Peer-to-Peer Collaborative Development Using GIT

May 14th, 2012

Wow how time flies when you're having fun! I first thought of writing this post back in January, when I was on a roll with writing blog posts. But it never materialized beyond notes I collected in preparation. Come several months later with a lot more experience on what I am about to tell you, and you have this post. The notes I'm reffering to is about a development style that came about as a result of optimizing, how we at thinkCube organize and work with source code in a revision control system. Considering the wide use of such systems such as SVN and GIT, I thought I'd share our development experience in the hope it will help you to take another look at your own development style. But before I get into it, I'd like very briefly touch on some background on the evolution of development styles around version control systems.

Thou shalt not commit, Yet!

Back in the days of CVS, source code lived centrally on a server called a repository. One had to earn the right to read/write to this repository in order to ensure "world order" (others had to submit a patch via Bugzilla). While this led to a centrally managed system of Collaborative software development, it also created a new software development style of "Earn your commitership", "Commit often" and "Communicate often". Nothing much changed when everyone moved over to SVN, which started out as an improved CVS. This development style didn't go down well with Linus Torvalds for his Linux work and so he created GIT instead and hates CVS/SVN.

Pray-Pull-Push style of development

We started with CVS back in 2005 and then SVN for managing our development at thinkCube. A little while after git came along when it was stable enough to use and also was usuable by mere mortals, we made the switch. At first we had some trouble wrapping our heads around git and so just used it like SVN where we used to always do `commit/pull/push` operations as if they were atomic. But after about a week or two, we realized the power of git was in its ability to let you commit locally and push when you were ready to share. And so, yet another development style arose where git acted as a Collaborative Whiteboard for sharing code changes. This also meant developers needed push access to the central repository in the same manner commit access was required with SVN/CVS.

The unfortunate development style this results in is that developers may occasionally push unfinished worked upstream just to share it with the "Developer next door". The consequence of that maybe felt by the poor developer that spent all night working on a feature, pulls in oder to push to find that his code now conflicts for no apparent reason!

Now I know a lot of you may be on this style of development and thinking, "If our developers do that, we punish them!" and so they don't! Good for you! But my take on this is, if the system is fundamentally broken then it's better to fix it than to enforce tough rules. For example, if I may digress for a bit, "Why are TukTuk drivers and Motor Cyclist so careless?". Is it likely that only careless drivers pick up these vehicles or is it more likely the vehicle made them reckless? My advice is if the tool is broken, then fix it! (and ban TukTuks :)

Therefore last year we decided to adopt our current style of development which is as follows.

Fetch-Merge-Push style of development

If Linus can do it, then so can you! We didn't invent this stuff but we did adapt it in a manner which scales for us. The idea is simple, stop devs from committing upstream as a means of sharing changes but instead get them to share peer to peer by fetching from each other. Our git repos are setup so that everyone has read access to clone the repo but only a couple of devs (usually just one dev) can push to a given repo. That means the dev who has push access usually will not need to `pull` prior to pushing because no one else can change it. This is what git was designed to do and yet it's probably one of the least used features among git converts.

Ok so lets get technical shall we?

First off, I'm assuming you know git basics and are already using it. If not then checkout my git article on Digit

Sharing your repo

Let's look at how you can share your git repo with another P2P style. Suppose you have a repo as follows:

```
/home/bud/repos/awesome.repo
```

You can easily share your awesome.repo with anyone on the local networking using git-daemon command by first cd-ing to it's parent directory.

```
$ cd /home/bud/repos
$ git daemon --export-all --base-path=.
```

The above will share all your git repos under the current directory as read-only for others in the network to fetch. Git daemon will run in the foreground by default and so once you're done sharing you can just Ctrl+C it.

Fetching a shared repo

In order to fetch from a fellow dev, you will first need to add him/her as a remote. Chances are you cloned the project from upstream in which case you have just one default remote called origin which points to your upstream repo.

To add another remote for your friend joe for example

```
$ cd /home/bud/repos/awesome.repo
$ git remote add joe git://joes-computer.local/awesome.repo
```

From within your repo, you add a remote using the git remote command. The url above uses git's special `git://` protocol that is understood by the git-daemon, while I'm relying on mDNS to resolve joes-computer.local automatically. If your network(or OS) doesn't support this, then you can just use the IP address.

Finally to fetch joe's changes over the network to your machine issue:

```
$ git fetch joe
```

The above command should give you some feedback as to the success of the fetch operation. Remember that fetch is safe since it only "fetches" as opposed to `pull` which fetches and then tries to merge. So while you could've used `pull` instead of `fetch`, I wouldn't recommend it!

Merging and deleting

Now that you have a copy of the remote changes, what you'd want to do next is to see which branches they were working on. Usually joe will tell you, hey my latest changes are on the `new-cool-feature` branch.

```
$ git branch -a
* master
experimental
remotes/joe/master
remotes/joe/new-cool-feature
```

Git branch will first show your local branches (master, experimental) followed by the remote ones. At this point you should checkout a remote branch you're planning to merge and just make sure everything is working.

```
$ git checkout -b joes-new-cool-feature remotes/joe/new-cool-feature
```

This creates a local branch called “joes-new-cool-feature” which tracks the remote branch `remotes/joe/new-cool-feature` and switches the current HEAD to it. Once your happy then you can switch back to master and merge.

```
$ git checkout master
$ git merge joes-new-cool-feature
```

But… if you have any merge conflicts then you will have to resolve it! If you don’t then your master will remain in a state of CONFLICT. If that sounds like additional work, then do what I do instead of above.

```
$ git checkout master
$ git checkout -b master-merge-joes-new-cool-feature
$ git merge joes-new-cool-feature
$ git checkout master
$ git merge master-merge-joes-new-cool-feature
$ git branch -d master-merge-joes-new-cool-feature
```

Wow that’s a handful of commands to type, you say. Trust me, it beats wasting time trying to resolve someone elses conflict! In above, we fork master as `master-merge-joes-new-cool-feature` in anticipation of a bumpy merge. If things go right, we then merge the merged to master :) The last line is just to delete the temporary branch which we no longer need.

Of course, as you go back and forth merging these micro commits with a dev, you will get into a comfort zone of realizing things won’t go wrong in which case you could merge the remote directly.

```
$ git merge remotes/joe/new-cool-feature
```

It all depends on how much you trust the other dev :) and how much of merge conflict resolution you’re prepared to take on. The branching approach is safer and if you’re the BOSS and you have a merge conflict you can simply abandon the branch and ask your peer dev to fetch from you and fix the conflict and let you know! (Which is what Linus would generally do)

One particularly useful technique to extract a good commit from a potentially set of conflicting ones is to use `Cherry Picking`. If you know the commit log’s SHA1 then you can use that to do a cherry-pick style merge.

```
$ git cherry-pick 623a3dfb5e86f4da4e043f26b6f075f6e3be77ad
```

Working remotely

The issue with git-daemon is that it is more difficult when you’re not on the same network and masked by a bogus IP. One technique is to get the router to DNAT the 9418 port. Another option is to setup a VPN. A third option is to use a bit of SSH tunneling magic to get everything to work. I’ll cover that in a different post, perhaps.

Posted by Buddhika Siddhisena May 14th, 2012 collaboration, git, p2p

Tweet

« A new desktop for a new year Why we started ZoomBA »

Comments

6 Comments Geek-a-holic adventures 1 Login

Recommend Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

td • 4 years ago

Hi, great post, really helpful! One comment about the merge conflicts though. It seems a lot of typing to make another branch to test the merge. Could one just do this:

```
$ git checkout master # make sure there are no uncommitted changes
$ git merge joes-new-cool-feature
```

If there are conflicts that you do not want to fix right now, then:

```
$ git merge --abort
```

• Reply • Share ›

geekaholic Mod → td • 4 years ago

Gr8 tip! Thanks for sharing. I guess you have to be extra careful to make sure everything is committed before attempting. I still feel safer using an intermediate branch, unless it's a small diff.
quoting man git-merge ...

"git merge --abort will abort the merge process and try to reconstruct the pre-merge state. However, if there were uncommitted changes when the merge started (and especially if those changes were further modified after the merge was started), git merge --abort will in some cases be unable to reconstruct the original (pre-merge) changes."

• Reply • Share ›

Seenivasan Thiruvalluvar • 5 years ago

Thanks. But i am getting the below error during fetch

```
git fetch seeni
fatal: read error: Invalid argument
```

• Reply • Share ›

Recent Posts

- VIM Reloaded
- Why We Started ZoomBA
- Peer-to-Peer Collaborative Development Using GIT
- A New Desktop for a New Year
- Track Your New Year Resolutions With ii.do

Copyright © 2014 - Bud - Powered by Octopress