Why Are Software Projects Moving From Centralized to Decentralized Version Control Systems?

Brian de Alwis
Dept of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada
brian.de.alwis@usask.ca

Jonathan Sillito
Dept of Computer Science
University of Calgary
Calgary, AB, Canada
sillito@ucalgary.ca

Abstract

*Version control systems are essential for co-ordinating work on a software project. A number of open- and closed-source projects are proposing to move, or have already moved, their source code repositories from a centralized version control system (*CVCS*) to a decentralized version control system (*DVCS*). In this paper we summarize the differences between a* CVCS *and a* DVCS*, and describe some of the rationales and perceived benefits offered by projects to justify the transition.*

1. Introduction

For many projects, their version control system (VCS), along with other tools such as the issue tracking system, is central to how development work is organized. The central challenge in managing software development is scaling the change process up to large numbers of possibly geographically-distributed software developers without sacrificing quality or introducing undue overhead. To a large extent these tools determine how easily people can contribute to a project, how new feature development is co-ordinated, how often separate development lines are merged, how code is reviewed and how the support of already released code is organized. Despite their importance, the impact these tools can have on the development work and the trade-offs involved is not yet well studied.

A new generation of VCS, called *decentralized* VCSs (DVCS), have emerged that address some of the limitations of current *centralized* VCSs (CVCS), such as CVS [1, 5] and Subversion [4], to better support decentralized workflows. Some of these new DVCSs, such as GIT,[1] MERCURIAL,[2] BZR,[3] and BITKEEPER,[4] have become sufficiently mature that many open- and closed-source projects are proposing to move, or have already moved, their source code repositories to a DVCS.

As part of a larger research project to explore practices and tool support around version management, we have begun a qualitative study to answer two research questions. First, what do these projects see as the benefits of a DVCS? Transitioning a source code repository to a new VCS requires significant effort,[5] and so we suppose that there must be compelling reasons for switching. Second, what changes have these projects made to their development processes in making the switch? To answer these questions, we examined publicly-available documents and mailing list discussions for four open-source projects (Perl, OpenOffice, NetBSD, Python) that have moved or are contemplating moving to a DVCS to identify rationales and perceived benefits offered by projects to justify such a transition. In this paper we present some initial observations from our ongoing analysis.

This paper is structured as follows. In Section 2, we provide some background, summarizing the uses of VCSs and a brief explanation of what makes a DVCS different from a CVCS. In Section 3 we summarize the anticipated benefits, and possible issues reported by several open-source projects that have made or are considering making a transition. We finally conclude in Section 4.

2. Background

In this section we provide a brief description of both centralized and decentralized VCSs, and highlight the open-source projects examined. Note that as the various DVCS

[1]git-scm.com; retrieved 2009/01/20
[2]www.selenic.com/mercurial/; retrieved 2009/01/20
[3]bazaar-vcs.org; retrieved 2009/01/20
[4]bitkeeper.com; retrieved 2009/01/20
[5]For example, the Perl Foundation reported that their transition to GIT took approximately 21 months, and involved significant manual work [9].

implementations are still undergoing rapid evolution, a detailed feature-by-feature comparison of the current tools would be quickly obsolete. Rather we compare the conceptual differences between CVCSs and DVCSs, and instead refer interested readers to Raymond's draft survey of VCSs for detailed comparisons [8].

2.1. Centralized VCSs

The most commonly-used version control system (VCS) used today are *centralized* VCSs, as typified by CVS [1, 5] and Subversion [4]. These VCSs are centralized as they have a single canonical source repository. All developers work against this repository through a *checkout* taken from the repository, essentially a snapshot at some moment in time.

Write-access to a group's repository is generally restricted to a set of known developers, or *committers*. In open-source situations, committers generally earn and maintain write-access through the submission of high-quality patches to demonstrate programming prowess and understanding of the project development conventions. Teams generally establish coding conventions and practices to control how changes are made to the repository to preserve the quality of the source code.

Modern VCSs support parallel evolution of a repository's contents through the use of *branches*. One wide-spread practice is to maintain a *mainline* branch to represent the current development efforts, and creating new branches of the mainline to represent released versions of the product and track bug fixes to the released product [11]. Branches are also often used for undertaking a substantial change of some long duration, with the goal of being merged back into the mainline.

2.2. Decentralized VCSs

DVCSs relax the requirement of CVCSs to have a central, master repository. With a DVCS, each checkout is itself a first-class repository in its own right, a copy containing the complete commit history. Write-access is no longer an issue as each developer is a first-class committer to their personal repository, regardless of whether they are an accepted committer to the project.

DVCSs maintain sufficient information to support easy branching and easy, repeated merging of branches The ease of branching has encouraged a practice called *feature branches*, where every prospective change is done within a branch and then merged into the mainline, rather than being directly developed against the mainline.

Because each DVCS repository is a full-fledged repository, there are no enforced master branches. Instead a canonical branch is identified by convention within a development group or community, and some projects may have several principal branches. For example, the Linux kernel's authoritative branch is Linus Torvalds' branch,[6] but there are also several other important branches, such as Andrew Morton's `-mm` branch, that are used for staging and evaluating proposed changes to the kernel. Once the patches in these intermediate branches have been stabilized and proved their value, Torvalds will select the result to merge to his master tree.

Since the DVCS repositories contain all the revision history, they lend themselves very well to distributed and disconnected development. Indeed this was the original motivation for the first recorded DVCS, Reliable Software's Code Co-op, introduced in 1997 [7]. Reliable Software needed to cater to teams developing across distributed locations, where the latency to access a central repository was prohibitive [7]. Using a DVCS naturally leads to the source code repository being replicated to a number of places, a side-benefit that reduces the risk of some disaster scenarios.

2.3. The Projects

We are in the process of analyzing public documents and mailing-lists from four open source projects.

Perl: The Perl Foundation recently completed switching their code repository to GIT. The Perl source code was previously maintained using the Perforce VCS, hosted by a company named ActiveState, who also provided free Perforce licenses to Perl committers. We collected data from several wiki pages dedicated to planning the transition and several archived discussion threads. Despite this, it is not yet clear to us how the decision to switch to GIT was made.

OpenOffice: OpenOffice is a large open-source project coordinated by Sun Microsystems. The OpenOffice source code was previously maintained using CVS, and the project had been actively planning to move to a new VCS to address accumulated problems arising from their use of CVS. The project was prematurely forced to switch to Subversion as "*the* [OpenOffice CVS repository] *is crumbling under the heavy weight of 8 years worth of OOo coding*," a side-effect of the 3000 branches resulting from their development process' heavy use of branching. Branching and tagging is a heavyweight and non-atomic operation in CVS that modifies every file in the repository to insert a reference to the branch or tag. The project plans to re-evaluate their choice of VCS in 2009. We collected data from several archived discussion threads, and wiki pages describing the current development processes, empirical assessments of candidate VCSs, and transition planning.

Python: The Python Software Foundation is considering switching their code repository from Subversion to a DVCS. The data we collected about the Python team's decision

[6]git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git

making process came from one extensive document, a Python Enhancement Request (PEP), and several archived discussion threads. The PEP is central to the foundation's decision making process and revolves around an exploration of how effectively various tools support a series of project relevant "usage scenarios." The usage scenarios and tools to consider were discussed on the development team's mailing list. No decision has yet been made and the PEP is currently under active development [2].

NetBSD: The NetBSD Project has maintained its source code in a CVS repository since its inception in 1993. Some NetBSD developers have suggested that as an increasing number of other open-source projects are switching from CVS, the project should also consider switching to using a new VCS. A switch would also provide an opportunity to address frustrations arising from use of CVS, particularly the difficulties in managing branches in CVS. Ongoing discussions the potential transition are being carried out on a public mailing list. The project is very concerned that any switch must preserve the full history of the project.

3. Findings

3.1. Anticipated Benefits

To provide first-class access to all developers: A key reason given for many projects for moving to a DVCS was to improve support for non-committers. Contributors without commit access can not get benefit from the CVCS for their work, and often resort to creating parallel repositories to manage larger changes. In the Python discussion this issue was highlighted as the most important limitation of CVCSs as "anyone writing a patch for Python or creating a custom version, [does] not have direct tool support for revisions." The Perl Foundation placed particular emphasis on using an open-source DVCS to ensure that the tools were available to all members of the community. The OpenOffice development process, based on using CVS branches, requires the developers to have write access to the repository.

In a DVCS, each contributor has their own repository and so they can "incrementally save their progress to make their development lives easier." Torvalds (the original author of GIT) argues that this feature of distributed systems removes much of the politics projects face around granting (and revoking) commit access to the central repository [10]. This support appears to be particularly important when contributors are making significant changes or when there is a review process that requires contributors to make repeated revisions to their submissions.

To support atomic changes: The NetBSD and OpenOffice projects most pressing requirement of a new VCS was to have repository-wide atomic commits. Both groups have encountered repository corruption from using CVS branches, which may go undetected.

Simple automatic merging: DVCSs maintain sufficient information to support automatic and repeated merges, as often occurs with long-lived branches. The Python project saw this as an important feature of a DVCS for two reasons. First, improved support for merges encourages developers to keep their branches up-to-date with the mainline development, and reduces the risk of their branch becoming stale or out-of-date. Second, improved support for merging reduces the burden on committers, also mentioned as important by the Perl project. Exchanging plain patches risks suffering from version mismatches should the patch originator's version of the repository differ from that of the patch examiners. Patches generated by a DVCS include sufficient dependency information to identify whether the patch depends on other unsubmitted revisions, thus reducing the work required for diagnosing badly submitted patches. This dependency information is also useful to identify common ancestors, which leads to improved merging of patches.

Improved support for experimental changes: The Perl and OpenOffice projects sought to enhance the ability for non-committers to work on experimental changes before submitting changes for incorporation. With cheap branches, developers can make local commits to snapshot a work-in-progress, which is often desirable before embarking on some experimental work with an uncertain outcome.

Support disconnected operation: The Python project in particular sought to support disconnected operation, a feature that is useful for developers while travelling by air. The CVCS tools require that the developer be able to connect to the server to access or query against the repository. This disconnected nature provides for separating the act of committing a change, such as to make a snapshot, from publishing a change for others to view [3].

3.2. Transition and Challenges

During the the Perl team's discussion, the work required for transitioning to their new VCS was discussed. For example, one contributor joked that if they did not revise their development tutorial documentation with the new workflows then "all committing is going to stop." The work involved in transitioning is significant and so we suppose that the reasons for switching must be compelling.

One of the challenges in making this transition is the possibility of changing the teams' development processes. Some teams were more open to this than others. When evaluating different VCSs, the OpenOffice team, for example, considered easy integration into their current (extensive) development processes and tools as very desirable. On the other hand the Python team specifically documented what they expected to be their new development approach once the transition has been made. We are interested in seeing how these development processes will change with the new possibilities opened by a new VCS.

The NetBSD and Perl developers were concerned that metadata from their previous VCS be somehow transferred to a new VCS. This metadata, such as file version numbers, are embedded and referenced from commits and other documents. In switching to GIT, Perl prefixed each commit with a special header containing the corresponding Perforce version number of the form "`p4raw-id:NNNN`". Additional effort was dedicated to managing authorship attributions as this is used for calculating developer metrics by sites such as ohloh.net.

Several NetBSD developers expressed a deep preference for human-meaningful commit identifiers (e.g., monotonically increasing version numbers), a property that the SHA1-based commit identifiers used by some DVCSs cannot satisfy. These identifiers are often used in e-mails sent to notify interested parties of particular bug fixes. This problem may be partially addressable through the support for tagging a particular revision with a symbolic name. Tags may not always be suitable in a DVCS, however. In a DVCS, a tag only becomes visible to other repositories as those repositories are updated from the tagged repository; thus for a tag to serve as a stable, global reference to a revision, the tag must be accepted by the canonical repositories. Indiscriminate tagging may also lead to pollution of the available tag space, although it is not clear to what degree this is considered a problem.

That each DVCS repository is a complete copy has some ramifications. The settlement of a set of lawsuits from 1992 between the UNIX System Laboratories vs. The Regents of the University of California and Berkeley Software Design Inc. required that certain source code files be expunged from the NetBSD repository [6]. Complying with such a settlement would be impossible with a DVCS: once a change has been published to a publicly available repository, the change may have been duplicated to a number of locations.

Finally, DVCSs involve a significant change to most developers' models of managing repositories. The various projects have a significant challenge in selling the rationale of a transition to their developers, and creating tutorials and transition documents to address the learning curve.

4. Summary and Future Work

DVCSs have captured a large mindshare, and many projects are at least debating the merits of transitioning their code repositories to a DVCS. Our study has provided some insights into the limitations of CVCSs and the consequences those limitations have for development teams. Although DVCSs address some of the problems of CVCSs, particularly the difficulty in repeated merges of branches, our findings suggest that DVCS may also introduce new issues.

So far we have only explored what team members *believe* the impact of transitioning to a DVCS will be. These beliefs have lead us to form several hypotheses (e.g., non-committers will make more significant contributions in a project with a DVCS) that we would like to explore further. Our next step in this research will be to conduct a survey of developers that have experience with both DVCSs and CVCSs. We are interested in asking questions such as: Has changing to a DVCS truly reduced barriers to participation? How have their development processes changed as a result of the DVCS? What new complications have been introduced by using a DVCS? What recommendations would they make to other projects considering switching to a DVCS? Under what circumstances would they advise against making such a change? Are there differences in DVCS use between open- and closed-source development?

References

[1] B. Berliner. CVS II: Parallelizing software development. In *Proc. USENIX Winter 1990 Technical Conference*, pages 341–352, Berkeley, USA, 1990. USENIX Association.

[2] B. Cannon, B. Warsaw, S. J. Turnbull, and A. Vassalotti. Migrating from Subversion to a distributed VCS. PEP 0374, Python Foundation, draft. URL http://www.python.org/dev/peps/pep-0374/. Retrieved 2009/01/16.

[3] I. C. Clatworthy. Distributed version control: Why and how. In *Proc. Open Source Development Conf. (OSDC)*, 2007.

[4] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion (for Subversion 1.5)*. O'Reilly, 2 edition, 2008.

[5] D. Grune. Concurrent Versions Systems: A method for independent cooperation. Technical Report IR 113, Vrije Universiteit, 1986.

[6] P. Jones. The 1994 USL–Regents of UCal settlement agreement. Groklaw, Nov. 2004. URL http://www.groklaw.net/articlebasic.php?story=20041126130302760.

[7] B. Milewski. Distributed source control system. In *Proc. ICSE Worksh. on System Configuration Management (SCM-7)*, pages 98–107, 1997.

[8] E. S. Raymond. Understanding version-control systems. Retrieved 2009/01/17 from http://www.catb.org/~esr/writings/version-control/, draft.

[9] The Perl Foundation. Perl 5 now uses git for version control, Dec. 2008. URL http://use.perl.org/articles/08/12/22/0830205.shtml.

[10] L. Torvalds. Linus torvalds on git. Transcript from Google Tech Talk, May 2007. URL http://git.or.cz/gitwiki/LinusTalk200705Transcript.

[11] L. Wingerd and C. Seiwald. High-level SCM best practices. In *System Configuration Management*, volume 1439 of *LNCS*, pages 57–66. Springer, 1998.