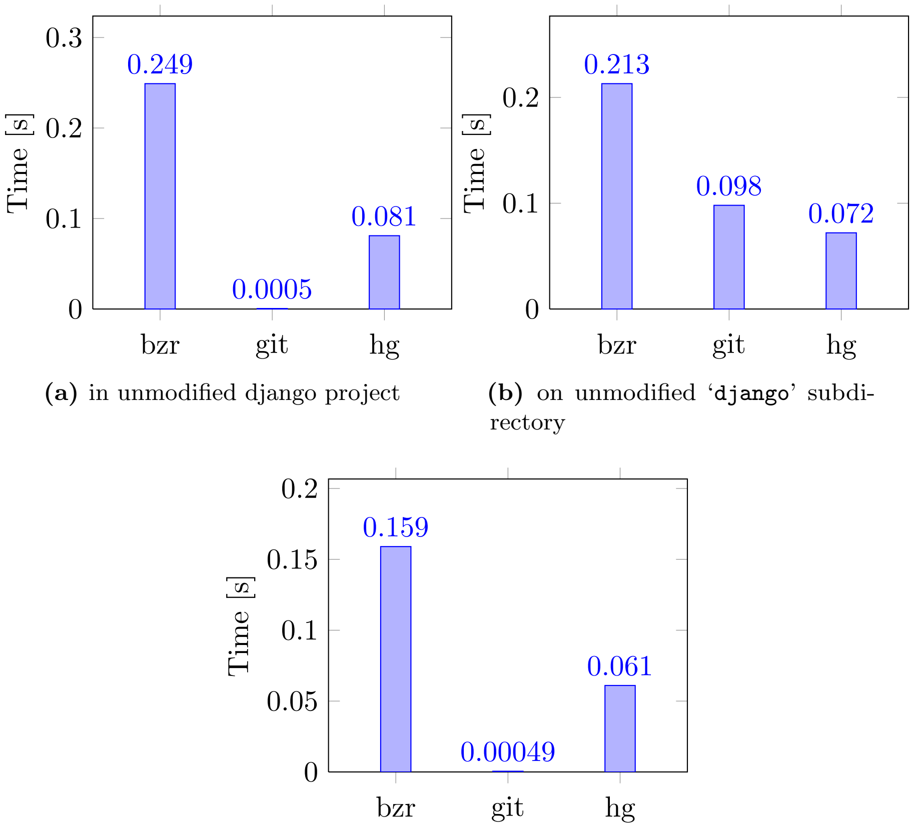
Status

Time to display status for a project, a subdirectory or a single file, was measured in an unmodified repository. There is no big difference between the analyzed systems. This command, which is one of the most used commands, runs in less than 0.2 s in all systems. The results can be seen in Figure 5.3.

Diff

The time it took the systems to display the diff in an unmodified working directory, for the whole project, a subdirectory and a single file are shown in Figure 5.4. Git is faster by a wide margin, especially for the whole project and single files, because it can compare the hashes of the objects with the hash from the repository.

Figure 5.4: Measured times to run `diff` command



(a) in unmodified django project

(b) on unmodified '`django`' subdirectory

(c) on unmodified '`AUTHORS`' file

Clone

The time it takes to create a full local clone from an existing repository. Cloning is necessary in Bazaar to create branches. Git and Mercurial use by default hardlinks when possible, to cut down on real disk space consumption. Times range from 33.7 s to 1.5 s and can be seen in Figure 5.5.

Add and Commit

Adding and committing files is one of the most commonly used operations in the daily use of version control. Multiple add operations usually proceed the commit operation. The following test added a single file with 1 MB and then created a commit.

Git follows a different philosophy when adding files and creating commits. Git immediately writes the file contents and the tree description to its database during add, while Bazaar and Mercurial only record metadata to

Figure 5.5: Creating a local clone of the django repository

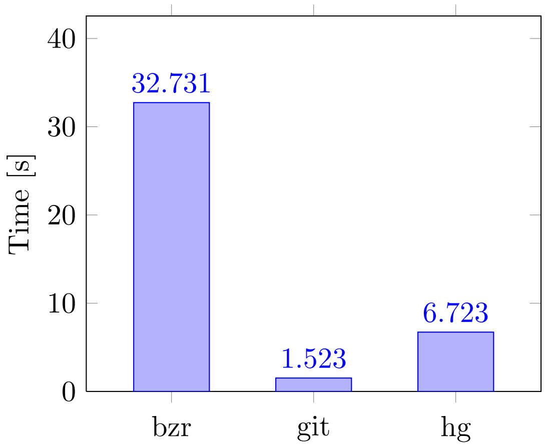
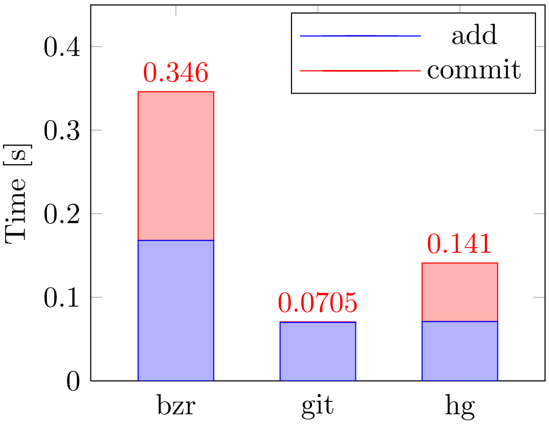


Figure 5.6: Adding and committing a 1 MB file



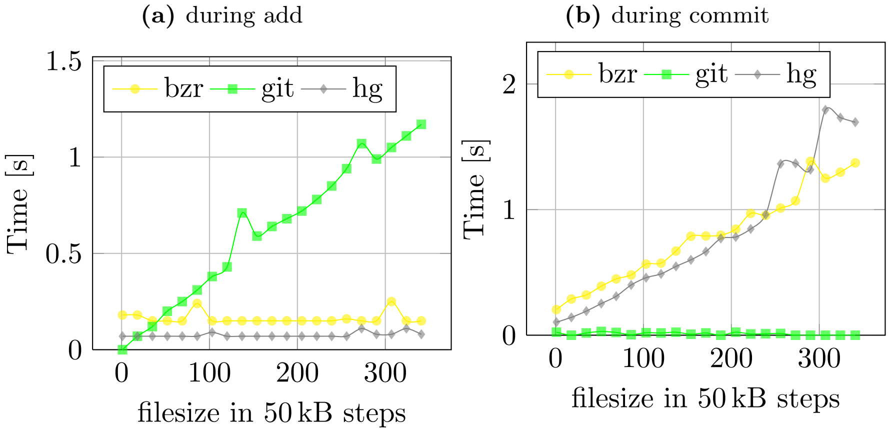
mark it for committing. Bazaar and Mercurial then write the file during commit, whereas Git only needs to write the commit object with a pointer to the tree description. The results are shown stacked in Figure 5.6.

This divergence can be seen more clearly in the following graphs which plot runtime depending on the size, respectively number, of files.

Filesize Dependency

Runtime of commit and add is shown in Figure 5.7, increasing the file of the committed file by 50 kB with each run. Git writes the object during the add phase, which is clearly visible in the plot. Bazaar and Mercurial add files of different sizes in constant time, as they only need to record metadata.

Figure 5.7: Filesize and time dependency



During the commit phase the opposite is the case. Git creates all commits in the same time, regardless of the commits size. Times of Bazaar and Mercurial increase proportionally with the size of the commit, although Mercurial shows a big increase after 230 steps (11.5 MB).

Treesize Dependency

The next tests created commits with an increasing number of files in the top level directory of the project. Each file was filled with 5 kB of random data. Bazaar shows a clear dependence on the number of files, whereas Git and Mercurial add up to 350 files in the same amount of time. Commit times exhibit the same pattern as add times. Git and Mercurial can process different tree sizes in constant time, while Bazaar's time goes up as the number of files increases. Both cases are shown in Figure 5.8.

Dependency on History Length

Distributed version control systems should handle projects with long history without problems. It is important, that the systems show little or no dependence on the length of history (number of commits). All three of the analyzed systems performed very well in this test and show no significant signs of degrading performance with growing repositories (tested for up to 25000 commits). The graphs in Figure 5.9 show the add and commit times for 500 commits.

Figure 5.8: Treesize and time dependency

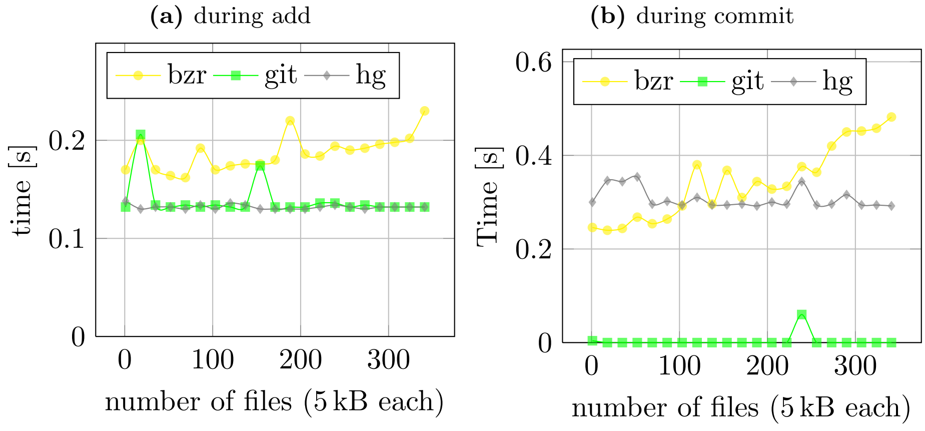
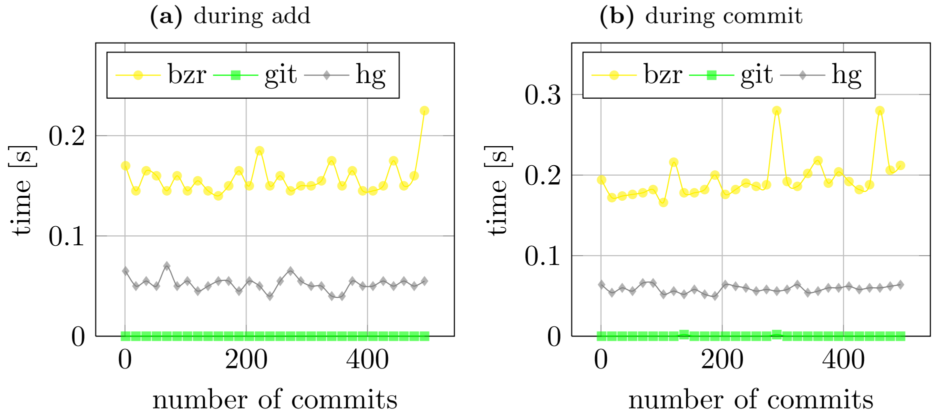


Figure 5.9: Time depending on number of history length



Chapter 6

Conclusion

Bazaar is the right choice, if a user does not want to completely switch to a distributed workflow, because it allows binding working copies to branches and directly committing to a common branch. Earlier repository formats of Bazaar were not very efficient and used a lot of disk space, but with the new *format-2a* format it often comes close to the size of packed git repositories. Bazaar is the only distributed version control discussed in this thesis, which is able to track empty directories.

Git and Mercurial use a history model which allows easy signing of a commit and all preceding history of that commit. This is as secure as the underlying hash function which is SHA1 for both Git and Mercurial. If SHA1 gets broken, these systems have to use a different repository format which will not be backward compatible – existing repositories would have to be converted to use the new hash function. Signatures in Bazaar do not take history into account and only sign the current state of the project.

Git is the clear winner when it comes to performance, which can be explained by its native implementation in C. While other systems were specifically designed to work on many platforms (using an interpreted language), Git first worked only on POSIX compliant systems, but recently ports for Microsoft Windows emerged[1].

Git had unique features, of which most are now available as plugins for other systems (rebasing, cherry-picking).

Distributed version control systems often perform worse than centralized version control systems when versioning (big) binary files, because the complete history of each file has to be transmitted; furthermore, binary files often exhibit bad delta compression behavior. Bazaar's bound branches and centralized workflow model provide an alternative.

All systems allow import or export from other systems, so most repositories can be converted without problems. Bazaar, Git and Mercurial provide

[1] MSYSgit, git extensions and TortoiseGit to name a few

Subversion wrappers or importers which allow read and write access to subversion repositories.

Bazaar allows lossless conversion from other formats due to its repository format: Commit and file ids can be set arbitrarily and can often use the original commit ids (or revision numbers when working with Subversion).

Bibliography

[Bang-Jensen and Gutin, 2008] Bang-Jensen, J. and Gutin, G. (2008). *Digraphs: Theory, algorithms and applications.* Springer Verlag.

[Baudiš, 2009] Baudiš, P. (2009). Current Concepts in Version Control Systems. Bachelor's thesis, Prague, Czech Republic.

[Canonical Ltd., 2005] Canonical Ltd. (since 2005). *Bazaar source code (https://code.launchpad.net/~bzr-pqm/bzr/bzr.dev).* Canonical Ltd.

[Chacon, 2009] Chacon, S. (2009). *Pro Git.* Apress, 1st edition.

[Fogel, 2005] Fogel, K. (2005). *Producing Open Source Software: How to Run a Successful Free Software Project.* O'Reilly Media, Inc.

[Hunt and McIlroy, 1976] Hunt, J. W. and McIlroy, M. D. (1976). An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ.

[Loeliger, 2009] Loeliger, J. (2009). *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development.* O'Reilly.

[Mackall, 2006] Mackall, M. (2006). Towards a better SCM: Revlogs and Mercurial. In *Proceedings of the Linux Symposium*, pages 91–98, Ottawa, Ontario, Canada. Citeseer.

[O'Sullivan, 2009] O'Sullivan, B. (2009). *Mercurial: The Definitive Guide.* O'Reilly.

[Pilato et al., 2004] Pilato, C. M., Collins-Sussmann, B., and Fitzpatrick, B. W. (2004). *Version Control with Subversion.* O'Reilly.

[Schneier, 1996] Schneier, B. (1996). *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc.

[Swicegood, 2008] Swicegood, T. (2008). *Pragmatic Version Control Using Git.* Pragmatic Bookshelf.

[Vesperman, 2003] Vesperman, J. (2003). *Essential CVS.* O'Reilly.

List of Figures

Listings