Final thesis

Graphical User Interfaces for Distributed Version Control Systems

by

Kim Nilsson

LIU-IDA/LITH-EX-A–08/057–SE

2008–12–05
Final thesis

Graphical User Interfaces for Distributed Version Control Systems

by

Kim Nilsson

LIU-IDA/LITH-EX-A–08/057–SE

2008–12–05

Supervisor: Anders Höckersten, Opera Software
Examiner: Henrik Eriksson, MDA
Abstract

Version control is an important tool for safekeeping of data and collaboration between colleagues. These days, new distributed version control systems are growing increasingly popular as successors to centralized systems like CVS and Subversion. Graphical user interfaces (GUIs) make it easier to interact with version control systems, but GUIs for distributed systems are still few and less mature than those available for centralized systems.

The purpose of this thesis was to propose specific GUI ideas to make distributed systems more accessible. To accomplish this, existing version control systems and GUIs were examined. A usage survey was conducted with 20 participants consisting of software engineers. Participants were asked to score various aspects of version control systems according to usage frequency and usage difficulty. These scores were combined into an index of each aspect’s “unusability” and thus its need of improvement.

The primary problems identified were committing, inspecting the working set, inspecting history and synchronizing. In response, a commit helper, a repository visualizer and a favorite repositories list were proposed, along with several smaller suggestions. These proposals should constitute a good starting point for developing GUIs for distributed version control systems.
Dedicated to my grandfather and mentor, who could not be with us long enough to see me finish my studies.
## Contents

1 Introduction 1
  1.1 Purpose .................................................. 2
  1.2 Method .................................................. 2
  1.3 Structure ................................................ 2
  1.4 Scope .................................................... 3

2 Background 5
  2.1 Version control explained ................................. 5
    2.1.1 Brief history ........................................ 6
    2.1.2 Checking out and committing ........................ 7
    2.1.3 Deltas ............................................... 7
    2.1.4 Branching ............................................ 10
    2.1.5 Merging .............................................. 12
    2.1.6 Conflicts ............................................ 14
    2.1.7 Recent developments ................................ 16
  2.2 Distributed version control .............................. 17
    2.2.1 Defining features of distributed systems .......... 18
    2.2.2 Other features of modern version control ......... 19
    2.2.3 Drawbacks of distributed version control .......... 22
    2.2.4 Existing distributed systems ....................... 23

3 User interface design 27
  3.1 General GUI design guidelines .......................... 27
    3.1.1 Conforming to user expectations .................... 28
    3.1.2 Providing smooth operation ........................ 29
    3.1.3 Providing feedback ................................ 30
    3.1.4 Encouraging exploration ............................ 31
  3.2 User interfaces for centralized systems ............... 31
    3.2.1 Prominent GUI features ............................. 31
    3.2.2 Examples of GUI frontends .......................... 34
    3.2.3 Web interfaces ..................................... 38
  3.3 User interfaces for distributed systems ................ 39
Chapter 1

Introduction

Almost all software projects of sufficient size and involving multiple developers today tend to utilize some form of version control system (VCS), also known as a revision control system (RCS). The purpose of a version control system is to act as a repository for the project data along with a complete development history, enabling any state of the project to be recreated at a later point. The VCS also helps synchronize the work of multiple developers, transparently allowing people to work on different parts of the project simultaneously.

Currently most version control systems are centralized, with a single server as the common hub of all developers’ clients. The server stores the single copy of the entire development history while clients check out a single state from the server, modify the data and then commit the result back to the server.

A newer variant of version control is a distributed version control system (DVCS). Distributed systems differ from centralized systems in various ways, but most prominently in that every client stores the complete development history along with the current state. When committing changes to the current state, the local development history is updated. In effect each client is running its own version control server and keeping its own repository. Repositories can then be synchronized, allowing changes made to one repository to be applied to another. Whereas centralized systems are always hub–based client–server networks, distributed systems are peer–to–peer based and have no distinction between client and server. They therefore do not have to adhere to any specific design when it comes to data flow.

With the additional freedom of choice granted by distributed version control systems, new types of work flow become possible. However, this also risks making distributed systems more difficult to learn for new users. For many users, well-designed graphical user interfaces provide a quicker and more intuitive means of learning and working with systems, but distributed systems are still relatively new and few graphical frontends are available.
By comparison, centralized systems have been around for much longer and have many available frontends as well as tools for integration with virtually any development environment.

1.1 Purpose

The purpose of this thesis is to propose graphical user interface techniques appropriate for making the features of distributed version control systems accessible to users. A DVCS can require new ways of thinking compared to centralized systems, therefore it is likely that a graphical user interface for such a system would benefit from corresponding new kinds of visualization and interaction. This thesis aims to investigate and define user interface concepts that fulfill this requirement, in order to serve as a brief guide to developers wishing to develop graphical frontends.

1.2 Method

The primary input for this thesis comes from analyzing existing version control systems, graphical frontends and usage patterns. The latter is largely based on a small survey conducted at Opera Software, aimed at discovering the areas of user interaction with the greatest need for simplification and/or visual enrichment through graphical user interfaces. Based on the survey results, a number of graphical user interface ideas are proposed that may help alleviate the identified problems. A brief evaluation of these proposals’ effectiveness is also carried out by analyzing them with regards to the original problems and established user interface design guidelines.

1.3 Structure

The report is divided into five parts, identifiable as chapters 2 through 6. Chapter 2 summarizes the background, functionality and history of version control. Chapter 3 describes the basics of graphical user interface design and some of the existing graphical frontends for version control systems. Chapter 4 details how the usage survey was carried out, its results, and analysis of those results.

The remaining parts account for the creative portion of this thesis. Chapter 5 interprets the results of the survey and condenses them into more specific requirements for a frontend, and then proposes specific features that fulfill those requirements. Finally, chapter 6 discusses the proposed features in relation to the original problem to evaluate if they are sound. The chapter concludes by discussing future work in this area.
1.4 Scope

This thesis is not meant to be a comprehensive guide to switching from centralized to distributed version control. It is not intended to extensively document the inner workings of these systems, nor to serve as a guide for learning how to use them. Further, it does not strive to produce a complete graphical frontend, only to evaluate ideas for visualizations and interactions for the most common tasks.

As an additional limitation, the thesis will only examine freely available version control systems and frontends (such as open source projects). Commercial systems, while certainly enjoying wide usage in corporate environments, are due to their nature less available to the general public. Also, they often include their own graphical user interface as part of the product. Therefore the need for usable graphical frontends is probably greater in the open source community since these systems are more readily available to novice users, hopefully justifying the focus of this thesis.

Finally, since there is already an abundance of graphical user interfaces available for centralized systems, the thesis will focus mainly on tasks where distributed systems differ from centralized systems.
Chapter 2

Background

This chapter will briefly introduce you to the history and mechanics of version control. While the text is written with a certain bias toward software development, the concepts discussed are equally applicable to any kind of project that involve editing text.

2.1 Version control explained

When creating and editing large amounts of text, such as source code, the need often arises to recreate an earlier stage of the project. The perhaps most prevalent example of this would be the “undo” functionality that is common in nearly all modern text editors. When the author makes a mistake, he can revert the document to the way it looked prior to his last change. However, undo functionality is typically limited to very recent changes only, such as those made during the current editing session. Version control instead saves the entire development history of the project, enabling the user to recreate any revision at any time.

By being aware of earlier revisions, version control also makes it possible for several people to work on the same file. Each of their changes can be recorded in the history, and in many cases merged automatically – the system determines how all of the changes can be included, even though they were made independently from each other.

Note that certain modern word processors such as Microsoft Word use document formats that can contain the editing history as well, allowing more advanced features such as showing/highlighting added and removed text in the program (typically for change review purposes). This is also a form of version control, since the historical states of the document are retained and can be reviewed at a later time.

For the rest of this report, the words revision and version will be used interchangeably.
2.1.1 Brief history

The first widely recognized version control system was the Source Code Control System (SCCS), developed by Marc J. Rochkind in the early 1970s. Rochkind identifies four important features of his system: (Rochkind, 1975, p. 364)

- **Storage:** The version control system improves upon simply saving each revision of the file separately by storing the data of all revisions collectively. Data that is unchanged between versions does not need to be duplicated, significantly reducing the space required.

- **Protection:** Access to the version controlled files is only possible through the system itself, enabling security policies to be implemented. Rochkind gives as examples situations where developers could be given access to only specific files and/or specific versions of those files.

- **Identification:** Each revision is uniquely identified by its version number, enabling simple retrieval of any revision as long as this number is known. Rochkind meant for SCCS to be tied into the build process of software projects, enabling these version numbers to be embedded into the final software for easy identification later.

- **Documentation:** Along with the file content itself, metadata such as author, time and comments are included with each revision. For example, the rationale for a given change can be found at any time.

SCCS was in use for several years, but limitations in its performance and its relatively simple design for handling multiple editors eventually led to alternative version control systems being developed. The first notable example of a “successor” to SCCS was Revision Control System (RCS), conceived by Walter Tichy in 1985.

RCS improved on the performance of SCCS and further developed ideas such as branching (explained below). However, it remained rather low-level and lacked features required for large scale collaborative development. Partially due to these reasons, Concurrent Versions System (CVS) was developed, initially as a set of wrapper scripts around the low-level RCS commands but later rewritten as a version control system in its own right, though still utilizing most of the same underlying concepts from RCS. (Berliner, 1990; Tichy, 1985)

CVS abstracted away some of RCS’s rough edges and also introduced features for managing entire sets of files and directories, a large improvement over RCS and SCCS which operated on individual files only. Despite its nature of mostly using RCS to do its work, CVS can be said to have made many of the concepts described in this chapter usable by a wider audience.

Today, CVS is still widely used and can be regarded as one of the most successful version control systems ever. During the years of usage certain
flaws have surfaced however (touched upon later in this chapter), leading to the development of successors like Subversion and eventually fully distributed version control systems.

2.1.2 Checking out and committing

With version control systems, the entire history of every file is stored in a location referred to as the repository. With centralized systems the repository is stored on the server. In order to work on the version controlled files, the user first checks out the latest version from the repository onto his own computer where a working copy is created.

The working copy consists of normal files based on a single version (typically the latest) fetched from the repository. The user modifies his working copy whereupon the updated files can be committed back to the repository where they are stored as new versions of the existing files. After committing, the new versions are made available to other users through the server.

2.1.3 Deltas

Instead of storing each individual revision in the repository, the history and content of a version controlled file is typically described through a series of deltas, where each delta represents a change to the file. The change may consist of adding lines, removing lines or both, but other kinds of changes are also supported in newer systems. The history starts with an empty file, onto which deltas are applied in order. Each point between deltas is a revision and typically has its own identifying version number. To retrieve any revision of the file, the system applies the corresponding number of deltas, with the combined result of all deltas being the newest (current) version.

The most straightforward way to represent a delta for text files is as a list of all lines added or removed (a modified line can be represented as removing the existing line and adding a new one). There are various formats for describing this, such as the ones supported by diff, a command–line tool that compares two files and outputs the differences between the two. A diff, short for difference, has become another common word for a delta (another word is patch). An example of the default output format of diff is shown in Figure 2.1.

In this case, lines to be removed are prefixed by “<” while lines to be added are prefixed by “>”. The diff result should be interpreted as follows:

- Line 4 in the first file (‘cheese’) changes into line 4 in the second file (‘pineapples’).
- Line 6 in the first file (‘sausages’) changes into lines 6 through 7 in the second file (‘beef’ and ‘yoghurt’).
At first the diff output format might seem overly verbose for the purpose of deltas. For example, when using a delta to patch one revision of a file into the next revision (patching is the process of applying a diff/delta/patch to the source file to transform it into the destination file), there is no need to know the content of lines that are to be deleted in the earlier file. However, the additional information makes the patch symmetric and reversible; the delta contains enough information to also undo the operation, transforming the destination file into the source file. The patch operation and the inverse patch operation is illustrated in Figure 2.2.

Transforming a diff into its inverse is a simple operation, in which essentially all references to and content from the two files are swapped. This property makes it easy to patch version controlled files to both future and past revisions.

Alternative methods

A file can be represented entirely as a series of deltas, with an empty file as a starting point. While this is an effective way to store the history of the file using as little space as possible it means that retrieving newer revisions can require applying a large number of deltas, resulting in a large performance impact. To improve this performance, version control systems can store complete copies of the files at points between certain revisions. A typical space versus time efficiency tradeoff, this increases the space required to store the history of the file but reduces the number of deltas that have to be applied to reach a given revision.

Recognizing that the most common revision to be retrieved is typically the most recent one, RCS uses a scheme where the latest revision is stored as a complete copy, so that older revisions can be retrieved by applying the reverse deltas. (Tichy, 1985)

Not all version control systems rely on storing deltas to manage the history of a file. The preferred method used by SCCS is a technique Rochkind refers to as “The Weave”. With this method, the entire history of a file is contained in a single file with all the revisions “woven” together. Conceptually, the weave contains all the lines of text that ever existed in the file, and with each line information is stored to describe in which particular revision that line exists. This way, any revision can be recreated by scanning the weave a single time. (Rochkind, 1975, p. 367–368)

The time required to recreate any revision grows with the number of total revisions stored in the history, due to the growing complexity of the weave file. This differs slightly from deltas where the access time grows with the distance from the nearest reference point (the empty file at the root, or a stored copy in between revisions).
2.1. VERSION CONTROL EXPLAINED

Figure 2.1: An example of the `diff` tool’s output

Figure 2.2: Patching and un-patching files
2.1.4 Branching

If you only treat the version control system like an undo feature, the resulting editing history is always a single straight line of deltas (and hence revisions). However, there are plenty of cases where multiple “histories” are preferable. We can imagine one such use case in the context of our undo example. After having undone a set of changes, the user will typically want to add a new change instead of the ones just reverted. But how do you represent this in the development history? Several options spring to mind:

1. Remove all the previous changes from the development history, and add the new change instead. This is not preferable since the history is irrevocably destroyed, completely contrary to the intentions behind using version control to begin with.

2. Revert all previous changes by adding their reverse deltas along with the new changes to the development history. This is a better solution since it preserves the history and accurately represents the user’s changes on the file, including the undo operations.

Using the second method, the user can undo his first set of changes and make a new set of changes on top of that. However, what if the user changes his mind again and wants to try with the first set of changes once more? He would then need to undo the new changes and add the original changes yet again, producing a lot of duplicated data in the history of the file. An alternative solution is:

3. Branch the history at the point just before the first changes were applied, and add the new changes in a new parallel history line. The new branch shares all history with the original up to the branch point, then diverges and contains the new changes.

With this approach, called branching, the history of a file is a tree rather than a single line. Each revision still has exactly one parent revision (except for the initial revision), but can have one or more child revisions. This difference between branching and non-branching histories is illustrated in Figure 2.3.

The advantage of branching is that multiple variations of a file can coexist in parallel, and the user can continue working on and adding changes to either variation.

The original branch’s revisions are numbered 1.x, signifying “the xth revision along the first branch from the root”. The second branch begins at revision 1.2 and is numbered 1.2.1.x, signifying “the xth revision along the first branch from revision 1.2” (or generally, each group of two numbers x.y read as “xth branch, yth revision”). This is the revision number scheme used by RCS and subsequently CVS, and it allows the user to create as many branches as he wishes from any given revision without encountering version number clashes. (Tichy, 1985, p. 5)
Even SCCS, the very first version control system, had a notion of branching, which Rochkind suggested as a solution for handling the stabilization phase of software projects. As a software project entered the final phase before a product release, a new “release” of the source code would be created from its current revision, its version numbers restarting with a new release number (e.g. after revision 1.x would follow revision 2.1 as a new release).

This kind of “plateau-based” branching was rather limited compared to the branching described above, but it was suitable for Rochkind’s software development use case. The old release branch could be used for stabilization and testing, while main development for the next version could continue uninterrupted on the new release branch. (Rochkind, 1975, p. 365)

Managing branched files

Remembering a branch solely by its branch number can be cumbersome, especially if your development process involves a lot of branches. Furthermore, the branching described above only concerns single files (since both SCCS and RCS operate on single files), so if your project consists of multiple files you would in the worst case have to remember the individual branch numbers for each file.

This problem was solved early on with the concept of named branches, a feature that gained widespread use with the introduction of CVS. This enables the user to have easy to remember branch names that represent a given branch for all files involved, abstracting away the specific branch number used for each file.

Tags

Named branches, as the name suggests, are aliases for branch numbers of a file or a set of files. CVS also introduced the concept of tags, which are aliases for a specific revision of a file (or set of files). Tags can refer to revisions on any branch have various uses, but the typical use case would be to identify a specific state of the entire repository as it looked at some
point in time (e.g. a software compilation), accomplished by tagging all files at once.

Some version control systems like CVS also allows tags to be moved after creating them, so you could have a floating current_release tag that would always point to the revisions of the last product release (being moved as necessary). However, other systems that disallow moving tags argue that modifying the history (i.e. redefining what current_release means) violates the integrity of the repository since checking out files with a given tag is not guaranteed to yield the same result every time. If one uses tags to identify a specific state of the repository and the tag is accidentally moved, there might no longer be any way to identify the original revisions since the tags themselves are not version-controlled and thus operations on them cannot be undone.

Because of this, it can be argued that tags themselves should be immutable once created, and that in our example one should instead create a new tag for each release such as release_1, release_2 etc. Both mutable and immutable implementations of tags exist in current version control systems. Subversion takes a different approach and implements tags as branches — these are mutable in the sense that you can commit to them, but immutable in the sense that they cannot be moved to some other part of the revision tree (Collins-Sussman et al., 2008, p. 110).

2.1.5 Merging

While branches can be useful when you need to separate a project into multiple variants, or to keep track of different stages of a project, in many cases there is a need to bring the diverged branches together again into a single version containing all changes from both branches. For example, multiple developers might work in parallel on separate branches so as to not disturb each other, and once their work is complete all the changes of their “work branch” should be carried over to the original branch.

The process of joining two branches together is called merging. After merging two branches, the resulting commit will contain all changes of both branches. This is conceptually illustrated in Figure 2.4. Neither SCCS, RCS or CVS has any particularly strong built-in support for merging, so for a long time merging has been a troublesome process involving a lot of manual editing and conflict resolution.

The most common form of merging does not actually involve any real branches at all; it is the process of keeping the user’s checked out working copy up to date with the repository. The user’s working copy is, for all intents and purposes, a small temporary work branch (though since it is not version controlled in itself, it always translates into a single delta when committed). If another user commits new changes to the repository, it becomes necessary to update the current working copy to contain the latest changes as well. Combining these changes with any changes the user has also
2.1. VERSION CONTROL EXPLAINED

Figure 2.4: Merging branches in a modern repository tree

made to his working copy (but not yet committed), is a merge operation.

Transforming deltas

Including changes from a different branch means applying deltas onto a file that most likely differs from the file for which the delta was originally constructed. This tends to yield incorrect results and corrupted files, hence we need some way of “translating” a delta to work for another file. Typically this would be a difficult task to solve automatically, but in a version controlled environment we know exactly which deltas each version of a file is comprised of, and hence we also know by which deltas two versions of the file differ. Expressed differently, since we know how the entire revision tree it is possible to walk through it from one revision to another, taking note of the nodes (deltas) encountered along the way.

The first step of applying a delta created for one branch onto another branch is to find some common ground, which in this case means the branch point. This is the revision at which the two branches diverged, and hence the last common revision of the two files. The delta now needs to be transformed so it can be applied to that common revision, which is done by compensating for each change introduced by all the revisions added afterwards. For example, if one of the deltas added 20 lines of text above the changes in the delta we’re trying to merge, the line number references in our delta need to be adjusted by $-20$ in order to accomplish the same change on a file that never had the other delta applied to it.

The developers of distributed version control system darcs recognized that working with deltas is at the very core of version control, and therefore strived to create a notation and algebra for working with them in a clearly defined manner. By this “patch algebra” the adjustment described above is a case of commutation; that is, modifying two deltas $AB$ so that applying
them in the reverse order $B'A'$ yields the same final result. Working with deltas in this manner is illustrated in Table 2.1. (Understanding darcs/Patch theory, 2008)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential application</td>
<td>$AB$</td>
</tr>
<tr>
<td>Commutation</td>
<td>$AB \leftrightarrow B'A'$</td>
</tr>
<tr>
<td>Inverse</td>
<td>$ABB^{-1} \leftrightarrow A$</td>
</tr>
<tr>
<td>Inverting multiple patches</td>
<td>$(AB)^{-1} \leftrightarrow B^{-1}A^{-1}$</td>
</tr>
</tbody>
</table>

Table 2.1: Delta operations expressed in a patch algebra

After the delta has been adjusted to work for the branch point revision, we can begin compensating for all the changes in the other branch one by one, but this time in the other direction. Finally we will arrive at the last revision, and we will have a delta that can be applied on top of it.

Doing this kind of merge with only a single delta from a foreign branch is a special case of merging and is usually referred to as cherry-picking, a term derived from “picking cherries from branches” as a way of picking out merely the good bits. To fully merge the two branches, we will repeat the same procedure for every delta of the branch. There are many different strategies for in how to apply the deltas, each with their strengths and weaknesses for various corner cases. (Revctrl Wiki, 2008, various articles)

Older version control systems such as CVS have support for merging arbitrary changes from any branch onto any other branch, but does not keep track of this operation as a merge as such. There is no information recorded in the history of the file to indicate that the two branches merged to form a single branch again, rather all the adjusted deltas of the branch to be merged are combined into a single new delta that is committed onto the destination branch. Newer version control systems keep track of merges and are able to utilize that information to simplify future delta transformations.

### 2.1.6 Conflicts

Merging is not guaranteed to succeed, regardless of the method used. When merging two branches that have modified the same area of a file in two distinct ways, which changes should be included in the merged file? If both changes should be included, which one should be included first? These questions cannot be answered by the version control system, and the result is a conflict. Figure 2.5 illustrates a conflict in the previous shopping list example.

There are different strategies to try to avoid merge conflicts, with varying properties. For example, many merge algorithms attempt to recognize if a certain change is identical to one already applied to the other branch, in which case the change does not need to be merged.

If the version control system cannot find a conflict-free way to merge the two branches, the merge operation has failed. In order to resolve the
situation, the user needs to inspect the conflicting areas in the file and manually choose which changes should take preference. If this is not enough to produce a semantically correct result, the user must manually edit the text in order to properly include the conflicting changes.

**Other merge errors**

Even when there are no conflicts, merging two branches by automatically applying the adjusted deltas is no guarantee that the resulting content will be syntactically or semantically correct. The version control system has little to no knowledge about the internal structure of files and only operates on lines of text that can either be included or removed. This behavior may not always be appropriate in all contexts, however.

For instance, imagine that the shopping list example from earlier was split into two branches and *cabbage* was added to both branches but in different locations in the list. If the two branches were merged, *cabbage* would appear at two locations in the list without conflict errors, even though it makes little sense in the context of a shopping list.

A similar issue could also occur if two editors, as in the example above, both added *cabbage* to the list in the *same place* (see Figure 2.6). The merge algorithm may recognize these two changes as identical and merge them into a single instance of *cabbage*, but has no way of knowing if the correct shopping list should contain one or two *cabbages*.

These secondary (‘soft’) errors always stem from the fact that two changes can be made independently of each other but require modification in order to work together in the same file. Since they are not errors in the merge operation itself, a version control system can’t detect them and doesn’t concern itself with them at all. Therefore, soft errors occasionally sneak in during
merging without warning and so the user should be prepared to encounter them and deal with them.

2.1.7 Recent developments

Before moving on to describing distributed version control, it is worth mentioning some of the weaknesses of older version control systems (up to and including CVS), and how these can be addressed without straying from the concept of centralized version control. The most notable development in this area is Subversion, a project started in 2000 with the explicit goal of replacing CVS but keeping many of its basic ideas so as to make Subversion feel like a natural successor to users (Collins-Sussman et al., 2008, pp. xx–xxi).

Subversion is designed with many of the problems and weaknesses of CVS in mind, so some of its improvements are:

- Directories are properly version-controlled. Since CVS is built upon RCS, which handles single files only, the storage of directories in CVS repositories has numerous quirks.

- All revision numbers are incremented on each commit. CVS increments revision numbers on only those files actually touched by a change, making it much harder to know which revisions “belong” together. When all revision numbers increment in sync, the need for and complexity of tags is greatly reduced.

- Improved handling of binary files. Since binary files cannot be handled on a line-by-line basis, CVS typically stores the entire file for each revision. Subversion includes delta handling for binary files as well.
2.2 DISTRIBUTED VERSION CONTROL

Subversion includes several of the modern features present in distributed version control systems, and can as such be thought of as a stepping stone into the world of distributed version control. A more prominent example of this in–between niche however is the SVK program.

SVK is technically a distributed version control system, though it uses a hierarchical network design that makes it closely related to centralized systems. It is based on Subversion’s repository format but provides additional features that can be expected to be found in distributed systems, such as being able to work offline without needing a server. It also includes more advanced algorithms for merging, though these will not be touched upon here.

Several other features that can be found in modern version control systems are described in closed detail in section 2.2.2.

2.2 Distributed version control

All the systems described so far have had in common that they typically operate against a single common point of storage, with all users carrying out changes through that shared nexus. This is more commonly known as the client–server model, an example of a star–shaped logical network topology (see Figure 2.7).

In later years a different kind of version control system has evolved which is not limited to a client–server topology, but can operate in a peer–to–peer fashion. Much like other peer–to–peer systems like the BitTorrent file sharing protocol or the Skype internet telephony application, a distributed version control system consists of multiple independent clients which can interconnect with each other in an ad hoc fashion to exchange information. ([BitTorrent Introduction, 2008](#))

![Figure 2.7: Different kinds of network topologies](#)

(a) Server–client topology

(b) Peer–to–peer topology

The client–server model as used by centralized version control systems consists of relatively thin clients and a single processing–intensive server.
CHAPTER 2. BACKGROUND

This allows for reduced complexity, since most complex tasks are carried out in a single place in a synchronized manner and the clients are kept relatively minimalistic. However, the low redundancy afforded by this model can be a cause of problems, some of which will be explained in the following subsection.

Distributed systems instead prioritize the equality of peers, including enough functionality for the peer to fulfill the functions of both client and server. Through this design decision there is generally a greater redundancy across the network. (Milojicic et al., 2002)

2.2.1 Defining features of distributed systems

As explained above, the difference between distributed and centralized systems is in the network topology design (the manner in which information is exchanged between users) and the corresponding amount of version control work that is done locally versus remotely. The features described in this section directly relate to these defining qualities.

Note that many of the features of distributed version control systems were born from perceived limitations in existing centralized systems such as CVS. Therefore, some of the features described in this and the following section will be explained in terms of previous limitations and their proposed solutions in the new systems.

Self–managed repository

One limitation of centralized version control is that you can typically only commit changes to a file while connected to the central server. At first glance this might not seem to be a serious limitation. After all, all of your changes are still intact in your currently checked out copy until you can connect to the server and commit them. However, if you spend a lot of time working on your copy without committing changes to the server, all your changes will accumulate and produce a single large delta to be committed, with only a single set of metadata (e.g. timestamp and commit message). If the history of the file needs to be examined later, it will be harder to pinpoint exactly when a given change was introduced since the history will not be as detailed as one would prefer.

With distributed systems there is no single server, so all clients carry all the logic necessary to themselves handle a repository. Typically, instead of checking out a copy of a certain state of a project from a central repository, you instead keep the entire repository itself, complete with the entire history and thus all recorded states of each file. If you keep multiple copies of the project on your computer, you will have multiple complete repositories.

By keeping the entire repository on your computer, you no longer need to be connected to a server in order to commit changes (and thus provide more granular history metadata), solving the above-mentioned problem with centralized version control systems. This also removes an earlier performance
2.2. DISTRIBUTED VERSION CONTROL

bottleneck, since all operations except synchronization between repositories no longer require data to travel across the network.

Synchronizing with remote repositories

Instead of the request/submit procedure of centralized systems, distributed systems allow any peer to exchange patches with any other peer. In a software development environment, one can imagine developers exchanging experimental patches amongst themselves for testing purposes before the final result is pushed to the build computer’s repository. This is in stark contrast to traditional centralized workflows, where only the data stored at the central server is version–controlled and anything else needs to be done manually.

The checkout of centralized version control is replaced by cloning the remote repository, which as the name suggests copies all the data from the remote repository and sets up an identical copy on your computer. The user can then edit his files and commit to the local repository clone as much as he likes. Later, all changes can be pushed back to the original repository (or any repository cloned from it).

The user usually also wants to pull changes from the original repository at regular intervals. This lets the user update his own repository clone with any additional changes that have been added to the source repository since he cloned it.

While the user can push to or pull from any remote repository, the version control system typically remembers which repository it was cloned from and treats it as the default repository for pushing and pulling, so the user does not need to type its name/location every time.

Replacing a central intelligent server and “dumb” clients with a community of equal and self–managed peer repositories in this fashion is the single most distinguishing quality of distributed version control systems.

2.2.2 Other features of modern version control

Not all features commonly found in distributed revision control are exclusive to distributed systems, but should rather be thought of as improvements to the concept of version control itself. As such, while the following features are fairly common among the examined distributed systems, they could just as well be implemented in any modern version control system.

Hash identifiers

As described in the section about Subversion earlier (page 16), having individually incrementing version numbers for each file means additional complexity for keeping track of repository states (e.g. tags). Subversion solves this by having all version numbers increment for each commit, keeping all
numbers in synch at all times but at the expense of having its identifiers more quickly grow long and complex.

Many distributed version control systems instead identify revisions by a *hash value*, also known as a *digest*. A hash is a value designed to represent some other “source” value from a larger domain as uniquely as possible, usually by putting the larger value through some algorithm (a hash function) to condense it into a hash value. The hash value can be thought of as the “fingerprint” of the original value. (Preneel et al., 1993)

Since there are more values than hash values there will inevitably be duplicates — multiple source values with the same hash value. A good hashing algorithm therefore takes into account the probable source values for the intended application and is designed to minimize hash duplicates among these values. Hash values are often used in this fashion as a means to quickly narrow down a search in a large set of values, similar to how one would look up the name ‘Smith’ a phone book by first locating names beginning with ‘S’.

A second application for hash values is as a means to verify data integrity. If when transmitting data the sender also sends a hash value for that data, the receiver can use the same hashing algorithm to compute his own hash value and compare it to the received hash value. If the two hash values match, it is unlikely that the received data or hash value have been corrupted during transmission.

Distributed version control systems like Git and Mercurial use specific kinds of hash functions called *cryptographic* hash functions. These functions are typically one–way and collision resistant, meaning that it is very difficult to find some source data that yields a specific hash value and that it is rare for two typical sets of source data to yield the same hash value. This provides some protection against intentional corruption, since a third party cannot easily forge new data that matches the existing hash fingerprint. (Preneel et al., 1993)

To create a hash identifier for a revision, the version control system passes all the data from all commits and deltas included in that revision through a cryptographic hash function. The resulting value is practically unique\(^1\) and depends solely on the history that makes up the file. The identifier itself is also used to verify data integrity when checking out files from the repository.

With normal version numbers, identifying the previous or next revision in line is trivial (merely increment or decrement the version number). Since hash identifiers are unpredictable however, the repository needs to explicitly store the identifiers of the previous and next revisions (hereafter referred to as parent and child revisions).

Storing this information explicitly also provides enough versatility to intuitively encode branches and merges, by allowing all commits to have up to two parents and any number of children. A merge commit has two parents.\(^1\)

\(^1\)Duplicates should be exceedingly rare with a good choice of hash function.
whereas normal commits have only one, and branches start by having a commit with more than one child. Note that having “proper” merges in the repository like this changes the revision tree into a more general acyclic graph.

First–class merging

Pushing and pulling patches to and from remote repositories is a very common operation in distributed version control systems (corresponding to updating and committing in CVS). These operations require good merging capabilities, since every pull requires foreign patches to be merged into the local repository tree.

A great weakness of CVS and earlier systems is that while merging was supported it is mainly a manual process of repeated cherry–picking (although CVS lets you merge a whole series of consecutive deltas in a single operation). Merging two branches means having to manually apply all of the branched revisions onto the first branch. Even finding the branch point needs to be done manually, often forcing users to keep track of their branch points by tagging them before the branch is created.

Modern version control systems in general and distributed systems in particular avoid problems by making merging a first–class operation, strongly supported at the very core of the system. This means merging two branches should be as easy as entering a single command with the two branch names, whereby the VCS performs all the necessary low–level work with no additional user input.

Atomic commits

Having already been included in most version control systems since CVS, this feature is by no means exclusive to distributed systems but nonetheless deserves explicit mention.

A problem in CVS stemming from its RCS roots is that it doesn’t handle multiple files very well. Most notably, commits are not atomic, meaning that in a commit spanning multiple files the changes for each file are committed separately. If an error occurs midway it’s therefore possible for a partial commit to be recorded in the repository, where only some of the changes are included. This causes the current version in the repository to end up in an unintentional bad state.

Modern version control systems have atomic commits, meaning that unless all changes to all files are successfully recorded, no changes are made to the repository. This ensures that the repository never ends up in an “in–between commits” state.

This feature is closely related to the global commits discussed earlier, where version identifiers refer to the entire repository state instead of only the files included in each commit.
CHAPTER 2. BACKGROUND

Other features

Current version control systems, including but not limited to distributed systems, often support modern technologies like Unicode and tunneling. Unicode support typically means the VCS can perform differencing on Unicode files and support Unicode characters in commit metadata, while tunneling is usually the means to communicate between peers (or client/server) through existing established network protocols like HTTP and SSH.

As technology advances, so do expectations on applications. Many modern version control systems therefore have very similar feature sets in this area, possibly in part due to not wanting to appear “behind the times” by not supporting established technologies.

2.2.3 Drawbacks of distributed version control

Distributed systems suffer from drawbacks compared to centralized version control as well, mostly originating from the very versatility that is also one of the main strengths of a DVCS. Some of the criticisms against distributed version control are summarized by Clatworthy (2007), in part relayed here.

While a DVCS doesn’t require a central repository, in reality nearly all software development environments will still need one for coordination purposes. There may therefore be frequent situations where a centralized system would provide greater usability, since a distributed system might cater to more generic workflows at the expense of overlooking the most common ones.

The very frequent (and hence small) commits encouraged by distributed systems may also pose a problem, “spamming” the repository with a great deal of incremental work commits that have very little individual meaning. This is to be contrasted with centralized systems that instead encourage committing only once a piece of work has been finished, making for a more concentrated and meaningful history.

Problems with plentiful small commits may be lessened by a more liberal use of branches — one for each work task — so that the merging of a branch may instead be considered the single “meaningful” commit for that task. This viewpoint is advocated by Linus Torvalds through Git (Branching and merging with git, 2006), but it is not clear if current workflows are readily adaptable into such branch–based variants. Having a larger history also places a greater constraint on the version control system’s performance, since it generally has to traverse a greater amount of metadata.

Another issue plaguing many distributed systems today is their relatively young age and immaturity. Compared to CVS distributed systems are still new and haven’t received nearly as much attention as the industry–embraced Subversion. New features, while promising, still suffer from an overall lack of the robustness necessary for a DVCS to be considered as realistic replacements for centralized systems.
2.2. DISTRIBUTED VERSION CONTROL

Because of such drawbacks the adaptation of distributed version control will most likely be an incremental one. Initially they are likely to be used to largely emulate existing centralized workflows, before making any extensive use of their distinguishing features.

2.2.4 Existing distributed systems

There are currently many distributed version control systems available, both commercial and open source projects. They usually vary slightly in their implementation and have slightly different feature sets, but overall they mostly behave in similar fashions.

One of the earliest open–source distributed version control systems was GNU Arch (started in 2001), though other earlier examples include the commercial systems Code Co-op (1997) and BitKeeper (1998).

A brief description of the some of the current most prominent open–source distributed systems follows (with no particular internal order):

**Bazaar**

Bazaar succeeded an earlier distributed version control system called Baz, which in turn was based on the GNU Arch system. Baz was supported by the Canonical Ltd. company, until the same company funded the development of Bazaar itself in 2005. Bazaar has since then become GNU Bazaar as it has been accepted as part of the GNU free software project. (*HistoryOfBazaar*, 2008)

Bazaar is implemented in the high–level interpreted Python language, making it slower than a VCS implemented in a fast lower–level language. However, for inter–repository operations the network latency becomes the primary bottleneck and Bazaar’s performance is comparable to other systems. Bazaar also claims to have efficient repository storage, which would make it a good choice when memory/space complexity is a limiting factor. (*Bazaar Benchmarking Results*, 2007)

**darcs**

Darcs has, as touched upon earlier, taken a more theoretical approach to version control, starting by clearly defining notations, algebras and theories for working with patches. While most version control systems work according to similar theories implicitly, darcs is aimed to work follow its explicit specifications from the underlying patch theory, possibly allowing for cleaner code and more predictable operation in corner cases.

The theory of patches has given darcs a solid scientific foundation, though most of it is actually applicable for all of version control itself. Unfortunately, following the theory so rigidly has afflicted darcs with some negative side effects.
Until recently, darcs has suffered from severe performance problems related to automatic conflict resolution, more specifically when trying to resolve conflicts by commuting (reordering) deltas. During merging, processing any deltas following a conflict yields exponential $O(e^n)$ performance, usually freezing the entire program (ConflictsFAQ, 2008). These issues will supposedly be resolved with a new repository format currently being developed for the next version of darcs (DarcsTwo, 2008).

Darcs is implemented in the high-level Haskell language, which it is sometimes criticized for as such languages usually make the application slower to execute. David Roundy, darcs’ original inventor, has rationalized his decision to use Haskell with that the language simplified the implementation and that none of the encountered problems proved insurmountable. (Roundy, 2005)

Git

Git was created by Linux creator Linus Torvalds in order to maintain the Linux kernel development. The Linux kernel had been managed using the commercial BitKeeper version control system up until 2005 when the BitKeeper free license was revoked, forcing open-source projects using it to seek out other solutions. (Google Tech Talk: Linus Torvalds on git, 2007)

Similar to CVS, Git was originally a collection of scripts around a set of core components written in C code, where the scripts have later been rewritten in C code as well for increased portability and performance. That’s where the similarities to CVS tend to end, though.

One of Torvalds’ explicit design goals for Git was to make it as opposite as possible to CVS due to his own dislike for the early version control system. Other design goals were for the system to be distributed, robust and to have high performance. When he was not able to find an existing VCS that fulfilled all of his criteria, he started writing Git instead.

Git’s most prominent feature is its speed, which it achieves mostly by having its core implemented in C code instead of in an interpreted language, but also by certain design choices such as letting garbage accumulate in the repository database instead of cleaning it up continuously. Git doesn’t store revisions as deltas, but instead stores the complete files. To save space it then periodically groups similar-looking files together and applies delta compression to those.

Some of Git’s other features make it unique among current version control systems, but it also has several drawbacks that are troublesome for new users. Firstly, Git is not as easy to use as other systems such as Mercurial (although this could be greatly mitigated by a graphical frontend), and secondly, Git relies heavily on UNIX/Linux operating system features, making it hard to port or use on Microsoft Windows platforms.
2.2. DISTRIBUTED VERSION CONTROL

Mercurial

Matt Mackall began creating Mercurial at roughly the same time Linus Torvalds started working on Git, and for the same reason — providing a replacement version control system for the Linux kernel. He announced his new version control system in early 2005, early on establishing ease of use as a primary design goal. (Mackall, 2005)

Mercurial is primarily implemented in the *Python* language. As with darcs there has been criticism for using such a high–level language for fear of decreased performance, though the use of Python has in return enabled a cleaner implementation and a readily available mechanism for extending Mercurial with custom add–on modules. (O’Sullivan, 2007, p. 7)

Monotone

Development on Monotone started at around the same time as Subversion, around 2001. Both these projects were born out of a desire to create a version control system that wasn’t limited by CVS’ weaknesses, but whereas Subversion was created to be a better CVS, Monotone was created to be a fully distributed system with innovative new features.

Monotone is widely thought of as the first successful open–source distributed version control system. It established the since then common feature to use cryptographic hashes as identifiers.
Chapter 3

User interface design

In this chapter we will explore the interactions between users and version control systems. We will be determining some of the common rules for user interfaces (more specifically, graphical user interfaces) after which we will look at the user interactions and existing user interfaces of centralized version control systems.

The key issue we will attempt to address is how to design good graphical user interfaces for version control systems in general, and later how to adapt this for distributed version control so that we fulfill the following requirements:

- **Recognizable**: The user interface should be readily usable for users accustomed to current (centralized) VCS frontends.

- **Illustrative**: The user interface should clearly illustrate the current state of the repository and implicitly help the user learn to use the system efficiently.

- **Usable**: The user interface should conform to usability guidelines relevant to the intended environment. (operating system, development environment, etc.)

3.1 General GUI design guidelines

Graphical user interface design is a field that has grown and gained increased recognition in later years, much thanks to the work of people like Alan Cooper. GUI design is also commonly known as interaction design, a term used by Cooper in his authoritative book *About Face*, which is also the primary reference for this chapter.

Graphical user interfaces typically need to have several properties in order to be successful, regardless of the application. For example, they need to be easy to use even if the user is a beginner, not waste the user’s time with
unnecessarily complex interaction, and support more advanced time-saving techniques as well as gently encourage the user to gradually learn them.

Cooper divides users into three categories according to their experience with a GUI; beginners, intermediates and experts. The user interface should be designed to help beginners quickly become intermediates, since users will not want to remain beginners. It must also cater to experts since these users typically have a lot of influence over other users and as such should not be allowed to be dissatisfied with the interface. There is however no need to flaunt such features in the user’s face, as intermediates wishing to become experts will seek these features out themselves. (Cooper & Reimann, 2003, pp. 33–38)

The group of intermediate users is the most important one for interaction designers, since most users will firmly belong to this category. Intermediate users have graduated from being mere beginners and as such will use some of the more advanced features of the interface, but they will still require help with using and remembering these features from time to time. Cooper argues that all user interfaces should be optimized for intermediates, ensuring that a good balance of powerful features and gentle hints is maintained and thus allowing effective interaction for the majority of users.

In the following subsections, we will examine some important rules for good interaction design.

3.1.1 Conforming to user expectations

Since the introduction of graphical user interfaces, various de facto standards have evolved though a process of inspiration and mimicry. New applications will often try to mimic other existing applications in terms of user interface in order to reduce the time required for new users to adapt to the new product, making it easier to switch from a competing product. (Cooper & Reimann, 2003, pp. 243–245)

Conforming to existing practices is typically favorable for the above reason, and not conforming would severely hinder an application’s chances of being successful. For example, on the Windows platform an application’s configuration editor is typically found as “Options...” under a “Tools” menu or similar. But there also exist applications that for other reasons put this functionality as “Preferences...” under the “Edit” menu. A user accustomed to other Windows applications will most likely have some trouble locating this functionality because of the non–standard placement.

However, obeying practices should not be done without questioning; what about when the standard way of doing things is misleading? Following standards for the sake of standards is not necessarily beneficial, and it might sometimes be better to take a risk by breaking with tradition in favor of new ideas more in line with the user’s mental models. The interaction designer should always ask himself what the user is trying to accomplish and design the user interface to conform to and assist that. Cooper refers to this as
3.1. GENERAL GUI DESIGN GUIDELINES

When designing a user interface, a balance between tradition and innovation must be achieved. The designer should not be afraid to break with tradition and promote non-standard kinds of user interaction, however if the user is to approve of such significant changes, the changes must be significantly better than the existing alternative. If the suggested innovation isn’t truly superior, it is better to obey standards. (Cooper & Reimann, 2003, pp. 29–31, 245)

3.1.2 Providing smooth operation

A basic principle of all good user interfaces is that they should aid the user in accomplishing his goals without getting in the way. For graphical user interfaces this is especially true; a good GUI can greatly improve the user’s productivity but a bad GUI can severely impact it. Every time the program takes a wrong turn or incorrectly guesses the user’s intent, the user must step in to manually correct it. It is therefore important to anticipate the common usage scenarios and make them as smooth as possible.

Cooper condemns several common “features” prevalent in many current GUIs for needlessly disrupting the user’s workflow. One of his primary concerns is with the message box, the small dialogs that pop up to present the user with small bits of textual information or request simple responses such as ‘Yes’ or ‘No’. The problem with message boxes is that they are modal, i.e. they interrupt the user in whatever he was doing and require input before the user is allowed to continue.

Modal windows, typically referred to as dialogs, break flow by requiring the user to take his attention off whatever task he was working on. In contrast, good GUIs tend to remain as transparent as possible without interrupting the user, letting him focus his attention on his primary task instead (e.g. writing a report or browsing a revision history).

Many dialogs are used merely to confirm an action the user has already chosen to perform, usually (but not always) because the application can’t undo that particular kind of action. This can greatly hinder the user’s productivity by forcing the user to click twice to perform these kinds of actions, and in fact serves no long-term purpose since the repeated clicks quickly become such a habit that in the rare occasion that the user actually executed the action incorrectly he will not notice it anyway until it is too late. It is therefore much more preferable to instead ensure the action is undoable and perform it without confirmation.

Not only dialogs break flow; normal, non-modal windows can do so as well if carelessly used. Cooper likens the act of displaying a dialog to asking the user to enter a separate room; similarly, separate windows should ideally be used only when the operation the user is performing is somehow separate from the primary task. Printing is a good example of when a separate window is warranted, since the user has by then already gone from thinking...
“I’m writing my document” to “I want to print my document”. Many GUIs even implement printing and the related settings as a modal dialog, though the user would be well within his rights to wonder why he is not allowed to make changes to his document just because the window with the printer settings is open.

To avoid lessening the user’s attention to the primary task, the interactions most closely related to this task should be put in the primary window itself. The user should only have to interact with other windows in order to accomplish less frequently used or more advanced operations.

Whenever the user has to click or enter something, valuable time which could have been spent on the actual work has been lost. Therefore the interaction designer should strive to make sure every click and action is spent as well as possible. A good example of a time-saving feature is autocomplete, a mechanism by which the program remembers things the user has previously entered to avoid him having to re-enter the entire thing when it’s needed again later.

### 3.1.3 Providing feedback

If we are to present the user with information without interrupting flow, we need to utilize non-modal feedback and what Cooper calls rich visual interaction. This refers to using visual hints to provide additional information, for example via color-coding and small representative images (icons) and other means which serve to provide a rich, clear overview of the information to be presented. Audial hints can also be used, but are less common (an example is the navigational ‘click’ sound in Microsoft Internet Explorer).

Visual feedback is also needed to communicate the behavior and usage of various GUI elements such as buttons and text boxes. For example, clickable buttons tend to appear raised or do so when hovering the mouse cursor over them, while editable areas tend to have a brighter background. If the designer conforms to established practices for such visual hints, the user quickly understands how to interact with the various parts of the GUI.

Another good use for visual hints is to provide feedback while the user is currently performing an operation, such as dragging the mouse to control something. Rich visual interaction like this can be something simple like visually indicating where something can be dropped while drag-dropping, or something more complex like presenting the user with live previews of the final result extrapolated from the current interactions.

Using rich feedback like color-coding and icons can introduce ambiguities stemming from the user’s interpretation of the visual elements. Cooper therefore recommends frequent use of another common technique for visual feedback: the Tooltip. ToolTips are small non-modal boxes with explanatory text that appear when the user hovers the mouse cursor over an element for a brief time. They can be used to provide explanations for any element the user might not understand at first glance (such as toolbars and iconic
controls), but can also provide additional information accessible by hovering (e.g. the exact value of data represented in a pie chart).

### 3.1.4 Encouraging exploration

Since the primary goal of an GUI is to help the user get his work done faster, we must also consider the skill of the user. When the user first starts using the application he is a beginner, but as long as he remains a beginner he’s not getting the work done as efficiently as possible. Therefore, the GUI should be designed to help the user quickly become an intermediate user, as described earlier in this section.

One of the best ways of teaching a user is to encourage him to explore the application and experiment with its features. There are several ways to improve the chances of this happening, one of which is what Cooper calls a “pedagogic vector”. By this, he is referring to subtly showing the user that there are more effective ways to perform certain actions, for example by having the same icon next to a menu item as on an equivalent toolbar button.

There are also several ways to discourage a user from exploring, which should naturally be avoided. “Dangerous” actions should not be easily accessible, since the user probably will not want to end up accidentally clicking the “erase hard drive” button while looking for how to make text a different color. Even more important is to keep all actions undoable, as touched upon earlier; the user will not feel comfortable with experimenting unless he knows that anything that goes wrong can be easily fixed.

### 3.2 User interfaces for centralized systems

Since version control systems tend to use command-line input as their only built-in means of user interaction, there exist a variety graphical frontends that abstract away some of the low-level details and commands of well-established systems such as CVS and Subversion. In this section we will look through some of the frequently used features of such frontends as well as examine a few specific examples.

#### 3.2.1 Prominent GUI features

When examining the various frontends that are available for centralized version control systems there are several recurring features, a possible indicator of a successful GUI pattern. We will now examine some of these to further determine their function and usability.
CHAPTER 3. USER INTERFACE DESIGN

File/directory listing

Since version control systems by nature deal with files, one of the most important parts of a GUI frontend is the displaying of files in your working copy, along with the status of each file. Examples of statuses are:

- **Added** or **removed**, for files that have been scheduled to be added to or removed from the repository.

- **Modified**, for files that have been modified in the working copy. These changes need to be committed to the repository to become permanent.

- **Unchanged**, for files that are identical to the version stored in the repository.

- **Conflicting**, for files that encountered conflicts when applying deltas during the last operation (e.g. merging) and hence need to be manually resolved.

- **Unchecked**, for files that are not part of the repository and are hence not version-controlled.

Files that are added, removed or modified are all result in non-zero deltas when compared to the latest versions in the repository, and so they are sometimes collectively referred to as **committable** files. In the list of files, the status of each file is often indicated with special icons or some color coded scheme.

Files that are conflicting are usually highlighted in a strong color such as red, since conflicts usually must be resolved before any further action can be taken on the files. For the same reason, some frontends (e.g. Eclipse) will offer to take the user into conflict resolution mode as soon as conflicts are detected, so as to minimize the time files are left in a conflicting state.

Filtering

For projects containing many files it can be difficult to keep track of everything in the file system. For this reason it is common for the frontend to also offer filtering functionality, allowing the users to filter which files to display. The user can typically filter files on their names, types and status, but some clients (like WinCVS) allow for filtering on pretty much any property of files.

Flat mode

As an alternative to the normal tree-based file listing, several frontends offer a concept known as “flat mode”. Much like its name implies, it involves flattening out the directory tree structure so that not only files in the current directory but also all files in all subdirectories become visible simultaneously.
3.2. USER INTERFACES FOR CENTRALIZED SYSTEMS

in the current view. Combined with filtering, flat mode can give the user a good overview of the entire project, such as showing all committable files in the entire working copy in a single view.

Diff visualizer

Whether the user is exploring the history of a file or inspecting the most recent changes before committing, some way to visually represent a delta (diff) is typically needed. There are several approaches to this used in current GUI frontends, but they typically fall into one of two categories: **textual representations and side–by–side comparisons.**

Textual representations are the easiest to implement and typically involve just printing the output of the `diff` tool or some equivalent function exposed by the underlying VCS. While this method isn’t very user friendly, it does have some advantages for power users. For example, the user can quickly see the size of the delta (i.e. the number of affected lines) and hence can estimate its complexity without having to look at any statistics. It is also likely that a great deal of power users will also already be familiar with the output of the `diff` tool and hence will readily understand a textual diff in the frontend.

Side–by–side comparisons are accomplished by displaying both versions of the file (i.e. with and without the delta applied) next to each other, typically also highlighting the lines that differ between the two versions. This functionality is often available separately in graphical differ tools, and some GUI frontends choose to rely on such external tools for showing diffs. Figure 3.1 shows a mockup of a modern diff visualizer.

![Figure 3.1: A diff visualizer](image)

Due to its more ambitious nature, the side–by–side approach typically runs into a few problems that need to be resolved by each implementation. For example, since the two views don’t have the same content they don’t
necessarily line up well when scrolling, but for comparison purposes each line on one side ought to be lined up to its counterpart on the other side. Different visualizers solve this problem differently, such as inserting blank lines when necessary, or by scrolling the two views at different rates to always keep the content roughly aligned.

**Conflict/merge assistant**

A feature that is closely related to a diff visualizer is a conflict/merge assistant. As the name implies, this is a feature intended to simplify the process of manual merging and/or conflict resolution.

It is typically based on the side–by–side diff visualizer but instead of only highlighting differences it highlights conflicts, i.e. areas that have been changed on both branches of a merge so that the system cannot automatically merge them. Along with these highlights, it offers commands such as choosing to use either or both of the changes in the result, or manually editing the area to resolve the conflict.

While a diff visualizer displays two views a merge assistant typically displays as many four views, one of which is an editable result view that both displays the result of the merge and allows for manual adjustments. The other three views display the changes introduced from both branches in one view each, and a separate view for the last common revision. This allows the user to clearly see all changes by which the file has diverged along the two branches by comparing with their common ancestor.

A four–view merge assistant can be seen in Figure 3.2, where the common revision is in the upper middle with the two diverging branches on either side of it.

### 3.2.2 Examples of GUI frontends

This subsection will examine a few example frontends in further detail. They were picked based on the results of the usage survey (see chapter 4). Addresses of relevant websites can be found in the “Further reading” section of the bibliography.

**MarmaladeCvs**

MarmaladeCvs is, according to the survey results, one of the most popular CVS frontends in use at Opera Software, probably owing in part to the fact that it is developed by Opera Software employee Emil Segerås. A screenshot of MarmaladeCvs in operation is shown in Figure 3.3.

The user interface of MarmaladeCvs has a noticeable focus on the basic text–based interaction with CVS, dedicating half of the screen space to display a representation of a command–line prompt. All operations in the rest of the GUI that require interaction with the CVS repository are visibly executed in the right half of the screen (or queued up if commands are chosen
3.2. USER INTERFACES FOR CENTRALIZED SYSTEMS

Figure 3.2: The visualization part of a merge assistant

faster than the server can process them). This makes MarmaladeCvs usable for power users of CVS as well, and it also accelerates the transformation of intermediate users into power users by teaching them about the underlying CVS commands through clear cause–and–effect scenarios.

The left half of the window is mostly occupied by a combined file/directory listing, complete with flat mode and filtering capabilities. MarmaladeCvs uses the strong colors red and green to highlight conflicting and committable files, respectively. Right–clicking on any item in the list presents a context menu with commands for the selected items.

Finally, in the upper left corner is a set of toolbars and other controls, arranged into sets grouped by functionality. The user can switch between the different groups of controls by means of a tabbed interface. All commands are executed either by these controls, the context menu, or manually entered into the command–line interface.

**WinCVS**

WinCVS is one of the oldest and most feature–rich CVS frontends. It is a part of the CvsGui suite, a set of similar frontends catering to Windows, MacOS and Linux, respectively. A screenshot of the main window with the file listing exposed is shown in Figure 3.4.

The WinCVS client contains many of the features described in the earlier subsection, as well as some additional ones. It lets the user filter which files are shown in the file view as well as flat mode support, and it contains a
 CHAPTER 3. USER INTERFACE DESIGN

Tabbed toolboxes grouped by functionality
Flat mode and filtering controls
Indicator for CVS operations in progress
Input field to manually enter CVS commands
File and directory listing
Context menu for files and directories
Color-highlighted CVS command output

Figure 3.3: The MarmaladeCvs frontend (for CVS)

special graph view for visualizing the history of a file complete with branches and tags. It also has support for scripting, allowing power users to add their own commands in terms of one or more CVS operations.

After having been developed for a long time, the client has very extensive support for many of the possible CVS operations, including some of the rarer ones like administrator tasks. These features makes WinCVS a very powerful tool for power users, but the sheer amount of exposed features also gives the program a steeper learning curve than the other frontends discussed here.

Most of WinCVS’s operations take place in various views all exposed in the main window.

Eclipse

Eclipse is different from the previous examples of GUIs in that it is not simply a frontend but a complete integrated development environment (IDE).
3.2. USER INTERFACES FOR CENTRALIZED SYSTEMS

Figure 3.4: The WinCVS frontend (for CVS)

Eclipse is built in the Java programming and heavily relies on plugins for providing functionality. For the purposes of this report, we will focus on the CVS plugin only, as it illustrates the approach of integrating version control with the user’s development environment rather than providing a standalone application.

Since Eclipse out of the box implements most of its functionality via plugins, so the underlying architecture is solid and allows for extensive features to be implemented on top of it. The application is centered around the concept of different views, between which the user can switch whenever he likes.

The CVS plugin in Eclipse adds some version control commands to the project view (acting as the file/directory explorer), with a secondary synchronization oriented view being activated as necessary. In this view, outgoing and incoming changes are displayed, and by double-clicking on a file
containing changes a diff visualizer or conflict resolution helper is shown.

The addition of a dedicated view along with common operations inserted into other views means the CVS plugin will mostly stay out of the user’s view until called upon, minimizing the risk of disturbing the user’s flow.

**TortoiseCVS/SVN**

While described here under the same heading, TortoiseCVS and its counterpart TortoiseSVN (for CVS and Subversion, respectively) are in fact two unaffiliated products. TortoiseCVS is the original application, but the concept proved popular enough for several variants for other version control systems to be developed, inspired by the original design. Apart from CVS and Subversion, there are now also Tortoise-style projects for at least darcs, Bazaar and Mercurial. Here, the original TortoiseCVS will be used as a case example.

TortoiseCVS and its counterparts have taken a rather unique approach to version control frontend design. TortoiseCVS is not a standalone program, nor does it integrate with an existing IDE. Instead it integrates with the file/directory browser of the operating system itself (Microsoft Windows Explorer). It does this by adding small overlay icons to files and directories (signifying their status) and adding commands such as ‘diff’ and ‘commit’ to their context menus.

Integrating with the system file browser brings several advantages such as instant familiarity with controls. However, working within another application places severe limitations on GUI design, and all commands are necessarily intermingled with the normal file browsing commands (which can already contain many other extensions).

Even though the plugin solution is elegant and adds a measure of CVS functionality to the system file browser, TortoiseCVS only controls a small part of the interface the user actually sees. The rest of the interface is designed to be as generic as possible and contains any and all extensions added by other programs as well, even though these might make little sense in a version control context. This means that TortoiseCVS can’t maintain a designed focus on CVS tasks in the application as a whole, making a standalone application or IDE plugin likely to be preferable whenever more serious CVS work needs to be done.

### 3.2.3 Web interfaces

Modern version control systems, having been developed in the Internet age, often have web-based user interfaces available for them. Git, for example, includes a built-in web interface. These web interfaces typically provide passive introspection only, such as viewing the history and commits for any file in the repository.

Web interfaces can be a powerful tool to examine the history in a user-friendly manner, with the added benefit of automatically being cross-platform.
3.3 USER INTERFACES FOR DISTRIBUTED SYSTEMS

However, since web pages typically have no access to the local file system, web interfaces are not suited for any synchronization work with a local check-out or cloned repository. Therefore, while being very useful for retrieving information from repositories, they cannot replace proper native GUIs in terms of usability.

3.3 User interfaces for distributed systems

Since distributed version control introduces many new concepts not found in previous systems (and hence not in previous graphical frontends), it is necessary to determine what kind of features a graphical frontend will need, and which parts of the underlying version control system are most important to make accessible.

Rather than be invented from scratch, a graphical user interface for the new generation of version control systems should draw upon the techniques described earlier in this chapter in order to fulfill the first requirement mentioned at the very beginning; recognizability.

In order to learn how to “evolve” the current user interface concepts to suit distributed version control systems, as part of this thesis a usage survey was carried out. This survey is described in the next chapter.
Chapter 4

Usage survey

As a part of this thesis a version control system usage survey was carried
out. The survey had multiple goals:

1. Determine which aspects of centralized version control systems are in
greatest need of improvement in terms of usability, and which tools
and frontends are being used to supplement them.

2. Determine which distributed systems are most popular.

3. Determine which aspects of distributed version control systems are in
greatest need of improvement in terms of usability, and gather ideas
and suggestions on how such improvements could be achieved via a
graphical frontend.

Results from the survey were used to provide a measure of statistical
support for the conclusions in this report, but more importantly served as
inspiration and as indicators of which areas to focus on.

The complete list of survey questions and their responses are included
in this report as appendices A and B for completeness.

4.1 Method

The survey was carried out as a self–administered online questionnaire, pri-
marily among Opera Software employees. The target population were users
of version control systems in general, and of distributed systems in particu-
lar. The questions closely matched the stated goals of the survey and were
chosen through a simple iterative process, gathering feedback from a trial
participant but without the aid of a proper pilot study.

The decision not to use more formal methods for the survey design was
mostly due to time constraints, but it was also expected that the statistical
validity of the results would be primarily limited by the small population
and not the question design. Following basic guidelines for questionnaire
design was deemed sufficient for the purposes of this thesis.

4.1.1 Constructing the questionnaire

In order to improve chances of participants completing the survey, the ques-
tions were deliberately few and concise. They were designed to be brief
with short descriptive texts and arranged to flow as naturally as possible
between the relevant topics in order to minimize confusion, since there was
no interviewer present to assist during the actual survey.

As a measurement of usability, pairs of matrix questions were used. The
rows in the matrix were the same in each pair and listed various aspects of
the version control system. For each of these, one question would ask the
user how frequently he used the particular aspect of the system, and the
other would ask how difficult to use he perceived it to be, both on a scale
of five options. The reasoning was that by looking at both of these values,
one could estimate the overall usability benefit of improving this one aspect
of the system (e.g. via a graphical frontend).

The questionnaire was divided into two distinct parts; one part dedicated
solely to centralized systems, and a second part dedicated to distributed sys-
tems. This way, participants with limited or no experience with distributed
systems could still contribute by answering the questions in the centralized
part only. The second half of the distributed part dealt specifically with
graphical frontends and asked participants to contribute their own ideas for
improving usability.

In the first attempt the two parts of the survey were contingent on an
opening question: “Do you have experience with centralized/distributed
version control?” This was later deemed unnecessary, and all questions
(including all matrix sub-questions) were made optional instead. The par-
ticipants were encouraged to answer the questions they had experience with
and leave other questions blank.

Experience with centralized systems

To measure the usability of centralized version control systems, the partic-
ipants were asked to evaluate the usage frequency and usage complexity of
the following major features (see chapter 2 for descriptions):

- Checking out from a repository
- Adding or removing files
- Examining uncommitted data
- Synchronizing with the repository
- Committing
- Tagging
- Branching
4.1. METHOD

- Merging branches
- Cherry–picking
- Examining history of changes

Additionally, participants were asked to list any additional tools or front-ends they used when working with centralized version control. This in conjunction with the usability question closely matches the first stated goal of the survey described at the beginning of this chapter.

Experience with distributed systems

When moving on to the section for distributed version control systems, the first question was derived from the second goal of the survey, namely to identify which systems are popular. The question asked participants how much experience they had with various different distributed systems, including “None”. The question was arranged as a matrix question with the different systems as rows, with an additional row for “Other”.

After that, the user was asked to evaluate a list of features roughly corresponding to the one in the centralized section, but adjusted for the different features present distributed systems:

- Creating new repositories
- Cloning existing repositories
- Adding or removing files
- Querying status of working set
- Committing
- Branching
- Merging branches
- Cherry–picking
- Synchronizing with remote repositories
- Examining history of changes

Instead of then asking for additional tools used, the participant was asked to identify any particular problems or weaknesses in the systems they used. This was done to provide a means of communicating problems other than those that fit into the predetermined features of the previous question. To help the participants get started a few example topics were suggested. These were kept as generic and non-leading as possible.

Input on graphical frontends

The last two questions were included to fulfill the second half of the third goal of the survey, namely to gather input and ideas for a graphical frontend. These questions were highly subjective and vulnerable to bias both from the participant and the author but were designed to be mainly usable as
inspiration and guidance for the later design work and not for statistical analysis.

The first question pitched six random concepts that could be present in a graphical frontend, and asked the participant to grade these hypothetical features on a scale from “must–have” to “useless”:

- Tree visualization of the repository history
- Graphical diffs between versions or for uncommitted data
- Remembering remote repositories with bookmarks/favorites
- Drag–and–drop interaction for committing/branching/merging
- Context menus for all applicable areas
- Better control of commits by letting the user select which changes to include

The second question was entirely free text, and asked the participant for any other ideas or suggestions that could be useful in a graphical frontend. As assistance, the participants were encouraged to think in terms of what would make up for a particular weakness identified earlier.

### 4.1.2 Sampling

As stated earlier, the target population for the survey was people that have experience with distributed version control, though people that have only used centralized version control could also participate by answering the first part of the questionnaire only. Out of convenience, most participants were employees of Opera Software. This was a logical choice since Opera developers have experience with version control from their daily work.

It was not known beforehand which of the hundreds of Opera employees had experience with distributed version control. In order to keep things simple, a general invitation was sent out to all employees, asking everyone with experience or interest to participate in the survey.

Selecting participants in this manner could be considered a mix of convenience sampling and purposive sampling, since participants are chosen based on being readily available but also based on their experience. Both of these are examples of nonprobability sampling, meaning that the selected participants are not statistically representative for the total population (i.e. all users). In exchange, such sampling is considerably easier to use.

The choice of population and sampling method brings at least two limitations that should be considered:

1. The respondents consist only of Opera Software employees.
2. The respondents consist only of people who noticed the invitation and took the time to answer the questionnaire.

Results from the survey cannot be easily generalized to the entire population of version control system users, since both points above may have introduced bias. For example, Opera Software employees may have experiences
4.2. RESULTS

with these systems that are skewed from those of the general population due to company-specific workflows.

Some generalization can still be performed by applying reason based on familiarity with the topic, even though it will carry limited statistical validity. For example, it can be reasonably argued that an identified usability problem should be addressed even if (in the worst case) the problem is unique to Opera Software employees. It is therefore believed that the results should be valid enough to support the conclusions in this thesis.

4.1.3 Maximizing response rate

As mentioned in the previous section, the questionnaire was kept short and compact to increase the chances of participants completing it. All responses were also by default anonymous, though participants were given the choice to voluntarily leave contact details.

The invitation to participate in the survey was sent twice, with a few weeks in between. The second invitation was sent hoping to attract the attention of anyone who didn’t notice the first one, and to serve as a gentle reminder for those who were interested but did not have time the first time around.

4.2 Results

A total of 20 people completed the usage survey, 18 of which were employees of Opera Software. The last two were invited at the last minute to participate independently and had no affiliation with the company, in an attempt to verify the statistical reliability of the survey. Their responses relating to usability did not significantly deviate from those of the Opera Software employees.

Regarding response frequency, 18 out of an estimated total of 300 developers participated in the survey (6%), however since the survey was explicitly directed at people with an interest in or experience with distributed version control, we must take into account how many people actually felt targeted by the invitation. Based on observing internal company discussions, the number of people with such experience could be loosely estimated to somewhere between 20 and 50, which would mean a response frequency of between 36% and 90%. Even though largely unknown, this was deemed acceptable for the purposes of gathering input.

To avoid duplication the responses to the questionnaire are not described here, but can be found in appendix B.

4.2.1 Usability

The most important questions of the survey were those relating to the usability of version control systems. The method used to analyze these responses
CHAPTER 4. USAGE SURVEY

will be described in this section, showing the responses for distributed systems. The corresponding results for centralized systems are not included here for brevity, but are briefly summarized at the end of the section.

For each kind of system, the questionnaire inquired about the usage frequency and usage complexity of various features of the system. Each feature sub-question could be answered ranging from “Never” to “Extensively” for frequency, and from “Transparent” to “Unusable” for complexity. Since these are merely ordinal indicators, the first step in analyzing the responses was to try to assign specific values to them. Each answer was scored according to Table 4.1, after which the average score of all participants was calculated for each feature. These averages for frequency and complexity in distributed systems are shown in Table 4.2.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Extensively</th>
<th>Often</th>
<th>Sometimes</th>
<th>Rarely</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Unusable</th>
<th>Complicated</th>
<th>Manageable</th>
<th>Easy</th>
<th>Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0.5</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 4.1: The scoring of frequency/complexity answers

Having scaled the frequency and complexity of features into easily manageable values, we can create an even more useful composite measure by combining the two; an “unusability” index.

If the usage frequency of a certain feature is low, then the user will rarely run into problems with it. Alternatively, if the complexity is low, then the user does not perceive the feature as a burden. In either case, the feature can be argued to work “well enough” already. However, if both values are high then the user is likely to be bothered by the complexity of the feature frequently; in other words, making this feature easier to use should be a priority.

To achieve the desired property of our unusability index, we multiply the frequency score with the complexity score of each feature. The resulting values, arranged in descending order, can be seen in Table 4.3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Frequency</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating new repositories</td>
<td>1.5962</td>
<td>0.5000</td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td>2.0385</td>
<td>0.4167</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td>2.0385</td>
<td>0.3958</td>
</tr>
<tr>
<td>Querying status of working set</td>
<td>3.2885</td>
<td>0.4792</td>
</tr>
<tr>
<td>Committing</td>
<td>3.3077</td>
<td>0.4808</td>
</tr>
<tr>
<td>Branching</td>
<td>1.8846</td>
<td>0.4091</td>
</tr>
<tr>
<td>Merging branches</td>
<td>1.3077</td>
<td>0.5500</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td>0.8654</td>
<td>0.9167</td>
</tr>
<tr>
<td>Synchronizing with remote repositories</td>
<td>2.3654</td>
<td>0.5455</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td>2.0577</td>
<td>0.7273</td>
</tr>
</tbody>
</table>

Table 4.2: Frequency/complexity scores for distributed systems
Table 4.3: The quantified “unusability” of DVCS features

<table>
<thead>
<tr>
<th>Task</th>
<th>Unusability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Committing</td>
<td>1.5902</td>
</tr>
<tr>
<td>Querying status of working set</td>
<td>1.5757</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td>1.4965</td>
</tr>
<tr>
<td>Synchronizing with remote repositories</td>
<td>1.2902</td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td>0.8494</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td>0.8069</td>
</tr>
<tr>
<td>Creating new repositories</td>
<td>0.7981</td>
</tr>
<tr>
<td>Cherry–picking</td>
<td>0.7933</td>
</tr>
<tr>
<td>Branching</td>
<td>0.7710</td>
</tr>
<tr>
<td>Merging branches</td>
<td>0.7192</td>
</tr>
</tbody>
</table>

The corresponding least usable features identified for centralized systems were merging, branching, cherry-picking and tagging. Notably, these features rank as the most usable on distributed systems, possibly indicating that their low usability resulted in them becoming a priority during the design of the new systems.

In response to the free text question on distributed systems, several participants complained about the very lack of graphical frontends and related issues such as complicated command-line interaction. Other complaints revolved around how different systems tend to solve things in slightly different ways, meaning that what you learn from working with one system isn’t necessarily applicable when dealing with another. For a complete listing of responses, see appendix B.
Chapter 5

Proposals

Based on the results of the usage survey and keeping in mind the GUI guidelines discussed earlier, this chapter proposes a number of suggested features that could be beneficial to a graphical frontend for a distributed version control system. Although many of these proposals are merely the product of brainstorming and not rooted in any particular scientific method, some were suggested by survey respondents during informal talks following their participation.

Some early suggestions were also presented in the usage survey in order to get some feedback on whether the users believed them to be worthwhile additions. Each of these suggestions has been given appropriate priority among these proposals according to the degree of positive feedback received.

5.1 User requirements

As shown earlier in Table 4.3, the four primary problem areas identified in the usage survey were:

- Committing
- Querying status of working set
- Examining history of changes
- Synchronizing with remote repositories

First we should note that the top two items are both closely related to the work the user is actively doing in the working set of his own repository. This is not a surprising result since a lot of the time the user spends interacting with the system will involve the working set, but the perceived unusability tells us that there is still more work that can be done to make these tasks go smoother.

To help resolve the first two items, it is important to give the user clear and intuitive tools for inspecting exactly what work the user has done since
last committing, and precise control over committing this work to the repository. The user should feel in complete command over this process, but it should also be optimized to get the task done as fast as possible.

The third item revolves around the user inspecting past changes, either his own or changes imported from a remote repository. It is important that this history is readily available in an intuitive manner, so that the user can get feedback about the effects his changes have on the repository’s structure.

The user will typically inspect the history to learn about the current document; he may be trying to understand why a certain change was made, or by who. Since the repository history is merely a database of changes that might not be suitable for being exposed directly to the user, it may be preferable to provide several different ways to access the history information depending on the use case.

The final item deals with sending and receiving changes to and from remote repositories (separate from committing to the local repository). This may require frontend assistance as a user familiar with centralized systems may not recognize that the necessity for this added step. The possibility to synchronize with multiple remote repositories can also benefit from additional support; for example, the frontend may provide fast access to frequently used repositories and indicate whether or not a synchronization is needed.

Additionally, the survey responses indicated that several different distributed systems are being used, neither of which possess a dominant market share. Instead it seems to be rather common to have experience with multiple systems. Under these circumstances, a graphic frontend creator will have to carefully consider which system(s) to support.

5.1.1 Survey suggestions

As mentioned in chapter 4, participants in the survey were asked to consider a few suggested features, as well as encouraged to suggest their own features for a graphical frontend. They were asked to score the following suggestions according to usefulness:

- Tree visualization of the repository history
- Graphical diffs between versions or for uncommitted data
- Remembering remote repositories with bookmarks/favorites
- Drag-and-drop interaction for committing/branching/merging
- Context menus for all applicable areas
- Better control of commits by letting the user select which changes to include

Out of these, the graphical diffs were scored as the most important feature to have in a graphical client, which makes sense since it is a feature already present in many frontends for centralized systems (see subsection 3.2.1).
Among the other features, history visualization was ranked as very important, closely followed by remote repository assistance, in line with them being identified as a primary problem area by the other questions. Finally, commit helpers received very high scores, indicating that users want more control in this area than current frontends tend to give them.

Out of the user-suggested features, graphical assistance for merges and conflict handling was requested as well as visual previews when synchronizing with remote repositories. The latter would seem to indicate that users do not yet feel comfortable in predicting the effects foreign changes will have on their repository when synchronizing, so a graphical frontend should try to assist in this area.

5.2 Reusing centralized GUI features

Since distributed systems work roughly the same as centralized systems until you start branching and synchronizing with remote repositories, there is plenty of opportunity to reuse features dealing with this common functionality from centralized GUI frontends. Specifically, the methods used to improve problem areas of centralized systems such as branching and merging may still be applicable, even if the problematic features have been “fixed” in the distributed system.

Firstly, a diff visualizer is just as useful in a distributed environment, even unchanged, so one should be included. The graphical side-by-side comparison view provides for a user interface that is easier to learn for beginners. The problem of how to align the two sides (see page 33) can be solved in either way since both methods have their advantages and drawbacks. The GUI can also provide a readily accessible switch in the diff view to alternate between these methods and even the textual representation if desirable.

Having an integrated diff visualizer is preferable over relying on external tools, since switching between windows/applications will risk breaking the user’s flow (Cooper & Reimann, 2003, pp. 326–330). However, since users may prefer working with their diff visualizer of choice it may be preferable to include an option for using an external tool instead of the integrated one.

Features like the file listing are also intrinsically tied to the concepts of version control themselves, and therefore should be included in a distributed GUI as well. Reusing an existing feature or “look” can be a good idea since it makes the new GUI more recognizable for users of the older GUIs.

Similarly, users may have grown so accustomed to certain features in their current frontend that they will feel reluctant to start using a GUI that lacks them. These may include filtering capabilities and flat mode, and are probably necessary to include in any frontend that has a traditional file listing, lest it feel “incomplete” to users.

The diff visualizer and file listing, if implemented in an accessible and intuitive manner, are useful tools for examining the work that will become
the next commit (i.e. the current working set), and as such special care should be taken to ensure these present as much relevant information as possible without intruding on the user’s ability to actually perform said work.

Closely related to the diff visualizer is the conflict/merge assistant. As described earlier, it shares the same basic split view of the file revisions involved, but also includes the composite window where the result is fine-tuned. The merge assistant should have as much information as possible available to it from the version control system so it make the merge operation easier (e.g. using a base revision for good three-way merges).

5.3 Proposed new features

Many of the problems experienced by users of distributed systems are caused by unfamiliarity with the new features of these systems themselves, and/or by these features having been implemented in a less than user friendly manner. Since they are new features, we will sometimes need to think in new ways to assist the user in the frontend as well.

The following features are either new in the sense that they highlight a new way of thinking introduced by distributed systems, or new in the sense that they have not appeared prominently in existing frontends despite being applicable in for centralized systems as well.

Commit helper

A commit helper, as the name implies, assists the user when committing changes to the repository. While all version control systems provide easy commands to commit the current working set, often users will want more fine-grained control over exactly what goes into the repository.

In the software industry developers will sometimes end up working on several things at once in their local working set. The changes can end up growing dependant on each other and the developer can’t commit until everything is done, resulting in a single large commit with multiple inseparable effects. Company guidelines will often discourage this scenario, since the commits won’t correspond to single changes and as such each change becomes less identifiable and can’t be individually reverted.

Distributed systems will help limit this problem by itself, for example by having a local repository with full history and much better handling of branches, allowing the developer to commit more often and separate his work with much less overhead than earlier systems.

A commit helper can further help by making it easier for the user to commit only parts of his current changes to the repository. Existing commit helpers typically allow the user to specify which of the modified files go into the commit, but one can go further and let the user choose which specific
changes inside the files should be included as well through some specialized form of a merge helper.

Through the commit helper, the user can construct a detailed “commit set” out of a more cluttered current working set. Hopefully, this commit set is more precise and/or requires less work to create than if the user has to separate his changes into neat commits manually. Once changes are scheduled for a commit the commit set can be displayed next to the file listing along with input fields for metadata like commit messages. The commit set mirrors the commit dialog found in current frontends, but would remain visible, inspectable and modifiable as part of the normal user interface until the user decides to commit the included changes. A mock-up of such a commit set view is shown in Figure 5.1.

Figure 5.1: An example of a “commit set” user interface

A non-modal commit set containing selected modified files or individual changes inside files lets the user directly control the precise changes included in commits from the frontend itself. The user won’t have to manually edit files to achieve partial commits and can more easily keep track of which changes logically belong together while working on them. One can even envision constructing multiple commit sets in parallel out of the current working set. The end result is a repository with more detailed (and hence useful) metadata, without significantly increasing the workload for the user.

Unfortunately, not all version control systems have built-in support for partial commits, or a notion of a commit set. As a result, the commit helper might have to implement this logic itself and maintain additional state on top of the version control system, meaning the frontend will no longer be as light-weight as one might desire. It will be up to the frontend designer to choose whether to model the commit set handling on top of whatever support the version control system offers or to implement his own handling.
Another concern is that if the commit set is maintained in parallel with the ongoing work on its contained changes, how should additional modifications be handled? One suggestion is for the frontend to detect these modifications and ask the user what to do, like offering to merge the new changes as illustrated in Figure 5.2. However, it is important not to disturb the user needlessly; for example, the frontend should not ask in cases where the correct course of action can be automatically deduced.

![Figure 5.2: Action to perform when files in the commit set are modified](image)

Repository tree visualizer

The internal repository is a structure the user hasn’t needed to familiarize himself with much when using centralized version control systems. However, making sense of local branches and synchronizations with remote repositories can be overly difficult without knowledge of such local branches and their behavior during synchronizations. Therefore, a repository tree visualizer can fill the function of teaching the user about the structure of the repository, shortening the time needed for a beginner to become an intermediate user.

Note that the term “repository tree” is actually incorrect; as was established earlier the repository of a merge–capable version control system forms an acyclic graph. However, a tree is a more readily available concept for most users and therefore makes for a better descriptive term from a learning standpoint to use in the GUI. Those who object to this as inaccurate may imagine the term as more of a metaphor to aid the learning process of beginner users.

As mentioned, the repository tree is necessary because the actual structure of the history may be non–obvious to the user, but can still affect the outcome of common operations. Without clearly seeing the repository and the effect of these operations, the user can easily become confused as to
5.3. PROPOSED NEW FEATURES

why his actions are not producing the desired effect. The repository tree should therefore be visible at least during commit operations, highlighting the current revision to clearly inform the user of the current state.

There already exist several standalone tools to visualize the repository tree; for Git there is *gitk* and *qgit* and for Mercurial there is *Hgk*, for example. An screenshot of the *gitk* visualizer is shown in Figure 5.3.

![Repository graph (list of all patches)](image)

1. Repository graph (list of all patches)
2. Filtering controls
3. Patch details (commit messages, diffs)
4. List of files affected by patch

Figure 5.3: The *gitk* frontend, a repository visualizer

The tree should preferably be shown with rich visual feedback, such as showing details for specific commits in tooltips that appear when the user hovers over them with the mouse cursor, as this will generally improve its usability (Cooper & Reimann, 2003, pp. 451–453). As shown by the existing tools, color coding can also be used with good effect to make the different branches more visually distinguishable.

The use of rich visual feedback can be important for being able to display a smaller variant of the repository tree view. As seen in *gitk* and other tools, if detailed information on commits are to be included in the view it must be rather large to be able to encompass all the information it contains. However, there may be cases where it is preferable to show the tree in only a small
part of the screen, for example during commit previews. A “mini–version” of the tree can be used in these cases, where all non–essential information is instead available only via tooltips that do not consume valuable screen space.

**Favorite/recently used repositories list**

Compared to centralized version control, where the user normally only interacts with a single remote server (which can then be remembered as the active server by the VCS itself), distributed systems will open up possibilities of the user having to frequently interact with multiple remote repositories. Rather than forcing the user to remember and keep track of these repositories, a GUI frontend should remember all entered and recently used repositories in a bookmark-like fashion. This will reduce the instances of repetitively requesting information from the user during his normal workflow, especially information that the user has already entered earlier.

Preferably GUI elements with similar semantics should be grouped together (Cooper & Reimann, 2003, pp. 229–230), so the interface should provide some way to display and change the currently selected remote repository somewhere close to the other controls. This ensures that whenever the user is dealing with remote repositories, he will remain informed of the current target repository of the operation and can change it as necessary.

Depending on the user’s preference it might be favorable to instead let the user select one remote repository to be used as a “central repository”, indicating that the GUI should try to emulate a centralized version control workflow as much as possible with the selected repository as its target. Such a feature would cater to work environments where commits should be made to a centralized server but a distributed system is used as a backend.

**Previews and other feedback**

Whenever the user performs a complex operation with hard to predict results (e.g. a drag and drop synchronization or merge), the GUI should display some form of live visual feedback such as a modeless preview of the result, along with a clear option to abort the operation. This kind of feedback will encourage the user to experiment with operations without fear of breaking anything. However, it is imperative that the previews are not modal (e.g. a dialog box) or otherwise impede the user’s workflow, or the user will become annoyed with them. (Cooper & Reimann, 2003, pp. 450–453)

There are some exceptions to this rule. The commit operation, for example, involves inputting additional information as a final step (i.e. the commit message). This is a good opportunity to display a preview or summary of the files about to be committed, since we have to interrupt the user anyway to ask for the commit message. It is therefore acceptable to use a dialog box or similar window for this purpose, however the process should be as streamlined as possible to encourage users to commit often.
If feedback is implemented in a visually non-intrusive manner it can be helpful for both beginners and intermediates, but most likely some power users will consider these kinds of “training wheels” to be superfluous. Therefore, there should be a setting for disabling specific visual aids like live feedback. (Cooper & Reimann, 2003, p. 137)

**Undo functionality**

Not all kinds of operation can be assisted with previews. For example, there might not be enough information available to estimate the result without actually performing the operation. In cases such as these, it is especially important that such operations are reversible.

Even though version control by nature is intended to keep all history instead of undoing (and hence deleting) unwanted changes, it is usually not preferable to keep actual mistakes in the history. Errors in the commit message should be undoable without having to revert and re-commit the delta, for example, as should erroneous branchings and other operations that affect the repository directly without the user having to confirm the result by committing it afterwards.

Many distributed version control systems already offer this functionality as a means of undoing the last operation on the repository. This should be exposed in the GUI whenever possible, to reassure the user that the operation can always be undone and thus encouraging learning by exploration (Cooper & Reimann, 2003, p. 158). In cases when the user is about to perform something that cannot be undone, the GUI should ask for confirmation while clearly informing the user of the expected effects of the operation.

### 5.4 Putting it all together

When combining the proposed features into an actual application, how should it be done? How should screen space be allocated, and at what times should a particular component be visible? These are all questions to which the answers greatly depend on the user itself.

This thesis will avoid favoring any particular style or screen layout apart from a basic recommendation that enough flexibility be included so that the user can modify the layout himself to his liking. To increase familiarity for users of the new frontend, inspiration may be drawn from one of the existing frontends described in subsection 3.2.2.

Certain features belong to mutually exclusive “modes”, for example you won’t be committing while doing conflict resolution. However, it is up to the implementor whether these distinctions should appear as separate modes of the application (like views in the Eclipse IDE), with multiple windows, or through any other solution. It is beyond the scope of this thesis to determine the exact composition of the final frontend.
5.4.1 Rich interaction

Even though the individual features described offer much needed functionality, as with so many other things in life we are aiming for something that is more than the sum of its parts. We want each part of the frontend to interact in an intuitive, consistent manner, with the right amount of power at the user’s fingertips.

Context menus

The concept of context menus have permeated user interface guidelines in various environments and systems, and provide a sensible way to add functionality where it’s needed without intruding upon the user when not needed.

By right–clicking with the mouse on an item, a menu pops up containing relevant commands for that selection. For example, right–clicking on a file in the file listing may provide commands for comparing revisions of that file or examining its history. Right–clicking a node in the repository history may offer functionality to filter and locate other commits made by the same user.

Determining what should be put in kind of each context menu is left as an exercise to the reader, but the key feature of context menus is clearly their contextuality; be careful about adding commands that are not specific to what the user actually clicked.

Drag and drop interaction

Drag and drop is a remarkably useful idiom as it often closely mirrors actions as we mentally perceive them. For example, dragging a file to the recycle bin and dropping it there is considerably more intuitive than right–clicking the same file and selecting “send to recycle bin”. In the same way, providing drag and drop functionality selectively to our graphical frontend can provide quick and easy ways to perform basic operations.

If used correctly, the idiom can be useful for intermediate and power users as well as for beginner users. In its core it implicitly carries a notion of “source” and “destination” so any action that can be performed between any two items in those terms can be suitably implemented via drag and drop. For example, dragging a modified file onto the tip of the repository history can be assumed to mean “commit here”, while dragging a remote repository link onto the local repository history can mean “synchronize here”.

If a pair of items have multiple possible meanings in a drag and drop situation it is not always suitable to select one as a main action. In these cases, drag and drop can be combined with context menus, for example by displaying a menu at the drop target where the user can select the kind of action desired.

Even more so than context menus, drag and drop is deeply contextual. Take great care in choosing what behaviors, if any, should be supported.
through drag and drop.

5.5 Other proposals

Other features were suggested explicitly or implicitly through the usage survey, but were not included among the main proposals in this report for one reason or another. However, since the prospective frontend developer may still find it of interest, a brief mention is warranted for at least one of them.

The multitude of different distributed version control systems that are available with varying feature sets and designs can make it difficult to migrate from centralized to distributed version control. For example, which system should be used? Time spent learning one system is not necessarily useful for another system.

A suggested feature for a frontend is therefore to abstract away (or selectively expose) the differences between multiple distributed systems, letting the user use the same frontend for any of the supported systems without having to learn the particular details of the underlying system in use. However, this kind of heavy-weight feature is enough to warrant its own report and as such falls outside the scope of this thesis.
Chapter 6
Discussion

This report started with a fundamental question: how can we use graphical user interfaces to make distributed version control more accessible to users? Starting from that point, a significant part of the report was dedicated to describing some of the inner workings of version control itself. This prepared us for discussing the topic and helped to understand the question.

To help answer the question, a version control system usage survey was then designed and conducted at Opera Software. Analyzing the results showed us several things:

- Branching and merging were perceived as the biggest problems for centralized systems, but were not perceived as a problem for distributed systems. Presumably this is because branching and merging is a much easier operation under distributed version control.

- The most important tasks to assist the user with are committing, inspecting the working set, inspecting the repository history and synchronizing with remote repositories. These tasks are very frequently used, needlessly complex to use, or both.

- Popular suggested features include commit helpers, conflict/merge assistance and visual previews during remote synchronizations.

Based on the other survey results, we also inspected some of the popular frontends for existing centralized systems, trying to isolate what features made these tools popular. Armed with this information as well as having identified some of the perceived problem areas of distributed systems, we were ready to come up with ideas for our hypothetical frontend.

6.1 Evaluating proposed features

The proposed features include features borrowed from existing centralized frontends as well as new features. In order to determine whether they serve
their purpose, we compare them to the identified problems.

6.1.1 Major features

The features borrowed from existing frontends (the diff visualizer, file listing and conflict/merge assistant) are tried and tested, meaning that their implementation in a new frontend should be very polished and intuitive to use. This is important since their high use frequency means any annoyance will have large usability penalties. When implemented correctly these features should help alleviate the “inspecting the working set” problem area, but they are also important to get right since they are used in conjunction with many of the other features.

The commit helper is a direct response to “committing” having been identified as a primary problem area is partially based on the author’s own experience with problems committing from cluttered working sets. Based on the positive response to this feature when suggested in the usage survey, such a feature is likely to reduce the complexity of committing enough for it to no longer be a problem area.

The repository tree visualizer is, predictably, a response to the “inspecting the repository history” problem area, but it also fills an important purpose of being able to illustrate the repository itself to the user, giving him a tangible representation to look at and showing how his actions affect it. This can greatly reduce the learning curve for beginner users of distributed systems.

Finally, the favorite repositories list mirrors the “synchronizing with remote repositories” problem area. Having the repositories appear on–screen with clear representation gives the user a clear means of interacting with them, and also serves as a reminder to beginner users that they exist (and that the user typically wants to synchronize with them).

6.1.2 Minor features

The suggested preview functionality partially mirrors the requested “visual previews when synchronizing” feature, but the proposed feature is not limited to synchronizations only. Showing previews is not only good for helping to avoid mistakes but is also an important learning tool, helping the user transition into an intermediate user more quickly.

As similar rationale applies to the undo feature, which was not requested by any user but can be considered a logical extension of the version control concept; version control wants the user to feel safe in making changes to their content thanks to the fact that nothing is permanent and everything is reversible. Naturally this should apply to the version control systems and their frontends themselves, too!
6.2 Conclusions and future work

Based on the above summary and evaluation, we find that the proposed features are appropriate for the identified problem areas and should help alleviate the problems experienced by users.

Since the survey results easily identified several problem areas with current distributed systems, more work should go into developing usable graphical frontends for these systems, as it will likely improve the chances of users successfully migrating from older centralized systems to newer distributed systems.

The features proposed in this report should constitute a good starting point for developing a working prototype frontend, which can then be evaluated by users and refined into a final product. In addition to the main proposed features, the minor features mentioned may also warrant closer examination when making an actual prototype.
Glossary

Bazaar  A distributed version control system.

branching  A method to contain multiple parallel histories of a single file by allowing diverging histories contained in separate branches.

cherry–picking  The process of including only particular changes from a different branch onto the current one.

conflict  An error when trying to merge two branches that have both modified the same area of a file, caused by the VCS not knowing how to combine the two changes.

CVS, Concurrent Versions System  A centralized version control system.

darcs  A distributed version control system.

delta  The difference between two files (or versions of a file), usually expressed as the parts/lines that are nonidentical.

diff  The same as delta. Also a tool for creating a diff/delta.

DVCS, distributed version control system  A version control system that operates with independent peers instead of in a client–server fashion.

Git  A distributed version control system.

GUI, graphical user interface  A user interface based on graphical representations rather than on text only.

HTTP, hyper–text transfer protocol  The protocol used to transfer content on the world–wide web.

Mercurial  A distributed version control system.

merging  Re–joining two branches into a single version that contains all changes included in either branch.
**Monotone**  A distributed version control system.

**patch**  The same as delta.

**RCS, Revision Control System**  A centralized version control system.

**revision control**  The same as version control.

**SCM, source code/control management**  The same as version control [system].

**SSH, Secure Shell**  A protocol for securely exchanging data over a network (e.g. the Internet).

**ToolTip**  A small window containing additional information that appears when the user hovers the mouse over an item in a GUI.

**Unicode**  A standard for how computers can represent text in most of the world’s writing systems.

**version control**  The technique of preserving the entire history of a file instead of only the current state of it.

**VCS, version control system**  A system implementing version control.
Bibliography

*Bazaar Benchmarking Results* (2007) [www].


*BitTorrent Introduction* (2008) [www].

Branching and merging with git (2006) [www].


*ConflictsFAQ* (2008) [www].


*Google Tech Talk: Linus Torvalds on git* (2007) [www].

*HistoryOfBazaar* (2008) [www].


Further reading

The following online resources may be of use to interested readers.

Version control systems

Bazaar http://bazaar-vcs.org/
CVS http://www.nongnu.org/cvs/
darcs http://darcs.net/
Git http://git.or.cz/
Mercurial http://www.selenic.com/mercurial/
Monotone http://monotone.ca/
RCS http://www.cs.purdue.edu/homes/trinkle/RCS/
Subversion http://subversion.tigris.org/
SVK http://svk.bestpractical.com/
Graphical frontends

<table>
<thead>
<tr>
<th>Name</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td><a href="http://www.eclipse.org/">http://www.eclipse.org/</a></td>
</tr>
<tr>
<td>MarmaladeCvs</td>
<td><a href="http://www.fiffigt.com/MarmaladeCvs/">http://www.fiffigt.com/MarmaladeCvs/</a></td>
</tr>
<tr>
<td>TortoiseBzr (for Bazaar)</td>
<td><a href="http://bazaar-vcs.org/TortoiseBzr">http://bazaar-vcs.org/TortoiseBzr</a></td>
</tr>
<tr>
<td>TortoiseCVS (for CVS)</td>
<td><a href="http://www.tortoisecvs.org/">http://www.tortoisecvs.org/</a></td>
</tr>
<tr>
<td>TortoiseDarcs (for darcs)</td>
<td><a href="http://tortisedarcs.sourceforge.net/">http://tortisedarcs.sourceforge.net/</a></td>
</tr>
<tr>
<td>TortoiseHg (for Mercurial)</td>
<td><a href="http://sourceforge.net/projects/tortoisehg">http://sourceforge.net/projects/tortoisehg</a></td>
</tr>
<tr>
<td>TortoiseSVN (for Subversion)</td>
<td><a href="http://tortoisesvn.tigris.org/">http://tortoisesvn.tigris.org/</a></td>
</tr>
<tr>
<td>WinCVS</td>
<td><a href="http://www.wincvs.org/">http://www.wincvs.org/</a></td>
</tr>
</tbody>
</table>
Appendix A

Usage survey questions

This appendix contains the questions that were presented to participants of the online usage survey. The survey responses are included as appendix B.

The questions in this listing have been slightly modified from their originals. Firstly, in the original survey, all matrix questions included a separate column for the “no answer” response. This column was not included in this appendix, but such responses are properly accounted for in appendix B. Secondly, the original survey included administrative questions regarding anonymity and an option to be informed of the results. These questions are not included here for brevity.

Experience with centralized version control

These questions are meant to provide a baseline comparison with existing version control systems and their frontends. If you have no experience with centralized version control, you don’t need to answer any questions in this group.
APPENDIX A. USAGE SURVEY QUESTIONS

Q1: Please rate how often you use the following features when working with CVS/SVN:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Extensively</th>
<th>Often</th>
<th>Sometimes</th>
<th>Rarely</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking out from a repository</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Examining uncommitted data</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Synchronizing with the repository</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Committing</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Tagging</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Merging branches</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

Q2: Now please rate how difficult or time-consuming you experience the operations to be:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unusable</th>
<th>Complicated</th>
<th>Manageable</th>
<th>Easy</th>
<th>Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking out from a repository</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Examining uncommitted data</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Synchronizing with the repository</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Committing</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Tagging</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Merging branches</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td></td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>
Q3: If you use any additional tools or frontends when working with CVS/SVN, please list them here along with your reason for using them:

Additional tools may include (but are not limited to) graphical frontends, history visualizers and web interfaces.

[[ Free text answer ]]

Experience with distributed version control

These questions will help me determine some of the requirements for a DVCS graphical frontend. If you have no experience with distributed version control, you don’t need to answer any questions in this group.

Q4: Which distributed version control systems do you have experience with?

<table>
<thead>
<tr>
<th></th>
<th>Strong</th>
<th>Average</th>
<th>Some</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bazaar</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>BitKeeper</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>darcs</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Git</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Mercurial</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Monotone</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>Other</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>
APPENDIX A. USAGE SURVEY QUESTIONS

Q5: Please rate how often you perform the following operations when working with your DVCS:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extensively</th>
<th>Often</th>
<th>Sometimes</th>
<th>Rarely</th>
<th>Never</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating new repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding or removing files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Querying status of working set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Committing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cherry-picking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronizing with remote repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Examining history of changes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q6: Now please rate how difficult or time-consuming you experience the operations to be:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unusable</th>
<th>Complicated</th>
<th>Manageable</th>
<th>Easy</th>
<th>Transparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating new repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding or removing files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Querying status of working set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Committing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cherry-picking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronizing with remote repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Examining history of changes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q7: In addition to the question above, can you try to identify and describe some specific weaknesses in the DVCS you’ve used?

To help you get started, you may consider:
- Working against multiple remote repositories
- Remembering the current working set
- Intuitiveness of branching and merging
- Undoing mistakes

Weaknesses may include unnecessarily complex design, inadequate information presented to the user or any other perceived flaws.

[[ Free text answer ]]

Q8: Would you consider any of the following features useful in a graphical frontend to your DVCS?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Must-have</th>
<th>Very useful</th>
<th>Useful</th>
<th>Barely useful</th>
<th>Useless</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree visualization of the repository history</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graphical diffs between versions or for uncommitted data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remembering remote repositories with bookmarks/favorites</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drag-and-drop interaction for committing/branching/merging</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context menus for all applicable areas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better control of commits by letting the user select which changes to include</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q9: Are there any other specific features you can think of that you would consider truly useful in a graphical frontend?

In particular, can you think of a way for a graphical frontend to address or make up for the weaknesses identified in the earlier questions?

[[ Free text answer ]]
Appendix B

Usage survey results

This appendix lists all responses to the usage survey in a collected form. The responses to matrix questions related to usability are presented in graph form, with a partially filled circle for each possible response. The degree to which a circle is filled indicates the percentage of participants who chose that response:

This form of graph was chosen since it visualizes the distribution of responses to the kind of matrix question used in this survey without requiring much space. The reader can thus perceive the distribution of responses of all rows at a glance.

The participants’ experience with various distributed systems is described as two bar graphs; one system-centric (describing the relative popularity of each system and the users’ familiarity with them) and one user-centric (describing the number of systems each user is familiar with).

Finally, the responses to free text answers are presented in list form, sorted by the number of mentions. Some editing has been done by the author by combining responses that were deemed to be the same.

The answers to the survey questions are presented in the same order as the questions in appendix A.
APPENDIX B. USAGE SURVEY RESULTS

Experience with centralized version control

Q1: Please rate how often you use the following features when working with CVS/SVN:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Extensively</th>
<th>Often</th>
<th>Sometimes</th>
<th>Rarely</th>
<th>Never</th>
<th>No answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking out from a repository</td>
<td>20%</td>
<td>35%</td>
<td>25%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td>10%</td>
<td>35%</td>
<td>35%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Examining uncommitted data</td>
<td>75%</td>
<td>15%</td>
<td>5%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Synchronizing with the repository</td>
<td>70%</td>
<td>25%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Committing</td>
<td>50%</td>
<td>40%</td>
<td>10%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Tagging</td>
<td>15%</td>
<td>35%</td>
<td>45%</td>
<td>5%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Branching</td>
<td>5%</td>
<td>30%</td>
<td>40%</td>
<td>20%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Merging branches</td>
<td>0%</td>
<td>25%</td>
<td>25%</td>
<td>45%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td>10%</td>
<td>40%</td>
<td>25%</td>
<td>25%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td>35%</td>
<td>55%</td>
<td>10%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>
Q2: Now please rate how difficult or time-consuming you experience the operations to be:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unusable</th>
<th>Complicated</th>
<th>Manageable</th>
<th>Easy</th>
<th>Transparent</th>
<th>No answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking out from a repository</td>
<td>0%</td>
<td>15%</td>
<td>15%</td>
<td>55%</td>
<td>15%</td>
<td>0%</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td>5%</td>
<td>0%</td>
<td>35%</td>
<td>55%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Examining uncommitted data</td>
<td>0%</td>
<td>15%</td>
<td>15%</td>
<td>60%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Synchronizing with the repository</td>
<td>0%</td>
<td>20%</td>
<td>45%</td>
<td>30%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Committing</td>
<td>0%</td>
<td>0%</td>
<td>30%</td>
<td>55%</td>
<td>15%</td>
<td>0%</td>
</tr>
<tr>
<td>Tagging</td>
<td>0%</td>
<td>25%</td>
<td>45%</td>
<td>20%</td>
<td>10%</td>
<td>0%</td>
</tr>
<tr>
<td>Branching</td>
<td>0%</td>
<td>30%</td>
<td>50%</td>
<td>15%</td>
<td>5%</td>
<td>0%</td>
</tr>
<tr>
<td>Merging branches</td>
<td>20%</td>
<td>60%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td>15%</td>
<td>50%</td>
<td>20%</td>
<td>10%</td>
<td>0%</td>
<td>5%</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td>15%</td>
<td>15%</td>
<td>30%</td>
<td>25%</td>
<td>15%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Q3: If you use any additional tools or frontends when working with CVS/SVN, please list them here along with your reason for using them:

3+ mentions: Opera-internal web-based CVS log (9), MarmaladeCvs (4), Emacs (3), viewcvs (3), WinCVS (3)

1–2 mentions: TortoiseSVN (2), WinMerge (2), direct CVS repository manipulation, Eclipse, KDiff3, Meld, Quilt, svnmerge, svnview, TortoiseCVS
Experience with distributed version control

Q4: Which distributed version control systems do you have experience with?

<table>
<thead>
<tr>
<th>System</th>
<th>Some</th>
<th>Average</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bazaar</td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitKeeper</td>
<td>5%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>darcs</td>
<td></td>
<td>30%</td>
<td></td>
</tr>
<tr>
<td>Git</td>
<td></td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>Mercurial</td>
<td></td>
<td></td>
<td>45%</td>
</tr>
<tr>
<td>Monotone</td>
<td></td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>10%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the answers, the following overlap was also calculated. The graph describes how many systems each respondent indicated they had experience with.
Q5: Please rate how often you perform the following operations when working with your DVCS:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Extensively</th>
<th>Often</th>
<th>Sometimes</th>
<th>Rarely</th>
<th>Never</th>
<th>No answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating new repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adding or removing files</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Querying status of working set</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Committing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merging branches</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cherry-picking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchronizing with remote repositories</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Examining history of changes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q6: Now please rate how difficult or time-consuming you experience the operations to be:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unusable</th>
<th>Complicated</th>
<th>Manageable</th>
<th>Easy</th>
<th>Transparent</th>
<th>No answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating new repositories</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>30%</td>
<td>20%</td>
<td>40%</td>
</tr>
<tr>
<td>Cloning existing repositories</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>40%</td>
<td>20%</td>
<td>40%</td>
</tr>
<tr>
<td>Adding or removing files</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>35%</td>
<td>25%</td>
<td>40%</td>
</tr>
<tr>
<td>Querying status of working set</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>25%</td>
<td>25%</td>
<td>40%</td>
</tr>
<tr>
<td>Committing</td>
<td>0%</td>
<td>0%</td>
<td>15%</td>
<td>15%</td>
<td>35%</td>
<td>35%</td>
</tr>
<tr>
<td>Branching</td>
<td>0%</td>
<td>0%</td>
<td>5%</td>
<td>20%</td>
<td>30%</td>
<td>45%</td>
</tr>
<tr>
<td>Merging branches</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>30%</td>
<td>10%</td>
<td>50%</td>
</tr>
<tr>
<td>Cherry-picking</td>
<td>0%</td>
<td>5%</td>
<td>25%</td>
<td>10%</td>
<td>5%</td>
<td>55%</td>
</tr>
<tr>
<td>Synchronizing with remote repos</td>
<td>0%</td>
<td>0%</td>
<td>10%</td>
<td>35%</td>
<td>10%</td>
<td>45%</td>
</tr>
<tr>
<td>Examining history of changes</td>
<td>0%</td>
<td>10%</td>
<td>0%</td>
<td>35%</td>
<td>10%</td>
<td>45%</td>
</tr>
</tbody>
</table>

Q7: In addition to the question above, can you try to identify and describe some specific weaknesses in the DVCS you’ve used?

**Multiple mentions:** Lack of graphical user interfaces (3), differing design decisions between systems (2), difficulties importing/exporting data between different systems (2), unintuitive branch behavior (2), weak support for cherry-picking (2)

**Single mentions:** Complicated synchronization, conflict resolution problems, difficulties getting proper changelogs, difficulties merging from the command-line, weak OS support (Microsoft Windows)
Q8: Would you consider any of the following features useful in a graphical frontend to your DVCS?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Must-have</th>
<th>Very useful</th>
<th>Useful</th>
<th>Barely useful</th>
<th>Useless</th>
<th>No answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree visualization of the repository history</td>
<td>15%</td>
<td>35%</td>
<td>30%</td>
<td>0%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>Graphical diffs between versions or for uncommitted data</td>
<td>45%</td>
<td>20%</td>
<td>10%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
</tr>
<tr>
<td>Remembering remote repositories with bookmarks/favorites</td>
<td>5%</td>
<td>25%</td>
<td>45%</td>
<td>5%</td>
<td>5%</td>
<td>15%</td>
</tr>
<tr>
<td>Drag-and-drop interaction for committing/branching/merging</td>
<td>0%</td>
<td>10%</td>
<td>35%</td>
<td>20%</td>
<td>10%</td>
<td>25%</td>
</tr>
<tr>
<td>Context menus for all applicable areas</td>
<td>10%</td>
<td>20%</td>
<td>35%</td>
<td>5%</td>
<td>5%</td>
<td>25%</td>
</tr>
<tr>
<td>Better control of commits by letting the user select which changes to include</td>
<td>25%</td>
<td>40%</td>
<td>20%</td>
<td>0%</td>
<td>0%</td>
<td>15%</td>
</tr>
</tbody>
</table>

Q9: Are there any other specific features you can think of that you would consider truly useful in a graphical frontend?

**Answers:** Simplified conflict/merge handling (2), visual preview of synchronization (2), full integration into editor of choice, graphical cherry-picking
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera åktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Kim Nilsson

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-15808