

GIT vs. Perforce- Two VCS will enter... one will leave [closed]

Asked 16 years, 2 months ago Modified 10 years, 8 months ago

Viewed 112k times

85
votes



Closed. This question needs to be more [focused](#). It is not currently accepting answers.

Closed 10 years ago.



Locked. This question and its answers are [locked](#) because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

So I'm in the process of getting GIT sold at work. First thing I need is to convince everyone that GIT is better at what they're already used to doing. We currently use Perforce. Anybody else go through a similar sale? Any good links/advice?

One of the big wins is that we can work with it disconnected from the network. Another win IMO is the way adds/checkouts are handled. More points are welcome! Also we have about 10-20 devs total.

[git](#)[perforce](#)[process-management](#)[Share](#)

edited Jul 26, 2010 at 3:54

[Phil Miller](#)

38k ● 13 ● 70 ● 91

asked Oct 21, 2008 at 17:49

[Justin Bozonier](#)

7,702 ● 9 ● 46 ● 46

Comments disabled on deleted / locked posts / reviews

14 Answers

Sorted by:

Highest score (default)

84
votes

I use Perforce at work. I also use Git because I would still like some form of version control when I'm working on code and can't connect to the server. No, reconcile offline work just is not the same. Here is where I've found Git to be a great benefit:

1. Branching speed - Git takes a few seconds at most.
2. Conflicts - P4Merge's auto resolve destroyed a week's worth of work once. Ever since then I would rather resolve by hand when merging. When Git prompts me about a conflict, it is actually a conflict. The rest of the time, Git resolves stuff correctly and I save heaps of time.
3. Keeping track of merges - If you have one branch that is continuously receiving merges from two other branches, you know what a headache this can be with

performer. With git, the headache is minimized because the result of a merge in git is actually a new commit which knows who its ancestors are.

4. Permissions - I have lost track of the number of times I've tried to work on a file but couldn't because it was not checked out in Performer. If you worked with XCode (or any editor that does not have a solid Performer SCM plugin) offline, you know how irritating this can get. I don't have to worry about that with Git. I make my changes. Git does not stop me and tracks them in the background.
5. Keeping the main tree tidy - With git, I can sort my commits and tidy the code up so that the history looks nice and tidy. None of that "checking in this file because it was supposed to be part of the previous checkin" rubbish. I squash commits like that, because they help nobody.
6. Stashing - Your performer server needs to be version 2010.1 or newer to use the p4 shelve command.
7. Creating patches - Easy to do in git. Don't know if it is possible in Performer without using the command line.
8. Mailing patches from the GUI - again, git wins here.
9. Disk space - With performer, every branch is a copy. That means if your source tree is huge, your disk space gets eaten up fast. This is not even counting the additional space once you start building. Why even have a link between branches and disk space? With git, you can have 100 branches and only one branch at

a time ever exists. If you specifically want to work on two versions simultaneously you can clone, do your work and then get rid of one clone if you want, without losing anything.

10. If you're on XCode4, perforce support has been dropped and git support is now built in. If you do cross platform work like I do, this matters a lot. With Visual Studio, you can use git extensions. With perforce, it's equally yuck on both OSes. Well, maybe a little more on mac now with XCode4 on the scene.
11. Finding the faulty checkin (or, git bisect rules) - Ever tried to do a binary search with perforce to figure out where a bug was introduced? Quite a hassle, yes? Even more of a hassle when there have been integrates from other branches in the middle. Why? Because there is no automation for such tasks. You need to write your own tool to talk to perforce and you usually don't have the time. With git, you give it the starting points (the "good" point and the "bad" point) and it automates the search for you. Even better, if you have a script that can automate the build and test process, you can hook git up to the script and the whole process of finding the checkin is automated. That is how it should be.
12. Tracking changes across refactors - Try splitting BigClass into SmallClass1 and SmallClass2. To Perforce, BigClass has now ceased to exist and two new classes (SmallClass1 and SmallClass2 have joined the source tree). To Perforce, there is no relation between BigClass and SmallClass1 and SmallClass2.

Git, on the other hand, is smart enough to know that x% of BigClass is now in SmallClass1 and y% of BigClass is in SmallClass2 and that BigClass has ceased to exist. Now, from the point of view of someone who is reviewing changes across multiple branches, you tell me which approach you'd find more useful - Git's or Perforce's. Personally, I prefer Git's approach because it more accurately reflects the actual change in the code. Git is able to do this because it tracks content within the file and not the file itself.

13. Centralized or decentralized: Git is a [DVCS](#) system while perforce is centralized. A centralized VCS cannot be decentralized later, but a DVCS (especially git) can be centralized. There are several products that add very fine grained access control to git, if that is something the business needs. Personally, I would go with a system that gives me greater flexibility in the long term.
14. Branch mappings: If you want to do branching right in Perforce, you need to create a branch mapping. There are reasons for this, but they are tied to how Perforce conceptualizes a branch. As a developer, or a team, this simply means one more step in the work flow, which I do not consider efficient at all.
15. Sharing work between teams: With Perforce, you cannot break up a submission. Team A is working on feature A. Team B on feature B. Team C works on bug fixes. Now, Teams A & B have to fix a bunch of bugs in order to implement their features. The only thing is,

they weren't so disciplined when committing their changes (probably because they're rushing to a deadline) and so their "bug fixes" are parts of larger submissions that also contain new stuff as far as version control on their branches are concerned. However, Team C is now doing a point release and would like to get the bug fixes from the other teams. If they were using Git, Team C could cherry pick the other teams' relevant changes, split them up and only take what they needed without worrying about introducing any partially implemented features. With Perforce, Team C can get the affected files, but would have to separate the relevant changes using a much more manual process.

16. Changing platform - If, for whatever reason in the future, you decide to change your platform of choice, with Perforce, you're at the mercy of Perforce.com and the availability of the tools for the platform of your choosing.
17. Changing to future amazing source control engine X - If you decide to change what you use for source control, extracting your source control history from Perforce and moving it to new system X is going to be a nightmare, because it is closed source and the best you can do is guess - just Google for Perforce to Git migration to get a sense of what I'm talking about. At least with Git, its open source, so it eliminates a lot of the guesswork involved.

Well, that's my 2 cents. In Perforce's defense, I gotta say their customer support rules and so does their Time Lapse View tool. I do not know how to get a time lapse view with git. But for the convenience and time saved, I'd go with git any day.

Share

edited Mar 12, 2013 at 20:43

answered May 6, 2010 at 3:54



Carl

44.3k ● 10 ● 83 ● 109

2 re #6 (stash): p4 shelve, it's new. – [Trey](#) Jun 11, 2010 at 0:10

8 The biggest difference between Perforce and Git is the mind set required. If you're running a centralized VCS shop, git is going to be a really hard sell, because it requires a change in the way you, your team and the business think about version control. In other words, git is an excellent and efficient tool which is technically more capable than Perforce. The hard part is convincing the humans :) – [Carl](#) Feb 6, 2012 at 21:05

1 I'm in love with Perforce. Reading this post feels like cheating ... – [KlausCPH](#) Jan 7, 2013 at 22:55

2 @KlausCPH at least you won't have to prepare a breakup speech when you leave Perforce :) – [Carl](#) Jan 7, 2013 at 22:57

@carleeto Can you elaborate #13? What product helps access control? – [chen](#) Apr 7, 2013 at 17:18 ✎

75 The Perl 5 interpreter source code is currently going through the throes of converting from Perforce to git.

votes



Maybe Sam Vilain's `git-p4raw` importer is of interest.

In any case, one of the major wins you're going to have over every centralised VCS and most distributed ones also is raw, blistering **speed**. You can't imagine how liberating it is to have the entire project history at hand, mere fractions of fractions of a second away, until you have experienced it. Even generating a commit log of the whole project history that includes a full diff for each commit can be measured in fractions of a second. Git is so fast your hat will fly off. VCSs that have to roundtrip over the network simply have no chance of competing, not even over a Gigabit Ethernet link.

Also, git makes it very easy to be carefully selective when making commits, thereby allowing changes in your working copy (or even within a single file) to be spread out over multiple commits – and across different branches if you need that. This allows you to make fewer mental notes while working – you don't need to plan out your work so carefully, deciding up front what set of changes you'll commit and making sure to postpone anything else. You can just make any changes you want as they occur to you, and still untangle them – nearly always quite easily – when it's time to commit. [The stash](#) can be a very big help here.

I have found that together, these facts cause me to naturally make many more and much more focused commits than before I used git. This in turn not only makes

your history generally more useful, but is particularly beneficial for value-add tools such as [git bisect](#).

I'm sure there are more things I can't think of right now. One problem with the proposition of selling your team on git is that many benefits are interrelated and play off each other, as I hinted at above, such that it is hard to simply look at a list of features and benefits of git and infer how they are going to change your workflow, and which changes are going to be bonafide improvements. You need to take this into account, and you also need to explicitly point it out.

Share

edited Oct 21, 2008 at 23:46

answered Oct 21, 2008 at 23:41



Aristotle Pagaltzis

118k ● 23 ● 101 ● 100

Allegedly, [p4sandbox](#) provides some offline abilities to p4. Nonetheless I still like git. :) – [Dominic Mitchell](#) Nov 8, 2011 at 11:55

13 Git doesn't provide 'offline abilities', it **is** offline. The only you send data over the wire is when you push commits to the origin or pull changes from other sub-repositories.

– [Evan Plaice](#) Apr 23, 2012 at 5:45

2 "Git is so fast your hat will fly off" love that :) Only thing its not very true when you start to check-in binary files. large repos are troublesome in git. – [v.oddou](#) Aug 27, 2013 at 8:06

- 1 Unfortunately true. The way Git works depends on reading files and somewhat on diffing them, so they need to be modestly sized and well-diffable. Then it'll be *extremely* fast (as in the example of instant full-diff history for hundreds of commits). With huge blobs? Not so much...
– [Aristotle Pagaltzis](#) Nov 10, 2013 at 18:08
-

46
votes



It would take me a lot of convincing to switch from perforce. In the two companies I used it it was more than adequate. Those were both companies with disparate offices, but the offices were set up with plenty of infrastructure so there was no need to have the disjoint/disconnected features.

How many developers are you talking about changing over?

The real question is - what is it about perforce that is not meeting your organization's needs that git can provide? And similarly, what weaknesses does git have compared to perforce? If you can't answer that yourself then asking here won't help. You need to find a business case for your company. (e.g. Perhaps it is with lower overall cost of ownership (that includes loss of productivity for the interim learning stage, higher admin costs (at least initially), etc.)

I think you are in for a tough sell - perforce is a pretty good one to try to replace. It is a no brainer if you are trying to boot out pvcs or ssafe.

Share

[edited Oct 4, 2010 at 1:56](#)

answered Oct 21, 2008 at 18:19



Tim

20.4k ● 24 ● 122 ● 219

18 I'll add that Git's remarkable feature set comes at the cost of a steep learning curve. (Though I never found Perforce all that intuitive either.) – [savetheclocktower](#) Oct 21, 2008 at 18:26

1 Great answer Tim. Justin - why is your boss sold? Surely you must have addressed Tim's question to do that? I would be interested in the justification too. – [Greg Whitfield](#) Oct 22, 2008 at 9:18

4 You do have to pay for Perforce in a commercial environemnt, and really I have always find that Perforce jarring to use. And the only strength I really see is that it's good at handling huge binary blobs. – [Calyth](#) Jan 28, 2009 at 16:27

2 @Justin: Beware of the "we will only use the basic features" reason. With git, you will end up using the more advanced stuff. Why wouldn't you? Bisect comes to mind immediately. So do rebase and cherry-pick. – [Carl](#) Feb 16, 2011 at 1:14 ✎

3 Actually, us having a *relevant* conversation in comments which ended as soon as the "are you sure you wouldn't like to move this to chat" prompt came up caused someone to finally notice that this question is in fact, not suitable for SO. It's essentially asking for why one system is "better" than another, and inviting a whole lot of spirited debate along with it. – [Adam Parkin](#) Aug 6, 2012 at 4:40

15

votes



I think in terms of keeping people happy during/ post switch over, one of things to get across early is just how private a local branch can be in Git, and how much freedom that gives them to make mistakes. Get them all to clone



themselves a few private branches from the current code and then go wild in there, experimenting. Rename some files, check stuff in, merge things from another branch, rewind history, rebase one set of changes on top of another, and so on. Show how even their worst accidents locally have no consequences for their colleagues. What you want is a situation where developers feel safe, so they can learn faster (since Git has a steep learning curve that's important) and then eventually so that they're more effective as developers.

When you're trying to learn a centralised tool, obviously you will be worried about making some goof that causes problems for other users of the repository. The fear of embarrassment alone is enough to discourage people from experimenting. Even having a special "training" repository doesn't help, because inevitably developers will encounter a situation in the production system that they never saw during training, and so they're back to worrying.

But Git's distributed nature does away with this. You can try any experiment in a local branch, and if it goes horribly wrong, just throw the branch away and nobody needs to know. Since you can create a local branch of anything, you can replicate a problem you're seeing with the real live repository, yet have no danger of "breaking the build" or otherwise making a fool of yourself. You can check absolutely everything in, as soon as you've done it, no trying to batch work up into neat little packages. So not just the two major code changes you spent four hours on today, but also that build fix that you remembered half way

through, and the spelling mistake in the documentation you spotted while explaining something to a colleague, and so on. And if the major changes are abandoned because the project is changing direction, you can cherry pick the build fix and the spelling mistake out of your branch and keep those with no hassle.

Share


answered Oct 22, 2008 at 13:28



tialaramex

3,801 ● 1 ● 23 ● 23

There's no reason you cannot have a private branch of a centralized system either. DVCSs sometimes do better at merging, but just because the private branch doesn't exist on the remote repo doesn't mean that you can't create a branch only for you which exists on the remote repo. – [Billy O'Neal](#) Oct 4, 2010 at 2:36

2 This comment is technically correct, but that's sort of missing the point. Revision control systems are a social tool. Socially the branch with your name on it in the shared server isn't "private" it's shared. Yes, even with ACLs and whatever in place. There are technical differences too (obviously my git private branch is used while I'm commuting home, with no/unreliable Internet) but they're subsidiary to the social difference. – [tialaramex](#) Oct 11, 2010 at 11:34 

2 A private branch with perforce, just sucks. The ease at which you create and switch between branches in git, can not be compared with perforce. How private is really a perforce "private" branch anyway. You don't keep it local. You are completely dependent on network access. – [Erik Engheim](#) Jan 21, 2011 at 19:00

9

votes



The command that sold me on git personally was [bisect](#). I don't think that this feature is available in any other version control system as of now.

That being said, if people are used to a GUI client for source control they are not going to be impressed with git. Right now the only full-featured client is command-line.

Share

answered Oct 21, 2008 at 17:52



Ryan

9,928 ● 7 ● 44 ● 57

-
- 1 For fairness (I picked git over Hg) it has to be noted that Mercurial has bisecting capability too – although it ships as a plugin that you have to load explicitly. – [Aristotle Pagaltzis](#) Oct 21, 2008 at 23:19
 - 2 darcs has had "trackdown" since before git existed. The early versions were pretty rough, admittedly. – [wnoise](#) Oct 23, 2008 at 22:01
 - 2 regarding UI - GitX on OSX is excellent. – [Antony Stubbs](#) Mar 1, 2010 at 10:06
 - 4 SourceTree is also another nice native osx client. It became free after it was acquired. I have been using it for sometime now and I like it. I was mostly commandliner before using SourceTree. – [Prakash Nadar](#) Dec 30, 2011 at 4:18
 - 1 In my experience with Git, you really need both the command line and a graphical client to use it effectively. You need the command line because there's a lot of power that's not easy to put into a GUI, and you need the GUI (or at least `git log --graph`) because Git revision histories have a tendency to be nonlinear and hard to visualize without a picture. I use GitX and SourceTree as GUIs, though gitk (which comes with Git now) is

9

votes



What Perforce features are people using?

- Multiple workspaces on a single machine
- Numbered changelists
- Developer branches
- Integration with IDE (Visual Studio, Eclipse, SlickEdit, ...)
- Many build variants
- Composite workspaces
- Integrating some fixes but not others
- etc

I ask because if all folks are doing is get and put from the command line, git has that covered, and so do all the other RTS.

Share

answered Oct 21, 2008 at 18:56



[Thomas L Holaday](#)

13.8k ● 6 ● 43 ● 52

-
- 2 Not sure about what "multiple workspaces on single machine" means -- other VCSs simply don't have the concept of a workspace so it really cannot be touted as a feature for Perforce. The others make sense though. – [Billy ONeal](#) Oct 4, 2010 at 2:34
-

Multiple workspace example: client A has dev and release versions of A-only files plus a subset of internal tools at version 1.52; client B has dev & release B-only files plus a different but overlapping subset of internal tools, both dev and version 1.52. Developer is working on both simultaneously, and may choose to make the changed internal tools visible to A to see what breaks. – [Thomas L Holaday](#) Oct 7, 2010 at 22:20

- 6 @Tomas: Why don't you... Simply have it checked out twice? Perforce needs to have it as a "feature" because otherwise it gets strange due to having to ensure that environment variables are set properly, registry entries correct, etcetera. – [Arafangion](#) Jun 27, 2011 at 4:23 ✎
-

@Arafangion, it is not obvious how checking out twice facilitates the selective exposure of files to build sets. – [Thomas L Holaday](#) Jun 29, 2011 at 22:53

6 votes Apparently [GitHub now offer git training courses to companies](#). Quoth [their blog post about it](#):



I've been down to the Google campus a number of times in the last few weeks helping to train the Androids there in Git. I was asked by Shawn Pearce (you may know him from his Git and EGit/JGit glory – he is the hero that takes over maintenance when Junio is out of town) to come in to help him train the Google engineers working on Android in **transitioning from Perforce to Git**, so Android could be shared with the masses. I can tell you I was more than happy to do it.

[...]

Logical Awesome is now [officially offering](#) this type of custom training service to all companies, where we can help your organization with training and planning if you are thinking about switching to Git as well.

Emphasis mine.

Share

edited Oct 23, 2008 at 22:59

answered Oct 23, 2008 at 21:49



[Aristotle Pagaltzis](#)

118k ● 23 ● 101 ● 100

4

votes



Perforce features:



1. GUI tools seem to be more feature rich (e.g. Time lapse view, Revision graph)
2. Speed when syncing to head revision (no overhead of transferring whole history)
3. Eclipse/Visual Studio Integration is really nice
4. You can develop multiple features in one branch per Changelist (I am still not 100% sure if this is an advantage over GIT)

5. You can "spy" what other developers are doing - what kind of files they have checked out.

GIT features:

1. I got impressions that GIT command line is much simpler than Perforce (init/clone, add, commit. No configuration of complex Workspaces)
2. Speed when accessing project history after a checkout (comes at a cost of copying whole history when syncing)
3. Offline mode (developers will not complain that unreachable P4 server will prohibit them from coding)
4. Creating a new branches is much faster
5. The "main" GIT server does not need plenty of TBytes of storage, because each developer can have it's own local sandbox
6. GIT is OpenSource - no Licensing fees
7. If your Company is contributing also to OpenSource projects then sharing patches is way much easier with GIT

Overall for OpenSource/Distributed projects I would always recommend GIT, because it is more like a P2P application and everyone can participate in development. For example, I remember that when I was doing remote development with Perforce I was syncing 4GB Projects over 1Mbps link once in a week. A lot of time was simply wasted because of that. Also we needed set up VPN to do that.

If you have a small company and P4 server will be always up then I would say that Perforce is also a very good option.

Share

answered Feb 15, 2011 at 23:37



[user2138912](#)

1,686 ● 3 ● 20 ● 41

Git feature #1 is a dubious claim at best. Perhaps Git wins on setup, but day to day usage is pretty clumsy (often requiring multiple commands to accomplish a simple task)

– [Adam Parkin](#) Aug 2, 2012 at 20:26

1 @AdamParkin What tasks do you find clumsy from the command line? (I use GUIs where possible, but I think Git's command structure is decent.) – [Marnen Laibow-Koser](#) Aug 5, 2012 at 18:40

1 @AdamParkin Well, the ease of using command line is aesthetic aspect, hence it is subjective at least. The reason why I personally consider Git's Command line simpler than Perforce's, is that in Perforce you have to set up workspaces and environment variables (P4USER e.t.c) before you can even start to work with the repository files, compared to a single "git clone" command. Of course there are some advanced git Commands that are absent in Perforce (e.g. rewriting local history) and they might seem like "rocket science" for regular Perforce users. – [user2138912](#) Aug 6, 2012 at 17:58

I havn't used it but it seems to me that managing change sets is just a poor man's branch considering in git you can squash your feature branches or rebase them onto master.

– [Ryan Leach](#) May 23, 2014 at 16:19 ✎

4
votes



We have been using Git for sometime, recently our Git server's harddrive crashed and we could not revert back to the latest state. We managed to get back to few days old state. When the server was back up. Everyone in the team pulled/pushed their changes and voila, the server is back to current state.

Share

answered Dec 30, 2011 at 4:22



Prakash Nadar

2,711 ● 1 ● 21 ● 22

3 In a talk Linus gave @ Google about Git he talks about how he doesn't do backups as everyone's clone of the Linux kernel is a full backup of it. Really good point actually. – [Adam Parkin](#) Aug 2, 2012 at 20:27

That is true in all sense, each close is a "backup" but the git in many cases is still used as a "centralised-distributed" tool. i.e. just like SVN with added branching benefits. The organisation always wants a back-up of everything they have.

– [Prakash Nadar](#) Aug 2, 2012 at 21:36

3
votes



The one important difference between Perforce and git (and the one most commonly mentioned) is their respective handling of huge binary files.

Like, for example, in this blog of an employee at a video game development company:

<http://corearchitecture.blogspot.com/2011/09/git-vs-perforce-from-game-development.html>

However, the important thing is that, the speed difference between git and perforce, when you have a huge 6gb repository, containing everything from documentation to every binary ever built (and finally, oh yes! the actual source history), usually comes from the fact that huge companies tend to run Perforce, and so they set it up to offload all significant operations to the huge server bank in the basement.

This important advantage on Perforce's part comes only from a factor that has nothing whatsoever to do with Perforce, the fact that the company running it can afford said server bank.

And, anyway, in the end, Perforce and git are different products. Git was designed to be solely a VCS, and it does this far better than Perforce (in that it has more features, which are generally easier to use, in particular, in the words of another, branching in Perforce is like performing open-heart surgery, it should only be done by experts :P) (<http://stevehanov.ca/blog/index.php?id=50>)

Any other benefits which companies that use Perforce gain have come merely because Perforce is not solely a VCS, it's also a fileserver, as well as having a host of other features for testing the performance of builds, etc.

Finally: Git being open-source and far more flexible to boot, it would not be so hard to patch git to offload important operations to a central server, running mounds of expensive hardware.

answered Apr 3, 2012 at 3:39



3

votes



I think the one thing that I know GIT wins on is it's ability to "preserve line endings" on all files, whereas perforce seems to insist on translating them into either Unix, Dos/Windows or MacOS9 format ("`\n`", "`\r\n`" or "`\r`").

This is a real pain if you're writing Unix scripts in a Windows environment, or a mixed OS environment. It's not even possible to set the rule on a per-file-extension basis. For instance, it would convert `.sh`, `.bash`, `.unix` files to Unix format, and convert `.ccp`, `.bat` or `.com` files to Dos/Windows format.

In GIT (I'm not sure if that's default, an option or the only option) you can set it up to "preserve line endings". That means, you can manually change the line endings of a file, and then GIT will leave that format the way it is. This seems to me like the ideal way to do things, and I don't understand why this isn't an option with Perforce.

The only way you can achieve this behavior, is to mark the files as binary. As I see that, that would be a nasty hack to workaround a missing feature. Apart from being tedious to have to do on all scripts, etc, it would also probably break most diffs, etc.

The "solution" that we've settled for at the moment, is to run a sed command to remove all carriage returns from the scripts every time they're deployed to their Unix environment. This isn't ideal either, especially since some of them are deployed inside WAR files, and the sed line has to be run again when they're unpacked.

This is just something I think gives GIT a great advantage, and which I don't think has been mentioned above.

EDIT: After having been using Perforce for a bit longer, I'd like to add another couple of comments:

A) Something I really miss in Perforce is a clear and instance diff, including changed, removed and added files. This is available in GIT with the `git diff` command, but in Perforce, files have to be checked out before their changes are recorded, and while you might have your main editors (like Eclipse) set up to automatically check files out when you edit them, you might sometimes edit files in other ways (notepad, unix commands, etc). And new files don't seem to be added automatically at all, even using Eclipse and p4eclipse, which can be rather annoying. So to find all changes, you have to run a "Diff against..." on the entire workspace, which first of all takes a while to run, and secondly includes all kind of irrelevant things unless you set up very complicated exclusion lists, which leads me to the next point.

B) In GIT I find the .gitignore very simple and easy to manage, read and understand. However, the workspace ignore/exclude lists configurable in Perforce seem unwieldy

and unnecessarily complex. I haven't been able to get any exclusions with wildcards working. I would like to do something like

```
-//Server/mainline/.../target/...  
//Svend_Hansen_Server/.../target/...
```

To exclude all target folders within all projects inside Server/mainline. However, this doesn't seem to work like I would have expected, and I've ended up adding a line for every project like:

```
-//Server/mainline/projectA/target/...  
//Svend_Hansen_Server/projectA/target/...  
-//Server/mainline/projectB/target/...  
//Svend_Hansen_Server/projectB/target/...  
...
```

And similar lines for bin folders, .classpath and .projet files and more.

C) In Perforce there are the rather useful changelists. However, assume I make a group of changes, check them all and put them in a changelist, to then work on something else before submitting that changelist. If I later make a change to one of the files included in the first changelist, that file will still be in that changelist, and I can't just later submit the changelist assuming that it only contains the changes that I originally added (though it will be the same files). In GIT, if you add a file and then make further changes to it, those changes will not have been added (and would still show in a `git diff` and you wouldn't be able to

commit the file without first adding the new changes as well. Of course, this isn't usefull in the same way the changelist can be as you only have one set of added files, but in GIT you can just commit the changes, as that doesn't actually push them. You could then work on other changes before pushing them, but you wouldn't be able to push anything else that you add later, without pushing the former changes as well.

Share

edited Aug 20, 2012 at 13:35

answered Apr 13, 2012 at 9:24



Svend Hansen

3,333 ● 3 ● 34 ● 53

2

votes



I have no experience with Git, but I have with Mercurial which is also a distributed VCS. It depends on the project really, but in our case a distributed VCS suited the project as basically eliminated frequent broken builds.

I think it depends on the project really, as some are better suited towards a client-server VCS, and others towards a distributed one.

Share

answered Oct 21, 2008 at 17:57



Terminus

902 ● 2 ● 10 ● 21

(Granted, this is an old answer, but...) You can run Git (and I assume also Mercurial) as if it were a client-server VCS. It *still* works better than client-server VCSs, due to ease of branching and merging and the possibility of private commits. I don't see much use for client-server VCSs anymore, at least until they get their merge skills up to par. – [Marnen Laibow-Koser](#) May 21, 2012 at 14:35 ✎

-5 Here's what I don't like about git:

votes



First of all, I think the distributed idea flies in the face of reality. Everybody who's actually using git is doing so in a centralised way, even Linus Torvalds. If the kernel was managed in a distributed way, that would mean I couldn't actually download the "official" kernel sources - there wouldn't be one - I'd have to decide whether I want Linus' version, or Joe's version, or Bill's version. That would obviously be ridiculous, and that's why there is an official definition which Linus controls using a centralised workflow.

If you accept that you want a centralised definition of your stuff, then it becomes clear that the server and client roles are completely different, so the dogma that the client and server softwares should be the same becomes purely limiting. The dogma that the client and server *data* should be the same becomes patently ridiculous, especially in a codebase that's got fifteen years of history that nobody cares about but everybody would have to clone.

What we actually want to do with all that old stuff is bung it in a cupboard and forget that it's there, just like any normal

VCS does. The fact that git hauls it all back and forth over the network every day is very dangerous, because it nags you to prune it. That pruning involves a lot of tedious decisions and it can go wrong. So people will probably keep a whole series of snapshot repos from various points in history, but wasn't that what source control was for in the first place? This problem didn't exist until somebody invented the distributed model.

Git actively encourages people to rewrite history, and the above is probably one reason for that. Every normal VCS makes rewriting history impossible for all but the admins, and makes sure the admins have no reason to consider it. Correct me if I'm wrong, but as far as I know, git provides no way to grant normal users write access but ban them from rewriting history. That means any developer with a grudge (or who was still struggling with the learning curve) could trash the whole codebase. How do we tighten that one? Well, either you make regular backups of the entire history, i.e. you keep history squared, or you ban write access to all except some poor sod who would receive all the diffs by email and merge them by hand.

Let's take an example of a well-funded, large project and see how git is working for them: Android. I once decided to have a play with the android system itself. I found out that I was supposed to use a bunch of scripts called repo to get at their git. Some of repo runs on the client and some on the server, but both, by their very existence, are illustrating the fact that git is incomplete in either capacity. What happened is that I was unable to pull the sources for about

a week and then gave up altogether. I would have had to pull a truly vast amount of data from several different repositories, but the server was completely overloaded with people like me. Repo was timing out and was unable to resume from where it had timed out. If git is so distributable, you'd have thought that they'd have done some kind of peer-to-peer thing to relieve the load on that one server. Git is distributable, but it's not a server. Git+repo is a server, but repo is not distributable cos it's just an ad-hoc collection of hacks.

A similar illustration of git's inadequacy is gitolite (and its ancestor which apparently didn't work out so well.) Gitolite describes its job as easing the deployment of a git server. Again, the very existence of this thing proves that git is not a server, any more than it is a client. What's more, it never will be, because if it grew into either it would be betraying it's founding principles.

Even if you did believe in the distributed thing, git would still be a mess. What, for instance, is a branch? They say that you implicitly make a branch every time you clone a repository, but that can't be the same thing as a branch in a single repository. So that's at least two different things being referred to as branches. But then, you can also rewind in a repo and just start editing. Is that like the second type of branch, or something different again? Maybe it depends what type of repo you've got - oh yes - apparently the repo is not a very clear concept either. There are normal ones and bare ones. You can't push to a normal one because the bare part might get out of sync with its

source tree. But you can't cvsimport to a bare one cos they didn't think of that. So you have to cvsimport to a normal one, clone that to a bare one which developers hit, and cvsexport that to a cvs working copy which still has to be checked into cvs. Who can be bothered? Where did all these complications come from? From the distributed idea itself. I ditched gitolite in the end because it was imposing even more of these restrictions on me.

Git says that branching should be light, but many companies already have a serious rogue branch problem so I'd have thought that branching should be a momentous decision with strict policing. This is where perforce really shines...

In perforce you rarely need branches because you can juggle changesets in a very agile way. For instance, the usual workflow is that you sync to the last known good version on mainline, then write your feature. Whenever you attempt to modify a file, the diff of that file gets added to your "default changeset". When you attempt to check in the changeset, it automatically tries to merge the news from mainline into your changeset (effectively rebasing it) and then commits. This workflow is enforced without you even needing to understand it. Mainline thus collects a history of changes which you can quite easily cherry pick your way through later. For instance, suppose you want to revert an old one, say, the one before the one before last. You sync to the moment before the offending change, mark the affected files as part of the changeset, sync to the moment after and merge with "always mine". (There was something

very interesting there: syncing doesn't mean having the same thing - if a file is editable (i.e. in an active changeset) it won't be clobbered by the sync but marked as due for resolving.) Now you have a changelist that undoes the offending one. Merge in the subsequent news and you have a changelist that you can plop on top of mainline to have the desired effect. At no point did we rewrite any history.

Now, supposing half way through this process, somebody runs up to you and tells you to drop everything and fix some bug. You just give your default changelist a name (a number actually) then "suspend" it, fix the bug in the now empty default changelist, commit it, and resume the named changelist. It's typical to have several changelists suspended at a time where you try different things out. It's easy and private. You get what you really want from a branch regime without the temptation to procrastinate or chicken out of merging to mainline.

I suppose it would be theoretically possible to do something similar in git, but git makes practically anything possible rather than asserting a workflow we approve of. The centralised model is a bunch of valid simplifications relative to the distributed model which is an invalid generalisation. It's so overgeneralised that it basically expects you to implement source control on top of it, as repo does.

The other thing is replication. In git, anything is possible so you have to figure it out for yourself. In perforce, you get an effectively stateless cache. The only configuration it needs

to know is where the master is, and the clients can point at either the master or the cache at their discretion. That's a five minute job and it can't go wrong.

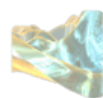
You've also got triggers and customisable forms for asserting code reviews, bugzilla references etc, and of course, you have branches for when you actually need them. It's not clearcase, but it's close, and it's dead easy to set up and maintain.

All in all, I think that if you know you're going to work in a centralised way, which everybody does, you might as well use a tool that was designed with that in mind. Git is overrated because of Linus' fearsome wit together with peoples' tendency to follow each other around like sheep, but its main *raison d'être* doesn't actually stand up to common sense, and by following it, git ties its own hands with the two huge dogmas that (a) the software and (b) the data have to be the same at both client and server, and that will always make it complicated and lame at the centralised job.

Share

edited Jun 21, 2012 at 3:39

answered Jun 21, 2012 at 3:30



Adrian May

2,184 ● 1 ● 15 ● 26

-
- 4 Oh, what the heck. I have a few minutes, so I'll try to comment on some of the larger errors. "If the kernel was managed in a distributed way, that would mean I couldn't actually download

the "official" kernel sources - there wouldn't be one - I'd have to decide whether I want Linus' version, or Joe's version, or Bill's version."—I can't speak to the Linux kernel project specifically, since I've never worked on it, but in general, this is in fact how open-source software works. You can download any fork you like. — [Marnen Laibow-Koser](#) Aug 5, 2012 at 18:52

- 2 "What we actually want to do with all that old stuff is bung it in a cupboard and forget that it's there, just like any normal VCS does. The fact that git hauls it all back and forth over the network every day is very dangerous, because it nags you to prune it." • Have you ever actually used Git? Git does not nag you to prune your repo. Besides, Git's data compression is extremely efficient; it's not uncommon for a Git repo—with complex branch history—to be significantly smaller than a SVN working copy (without history) of the same codebase.
— [Marnen Laibow-Koser](#) Aug 5, 2012 at 18:55
-

- 4 "Every normal VCS makes rewriting history impossible for all but the admins, and makes sure the admins have no reason to consider it. Correct me if I'm wrong, but as far as I know, git provides no way to grant normal users write access but ban them from rewriting history. That means any developer with a grudge (or who was still struggling with the learning curve) could trash the whole codebase." • You're 100% wrong. It's easy to disallow force-pushes to the repo while still allowing write access. Also, everyone can have a copy of as much of the repo as they like, so trashing wouldn't work.
— [Marnen Laibow-Koser](#) Aug 5, 2012 at 19:00
-

- 3 "Git says that branching should be light, but many companies already have a serious rogue branch problem so I'd have thought that branching should be a momentous decision with strict policing. This is where perforce really shines..." • What is a "rogue branch problem"? Why do you think branching should be a "momentous decision"? In practice, it's really useful to be able to create branches (private or public) for each new feature or experiment, so that each one lives in its own alternate universe. Unlike, say, SVN, Git makes merging easy so that

this works very well. – [Marnen Laibow-Koser](#) Aug 5, 2012 at 19:06 

- 3 The fact that this answer only has one downvote aside from mine (apparently @Marnen) astounds me given how objectively wrong this answer is. – [alternative](#) Aug 20, 2012 at 14:59
-

-8
votes



Using GIT as substitute for bad code line management is common. Many of the disadvantages of Perforce are a result of bad branching strategies. The same for any other centralized tool. If you have to create a ton of branches you are doing something wrong. Why do developers need to create so many branches?

Also, why is working disconnected so important anyway? Just so someone can work on a train? That's about the only place these days you can't get a wireless connection. And even most trains have decent WiFi.

Share

answered Apr 7, 2011 at 22:44



[sogboj](#)

11 ● 1

-
- 9 Some developers like to create branches so they can easily isolate and segment different developments, prototype, bug fixes etc. Often depends on the type of work. Perforce branches are quite heavyweight to manage compared to Git & Mercurial branches, and so there can be some genuine productivity improvements to be made. – [Greg Whitfield](#) Apr 9, 2011 at 22:27
-

8 The ability to work disconnected is not always related to the idea of working on a train or plane. Some companies may not have reliable networking infrastructure. Or you can get outages, server maintenance, general cockups etc. But the side effect of being able to work disconnected is that your source control operations are blindingly fast compared to systems that rely on a network round trip to do anything. – [Greg Whitfield](#) Apr 9, 2011 at 22:31

6 In my experience, using process to control people is indicative of a bad workflow design in the first place. A workflow should exist to keep people productive. If it's not being used, there is something wrong with it, because in general, people aren't idiots and will use a better tool if they come across it. – [Carl](#) Jun 20, 2011 at 1:32

6 Downvoting because of "If you have to create a ton of branches you are doing something wrong." That may be true in a centralized VCS with inadequate merging tools (like SVN or CVS -- never used Perforce). It's not true in Git, where branching and merging are easy and common. This gives the freedom for each feature under development to be in its own alternate universe until integration. Personally, I'd never go back to an environment where I *couldn't* branch on a whim. – [Marnen Laibow-Koser](#) May 18, 2012 at 20:08
