Making Sense of Revision-control Systems

Whether distributed or centralized, all revision-control systems come with complicated sets of tradeoffs. How do you find the best match between tool and team?

Bryan O'Sullivan

Modern software is tremendously complicated, and the methods that teams use to manage its development reflect this complexity. Though many organizations use revision-control software to track and manage the complexity of a project as it evolves, the topic of how to make an informed choice of revision-control tools has received scant attention. Until fairly recently, the world of revision control was moribund, so there was simply not much to say on this subject.

The past half-decade, however, has seen an explosion of creativity in revision-control software, and now the leaders of a team are faced with a bewildering array of choices.

CVS (Concurrent Versions System) was the dominant open source revision-control system for more than a decade. While it has a number of severe shortcomings, it is still in wide use as a legacy system. Subversion, which was written to supplant CVS, became popular in the mid-2000s. (A notable commercial competitor to Subversion is Perforce.) Both Subversion and CVS follow the client-server model: a single central server hosts a project's metadata, and developers "check out" a limited view of this data onto the machines where they work.

In the early 2000s, several projects began to move away from the centralized development model. Of the initial crop of a half-dozen or so, the most popular today are Git and Mercurial. The distinguishing feature of these distributed tools is that they operate in a peer-to-peer manner. Every copy of a project contains all of the project's history and metadata. Developers can share changes in whatever arrangement suits their needs, instead of through a central server.

Whether centralized or distributed, a revision-control system allows members of a team to perform a handful of core tasks:

- It allows a team to track the history of the files they work on during the development of a project. People can see who made a change; understand when and why it was made; inspect the details of the change; and re-create the state of the project at the time the change was made.
- People can work on independent subprojects without being disturbed by other people's changes and without affecting the work of their colleagues. These self-contained lines of development are usually referred to as *branches*. Branches are also used to manage the maintenance of releases that are no longer actively developed.
- When the work on a subproject is complete, it can be integrated back into the larger project. This is referred to as *merging*.

Each revision-control tool emphasizes a distinct approach to working and collaboration. This in turn influences how a team works. As a result, no revision-control tool will suit every team: each tool comes with a complicated set of trade-offs that can be hard even to see, much less to evaluate.

BRANCHES AND MERGING: BALANCING SAFETY AND RISK

On a large project, managing concurrent development is a substantial sticking point. Developers are sadly familiar with progress on their feature being stalled by a bug in an unrelated module, so they prefer to manage this risk by working in isolated branches. When a branch is sequestered for too long, a different kind of risk arises: that of teams working in different branches making conflicting changes to the same code.

Merging changes from one branch into another can be frustrating and dangerous—one that can silently reintroduce fixed bugs or create entirely new problems. These risks can arise in several ways:

- Developers working in separate branches may modify the same sections of one or more files in different ways. A revision-control system will identify these sections as conflicts that need to be resolved by hand. Whoever resolves the conflict must choose one branch's version, the other, or a hybrid.
- Code in one branch may depend on functionality that has changed in the other branch. In many cases, this dependency will be obvious: it will lead to a broken build. Sometimes the effects can be much more insidious, causing an unanticipated kind of failure.
- Some systems do not cope well if files have been renamed or copied in one branch but modified under their old names in another. (These are more often bugs than fundamental deficiencies, but longstanding bugs are important in their own right.)

Since merges introduce risk beyond the sort that normal development incurs, how a revision-control system handles both branches and merges is of great importance. Under Subversion, creating a new branch is a matter of making a copy of an existing branch, then checking out a local view of it. Although branches are relatively cheap to create, Subversion allows several developers to work concurrently in a single branch. Since working out of a single branch carries no immediately obvious costs, most teams maintain few branches.

This mode of work introduces a new risk. Suppose Alice and Bob are concurrently working on the same files in a single branch. Subversion treats the history of a branch as linear: revision 103 follows revision 102 and precedes revision 104. Alice and Bob have each checked out a copy of revision 105 of the branch from the server onto their own laptops. These working copies contain their uncommitted work, isolated from each other until one commits his or her changes.

If Alice commits her work first, it will become revision 106. Subversion will not allow Bob to commit his work as revision 107 until he has merged his work with Alice's revision 106. Since Bob cannot commit his work, what will happen if something goes wrong with his merge? He will have no permanent record of what he did and faces some scary possibilities: his work might be lost or quietly corrupted. Because Subversion offers working out of a shared branch as the path of least resistance, developers tend to do so blindly without understanding the risk they face. In fact, the risks are even subtler: suppose that Alice's changes do not textually conflict with Bob's; she will not be forced to check out Bob's changes before she commits, so she can commit her changes to the server unimpeded, resulting in a new tree state that no human has ever seen or tested.

Mercurial and Git are distributed, so they lack Subversion's concept of a single central server where metadata is hosted. A repository contains a stand-alone copy of a project's complete history and a working directory that contains a snapshot of the project's files. If Alice and Bob are working together on a project, Alice might clone a copy of Bob's repository, or she could clone a copy from some server. When she commits a change, it stays local to her repository on her machine until she

chooses to share it somehow. She could do this by publishing it to a server or by letting Bob pull it directly from her.

Both Mercurial and Git decouple fetching remote changes from merging them with local changes. If Bob fetches Alice's revisions, he can still commit his changes without needing to merge with hers first. When he merges afterward, he will still have a permanent record of his committed changes. If the merge runs into trouble, he will be able to recover his earlier work.

Under the distributed view of revision control, every commit is potentially a branch of its own. If Bob and Alice start from the exact same view of history, and each one makes a commit, they have already created a tiny anonymous fork in the history of the project. Neither will know about this until one pulls the other's changes in, at which point they will have to merge with them.

These tiny branches and merges are so frequent with Mercurial and Git that users of these tools look at branching and merging in a very different way from Subversion users. The parallel and branchy nature of a project's development is clearly visible in its history, making it obvious who made which changes when, and exactly which other changes theirs were based upon. Both Mercurial and Git can associate names with longer-lived lines of development (e.g., "the code that will become version 2.0"), so a development that is important enough to deserve a name can have one.

DEGREES OF FREEDOM

It is instructive to take a look at where Subversion and the distributed tools give users degrees of freedom. Subversion imposes almost no structure on the hierarchy of files and directories that it manages. It lacks the concept of a branch, beyond what it provides via the `svn copy` command. Users find branches by convention in a portion of the hierarchy where people agree that branches ought to live. By convention, a single "main line of development" is called `/trunk`, and branches live under `/branches`.

Since Subversion doesn't enforce a policy for structuring branches, it has some interesting behaviors. To perform an operation across an entire branch, you have to know where in the namespace the root of the branch is. Most Subversion commands operate only on whatever portion of the namespace they are told to. If Alice has checked out `/branches/myfeature` and runs `svn commit` in her working copy of `/branches/myfeature/deep/sub/dir`, she will commit changes only in and beneath the `deep/sub/dir` directory of the branch. An absent-minded commit from the wrong directory can leave Alice thinking that everything is fine but leave her colleagues with an inconsistent, broken tree.

The `svn update` command operates in the same way: it is possible to have portions of a working copy synced up to different revisions of a branch's history. This can easily lead to a working copy looking inconsistent when in fact it is accidentally composed of fragments from different times in a branch's history.

In contrast, the distributed tools treat the entire contents of a repository as the unit to work with. If you run `git commit -a` in any directory inside a repository, then it will take a snapshot of all outstanding changes. With Mercurial, `hg update` operates similarly, bringing the entire working directory up to date with respect to a specific point in history. Neither tool makes it possible to check out an inconsistent view of a branch accidentally. If you manually revert a file or directory to some specific revision, the user interfaces make this clear by displaying the affected files as modified.

PUBLISHING CHANGES

Even though Subversion does not impose a structure on projects that use branches, it suggests a convention for naming branches. Thus, Subversion users who collaborate through a central server are likely to have an easy time finding each other's projects. Both Mercurial and Git make it fairly easy to publish a read-only repository on a server, but the repository's owner has to tell other people where the repository is: it could be anywhere on the Internet, not merely a well-known location on a single server host. In addition, neither system makes read-write publishing especially easy. This is by design.

Subversion's single-server model demands that collaborators who want to share changes with other people must have write access to the shared repository, so that they may publish their changes. With Git and Mercurial, it is certainly possible to follow this centralized model, but this is a matter of convention. Users often host their repositories on their own servers or with a hosting provider. Instead of publishing their changes to a shared server, their collaborators pull changes from them and publish their own modifications elsewhere.

The major difference between Subversion and the distributed tools is this: with Subversion, committing a change implicitly publishes it, whereas with the distributed tools, the two are decoupled. Combining committing with publishing is convenient in settings where all participants have write access to the server and where everyone is always connected to the same network. Separating the two adds an extra publication step but opens up the possibilities of working offline and using novel publication techniques.

For an example of novel publication, Mercurial supports ad hoc publication of repositories over a LAN using its built-in Web server, and it supports discovery of repositories using the Bonjour protocol. This is a potent combination for rapid development settings such as a software project's sprint: just open your laptop, share your repositories, and your Wi-Fi neighbors can find and pull your changes immediately, with no server infrastructure required.

Both the centralized and distributed approaches to publication offer trade-offs. With a small, tightly knit team that is always wired, commit-as-publish can look like an easier choice. In a more loosely organized setting—for example, where team members travel or spend a lot of time at customer sites—the decoupling of commit from publication may be a better fit.

Centralized tools can be a good fit for highly structured “rule the team with an iron fist” models of management. Access can be controlled by managers, not peers. Whole sections of the tree can be made writable or readable only by employees with specific levels of clearance. Decentralized systems don't currently offer much here other than the ability to split sensitive data into separate repositories, which is a touch awkward.

THE PULL MODEL OF DEVELOPMENT

Many teams begin using a distributed revision-control system in almost exactly the same way as the centralized system they are replacing. Everyone clones one of a few central repositories and pushes the changes back. This familiar model works well for getting comfortable, but it barely scratches the surface of the possible styles of interaction.

Since the distributed model emphasizes pulling changes into a local repository, it naturally fits well with a development model that favors code review. Suppose that Alice manages the repository that will become version 2.4 of her team's software project. Bob tells her that he has some changes

ready to submit and gives her the URL from which she can pull his changes. When she reads through his changes, she notices that his code doesn't handle error conditions correctly, so she asks him to revise his work before she will accept, merge, and publish it.

Of course, a team may agree to use a "review before merge" policy with a centralized system, but the default behavior of the software is more permissive. Therefore, a team has to take explicit steps to constrain itself.

MERGES, NAMES, AND SOFTWARE ARCHAEOLOGY

Given their backgrounds, it is no surprise that Mercurial and Git have similar approaches to merging changes, whereas Subversion does things differently.

Since merges occur so frequently with Mercurial and Git, they have well-engineered capabilities in this realm. The typical cases that trip up revision-control systems during merges are files and directories that have been renamed or deleted. Both Mercurial and Git handle renames cleanly.

Subversion's merge machinery is complicated and fragile. For example, files that had been renamed used to disappear in merges. This severe bug has been partly addressed so that files are now renamed, but they may contain the wrong contents. It is not clear that this is really a step forward.

A subtler problem with file naming often hits cross-platform development teams. Windows, OS X, and Unix systems have different conventions for handling the case of file names (i.e., different answers to the question of whether FOO.TXT is the same name as foo.txt). Mercurial outshines its competition here. It can detect—and work safely with—a case-insensitive file system that is being used on an operating system that is by default sensitive to case.

Often, a developer's first response to receiving a new bug report will be to look through a project's history to see what has changed recently or to annotate the source files to see who modified them and when. These operations are instantaneous with the distributed tools, because all the data is stored on a developer's computer, but they can be slow when run against a distant or congested Subversion server. Since humans are impatient creatures, extra wait time will reduce the frequency with which these useful commands are run. This is another way in which responsiveness has a disproportionate effect on how people use their software.

A POWERFUL NEW WAY TO FIND BUGS

Although a simple display of history is useful, it would be far more interesting to have a way of pinpointing the source of a bug automatically. Git introduced a technique to do so via the `bisect` command (which proved so useful, Mercurial acquired a `bisect` command of its own). This technique is trivial to learn: you use the `bisect` command on a revision that you know did not have the bug, and the revision that you know does have the bug. It then checks out a revision and asks you whether that revision contains the bug; it repeats this until it identifies the revision where the bug first arose.

This is appealing to developers in part because it is easy to automate. Write a tiny script that builds your software and tests for the presence of the bug; fire off a `bisect`; then come back later and find out which revision introduced the problem, with no further manual intervention required. The other reason that `bisect` is appealing is that it operates in logarithmic time. Tell it to search a range of 1,000 revisions, and it will ask only about 10 questions. Widen the search to 10,000 revisions, and the number of questions increases to just 14.

It would be difficult to overemphasize the importance of `bisect`. Not only does it completely change the way that you find bugs, but if you routinely drive it using scripts, you'll have effectively developed regression tests on the fly, for free. Save those tests and use them!

The wily reader will observe that searching the commit history is much easier with Subversion than with the distributed tools, since its history is much more linear. The counterpoint to this is that the `bisect` command is built into the other tools, and hence more readily available and amenable to reliable automation.

DAGGY FIXES AND CHERRY-PICKING

Once you have found a bug in a piece of software, merely fixing it is rarely enough. Suppose that your bug is several years old, and there are three versions of your software in the field that need to be patched. Each version is likely to have a “sustaining” branch where bug fixes accumulate. The problem is that although the idea of copying a fix from one branch to another is simple, the practice is not so straightforward.

Mercurial, Git, and Subversion all have the ability to cherry-pick a change from one branch and apply it to another branch. The trouble with cherry-picking is that it is very brittle. A change doesn't just float freely in space: it has a context—dependencies on the code that surrounds it. Some of these dependencies are semantic and will cause a change to be cherry-picked cleanly but to fail later. Many dependencies are simply textual: someone went through and changed every instance of the word *banana* to *orange* in the destination branch, and a cherry-picked change that refers to bananas can no longer be applied cleanly.

The usual approach when cherry-picking fails because of a textual problem (sadly, a common occurrence) is to inspect the change by eye and reenter it by hand in a text editor. Distributed revision-control systems have come up with some powerful techniques to handle this type of problem.

Perhaps the most powerful approach is that taken by Darcs, a distributed revision-control system that is truly revolutionary in how it looks at changes. Instead of a simple chain or graph of changes, Darcs has a much more powerful theory of how changes depend on each other. This allows it to be enormously more successful at cherry-picking changes than any other distributed revision-control system. Why isn't everyone using Darcs, then? For years, it had severe performance problems that made it completely impractical. These have been addressed, to the point where it is now merely quite slow. Its more fundamental problem is that its theory is tricky to grasp, so two developers who are not immersed in Darcs lore can have trouble telling whether they have the same changes or not.

Let us return to the fold of Mercurial and Git. Since these tools offer the ability to make a commit on top of any revision, thereby spawning a tiny anonymous branch, a viable alternative to cherry-picking is as follows: use `bisect` to identify the revision where a bug arose; check out that revision; fix the bug; and commit the fix as a child of the revision that introduced the bug. This new change can easily be merged into any branch that had the original bug, without any sketchy cherry-picking antics required. It uses a revision-control tool's normal merge and conflict-resolution machinery, so it is far more reliable than cherry-picking (the implementation of which is almost always a series of grotesque hacks).

This technique of going back in history to fix a bug, then merging the fix into modern branches, was given the name “daggy fixes” by the authors of Monotone, an influential distributed revision-

control system. The fixes are called *daggy* because they take advantage of a project's history being structured as a directed acyclic graph, or dag. While this approach could be used with Subversion, its branches are heavyweight compared with the distributed tools, making the daggy-fix method less practical. This underlines the idea that a tool's strengths will inform the techniques that its users bring to bear.

STRENGTHS OF CENTRALIZED TOOLS

One area where the distributed tools have trouble matching their centralized competitors is with the management of binary files, large ones in particular. Although many software disciplines have a policy of never putting binary files under the management of a revision-control system, doing so is important in some fields, such as game development and EDA (electronic design automation). For example, it is common for a single game project to version tens of gigabytes of textures, skeletons, animations, and sounds. Binary files differ from text files in usually being difficult to compress and impossible to merge. Each of these brings its own challenges.

If a moderately large binary file is stored under revision control and modified many times, the space needed to store each revision can quickly become greater than the space required for all text files combined. In a centralized system, this overhead is paid only once, on the central server. With a distributed system, each repository on every laptop will have a complete copy of that file's history. This can both ruin performance and impose an unacceptable storage cost.

When two people modify a binary file, for most file formats there is no way to tell what the differences are between their versions of the file, and it is even rarer for software to help with resolving conflicts between their respective modifications. As a way of avoiding merging binary files, centralized systems offer the ability to lock files, so that only one person can edit a file in a given branch at any time. Distributed systems cannot by their nature offer locking, so they must rely on social norms (e.g., a team policy of only one person ever modifying certain kinds of files).

Relative to its distributed counterparts, a centralized tool will make the history of a branch appear more linear. Whether this is a strength or a weakness seems to be a matter of perspective. A more linear history is easier to understand, and so requires less revision-control sophistication from developers. On the other hand, a history containing numerous small branches and merges more accurately reflects the true history of a project and makes it clearer which project state a team member's code was based on when working. For teams that prefer to keep project history tidy, both Git and Mercurial offer **rebase** commands that can turn the chaotic history of a feature into a neater collection of logical changes, more suited to an eventual merger into a project's main branch.

Centralized tools can offer policy advantages that are more difficult to achieve with distributed tools. For example, it is possible to configure a pre-commit script that will reject an attempted commit if it introduces an automated test-suite failure. With a distributed tool, this kind of check can be put in place on a shared central server, but that cannot protect developers from sharing inadvertently broken changes with each other horizontally, from one laptop to another.

WHAT BEHAVIORS DOES A DISTRIBUTED TOOL CHANGE?

The availability of cheap local commits makes the use of a rapid-fire style of development attractive with distributed tools. Suppose Alice is partway through a complicated change and decides that she wants to speculatively refactor a piece of code. With a distributed tool, she can commit her change

as is, without worrying too much about whether the project is in a sane state, and try her speculative change. If that experiment fails, she can revert it and continue on her way, eventually using the `rebase` command to eliminate some of the in-progress commits she made while she figured out what she was doing.

While this style of development is certainly possible with Subversion, experience suggests that it is far more common with the distributed tools. My conjecture is that the privacy of a branch on a developer's laptop, coupled with the instantaneous responsiveness of the distributed tools, somehow combine to encourage more aggressive and pervasive use of revision control.

I have observed a similar effect with merges. Because they are such bread-and-butter activities with distributed tools, in many projects they occur far more frequently than with their centralized counterparts. Although all merges require effort and incur risk, when branches merge more frequently, the merges are smaller and less perilous. Ask any seasoned developer about a long-delayed merge following a few months of isolated work, and watch the blood drain out of his or her face.

WHAT THE FUTURE OFFERS

We are not by any means near the end of the road in the evolution of revision-control systems. The field has received only fitful attention from academia. Much work could be done on its formal foundations, which could lead to more powerful and safer ways for developers to work together. Alas, I know of only one notable publication on the topic in the past decade.¹ As a simple example, when merging potentially conflicting changes, almost everybody uses either three-way merging, which is decades old, or unpublished ad hoc approaches in which there is little reason to be confident.

More practically, there are plenty of advances to be made in the way that distributed tools handle large projects with deep histories, for which they are a poor fit because of the volume of data involved. For organizations that have sensitive needs around assurance and security, the centralized tools do somewhat better than the distributed ones, but both could improve substantially.

CONCLUSIONS

Choosing a revision-control system is a question with a surprisingly small number of absolute answers. The fundamental issues to consider are what kind of data your team works with, and how you want your team members to interact. If you have masses of frequently edited binary data, a distributed revision-control system may simply not suit your needs. If agility, innovation, and remote work are important to you, the distributed systems are far more likely to suit your needs; a centralized system may slow your team down in comparison.

There are also many second-order considerations. For example, firewall management may be an issue: Mercurial and Subversion work well over HTTP and with SSL (Secure Sockets Layer), but Git is unusably slow over HTTP. For security, Subversion offers access controls down to the level of individual files, but Mercurial and Git do not. For ease of learning and use, Mercurial and Subversion have simple command sets that resemble each other (easing the transition from one to the other), whereas Git exposes a potentially overwhelming amount of complexity. When it comes to integration with build tools, bug databases, and the like, all three are easily scriptable. Many software development tools already support or have plug-ins for one or more of these tools.

Given the demands of portability, simplicity, and performance, I usually choose Mercurial for new projects, but a developer or team with different needs or preferences could legitimately choose any

of them and be happy in the long term. We are fortunate that it is easy to interoperate among these three systems, so experimentation with the unknown is simple and risk-free.

ACKNOWLEDGMENTS

I would like to thank Bryan Cantrill, Eric Kow, Ben Collins-Sussman, and Brendan Cully for their feedback on drafts of this article.

REFERENCES

1. Löh, A., Swierstra, W., Leijen, D. 2007. A principled approach to version control; <http://people.cs.uu.nl/andres/VersionControl.html>.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BRYAN O’SULLIVAN is an Irish hacker and writer based in San Francisco. His interests include functional programming, HPC, and building large distributed systems. He is the author of the Jolt Award-winning *Real World Haskell* (2008) and *Mercurial: The Definitive Guide* (2009), both published by O’Reilly.

© 2009 ACM 1542-7730/09/0800 \$10.00