

## 9.1 Git and Other Systems - Git as a Client

The world isn't perfect. Usually, you can't immediately switch every project you come in contact with to Git. Sometimes you're stuck on a project using another VCS, and wish it was Git. We'll spend the first part of this chapter learning about ways to use Git as a client when the project you're working on is hosted in a different system.

At some point, you may want to convert your existing project to Git. The second part of this chapter covers how to migrate your project into Git from several specific systems, as well as a method that will work if no pre-built import tool exists.

### Git as a Client

Git provides such a nice experience for developers that many people have figured out how to use it on their workstation, even if the rest of their team is using an entirely different VCS. There are a number of these adapters, called "bridges," available. Here we'll cover the ones you're most likely to run into in the wild.

#### Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

`git svn`

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

#### Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of an SVN test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
  http://your-svn-server.example.org/svn/
```

This sets up the properties to run the sync. You can then clone the code by running:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .....[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it's ridiculously inefficient, but it's the only easy way to do this.

#### Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You'll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you're importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
   A   m4/acx_pthread.m4
   A   m4/stl_hash.m4
   A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
   A   java/src/test/java/com/google/protobuf/WireFormatTest.java
```

```
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch) 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. If, for example, the test project has only about 75 commits and the codebase isn't that big, Git nevertheless must check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git plumbing command `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bf79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebcd695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

## Committing Back to Subversion

Now that you have a working directory, you can do some work on the project and push your commits back upstream, using Git effectively as an SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r77
M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

## Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ, using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145 c80b6127dd04f5fcd218730ddf3a2da4eb39138 M      README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ, using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a b34372b25ccf4945fe5658fa381b075045e7702a M      README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M      README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M      README.txt
Committed r85
M      README.txt
r85 = 9c29704cc0bbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfd681b661d2a239cc53ef5 and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M      configure.ac
Committed r87
M      autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
M      configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcb1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefdb2b1d92806 and refs/remotes/origin/trunk differ, using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 e757b59a9439312d80d5d43bb65d4a7d0389ed6d M      autogen.sh
First, rewinding head to replay your work on top of it...
```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```
$ git svn rebase
M      autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Running `git svn rebase` every once in a while makes sure your code is always up to date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

## Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an experiment branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M      CHANGES.txt
Committed r89
M      CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
M      COPYING.txt
M      INSTALL.txt
Committed r90
M      INSTALL.txt
M      COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
```

```
No changes between 71af502c214ba13123992338569f4669877f55fd and refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the experiment branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

## Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

## Creating a New SVN Branch

To create a new branch in Subversion, you run `git svn branch [new-branch]`:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera) cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to trunk on the server, not opera.

## Switching Active Branches

Git figures out what branch your `dcommits` go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an opera branch that you can work on separately, you can run:

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your opera branch into trunk (your master branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say "Merge branch opera" instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the `dcommit` makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, opera) after you merge it into trunk.

## Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

### SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
-----
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

-----
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

-----
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommitted don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

### SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```
$ git svn blame README.txt
2   temporal Protocol Buffers - Google's data interchange format
2   temporal Copyright 2008 Google Inc.
2   temporal http://code.google.com/apis/protocolbuffers/
2   temporal
22  temporal C++ Installation - Unix
```

```

22  temporal =====
2   temporal
79   schacon Committing in git-svn.
78   schacon
2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2   temporal Buffer compiler (protoc) execute the following:
2   temporal

```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

#### SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```

$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)

```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

#### Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

#### Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

#### Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this bridge is implemented as a remote helper. The project's name is `git-remote-hg`, and it can be found at <https://github.com/felipec/git-remote-hg>.

#### git-remote-hg

First, you need to install `git-remote-hg`. This basically entails dropping its file somewhere in your path, like so:

```

$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg

```

...assuming `~/bin` is in your `$PATH`. `Git-remote-hg` has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

If you don't have Python installed, visit <https://www.python.org/> and get it first.

The last thing you'll need is the Mercurial client. Go to <https://www.mercurial-scm.org/> and install it if you haven't already.

Now you're ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we'll just use the "hello world" repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

#### Getting Started

Now that we have a suitable “server-side” repository, we can go through a typical workflow. As you’ll see, these two systems are similar enough that there isn’t much friction.

As always with Git, first we clone:

```
$ git clone hg::tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create
* 65bb417 Create a standard 'hello, world' program
```

You’ll notice that working with a Mercurial repository uses the standard `git clone` command. That’s because `git-remote-hg` is working at a fairly low level, using a similar mechanism to how Git’s HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project’s history, and does it fairly quickly.

The `log` command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns out some of these aren’t actually there. Let’s take a look at what’s actually in the `.git` directory:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags
```

9 directories, 5 files

`Git-remote-hg` is trying to make things more idiomatically Git-esque, but under the hood it’s managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with “ac7955c”, which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how `git-remote-hg` maps Git commit hashes to Mercurial changeset IDs. Let’s explore a bit:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefba0e2d9de7ad41a9
```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named “ac9117f” (the SHA-1 hash of the commit pointed to by `master`), with contents “0a04b98” (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don’t have to worry about all of this. The typical workflow won’t be very different from working with a Git remote.

There’s one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it’s likely you don’t want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that’s local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn’t included in commits.

## Workflow

Let’s assume we’ve done some work and made some commits on the `master` branch, and you’re ready to push it to the remote repository. Here’s what our repository looks like right now:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let’s see if anyone else has been doing important work at the same time:

```
$ git fetch
From hg::tmp/hello
   ac7955c..df85e87  master       -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some documentati
```

```
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Since we used the `--all` flag, we see the “notes” refs that are used internally by `git-remote-hg`, but we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we’re not going to do anything fancy.

```
$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
* 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD, refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some documenta
| * | ba04a2a Update makefile
| * | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard 'hello, world' program
```

Perfect. We run the tests and everything passes, so we’re ready to share our work with the rest of the team:

```
$ git push
To hg::/tmp/hello
 df85e87..0c64627 master -> master
```

That’s it! If you take a look at the Mercurial repository, you’ll see that this did what we’d expect:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
| |
| | o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben
| | | Update makefile
| | |
| | o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | | Goodbye
| | |
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
|/ Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard 'hello, world' program
```

The changeset numbered 2 was made by Mercurial, and the changesets numbered 3 and 4 were made by `git-remote-hg`, by pushing commits made with Git.

## Branches and Bookmarks

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a “bookmark,” and it behaves in much the same way as a Git branch.

Mercurial’s concept of a “branch” is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here’s an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset: 6:8f65e5e02793
branch: develop
tag: tip
user: Ben Straub <ben@straub.cc>
date: Thu Aug 14 20:06:38 2014 -0700
summary: More documentation
```

Note the line that begins with “branch”. Git can’t really replicate this (and doesn’t need to; both types of branch can be represented as a Git ref), but `git-remote-hg` needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch] featureA -> featureA
```

That’s all there is to it. On the Mercurial side, it looks like this:

```
$ hg bookmarks
featureA 5:bd5ac26f11f9
$ hg log --style compact -G
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700 ben
| More documentation
|
| o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
| |
| | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | | update makefile
| | |
| | o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | | goodbye
| | |
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
```

```

|/      Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
|   Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
|   Create a standard 'hello, world' program

```

Note the new [featureA] tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a “heavyweight” Mercurial branch also: just put a branch in the branches namespace:

```

$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::tmp/hello
* [new branch]      branches/permanent -> branches/permanent

```

Here's what that looks like on the Mercurial side:

```

$ hg branches
permanent          7:a4529d07aad4
develop            6:8f65e5e02793
default            5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch:    permanent
| tag:       tip
| parent:    5:bd5ac26f11f9
| user:      Ben Straub <ben@straub.cc>
| date:      Thu Aug 14 20:21:09 2014 -0700
| summary:   A permanent change
|
| @ changeset: 6:8f65e5e02793
|/ branch:    develop
| user:      Ben Straub <ben@straub.cc>
| date:      Thu Aug 14 20:06:38 2014 -0700
| summary:   More documentation
|
o changeset: 5:bd5ac26f11f9
|\ bookmark: featureA
| parent:    4:0434aaa6b91f
| parent:    2:f098c7f45c4f
| user:      Ben Straub <ben@straub.cc>
| date:      Thu Aug 14 20:02:21 2014 -0700
| summary:   Merge remote-tracking branch 'origin/master'
[...]
```

The branch name “permanent” was recorded with the changeset marked 7.

From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn't support rewriting history, only adding to it. Here's what our Mercurial repository looks like after an interactive rebase and a force-push:

```

$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
|   A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
|   Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
|   goodbye
|
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| |   A permanent change
| |
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| |/   More documentation
| |
| o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| |\   Merge remote-tracking branch 'origin/master'
| |
| | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | |   update makefile
| |
+---o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| |   goodbye
| |
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
|/   Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
|   Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
|   Create a standard "hello, world" program

```

Changesets 8, 9, and 10 have been created and belong to the permanent branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

## Mercurial Summary

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.



## Git and Bazaar

Among the DVCS, another famous one is [Bazaar](#). Bazaar is free and open source, and is part of the [GNU Project](#). It behaves very differently from Git. Sometimes, to do the same thing as with Git, you have to use a different keyword, and some keywords that are common don't have the same meaning. In particular, the branch management is very different and may cause confusion, especially when someone comes from Git's universe. Nevertheless, it is possible to work on a Bazaar repository from a Git one.

There are many projects that allow you to use Git as a Bazaar client. Here we'll use Felipe Contreras' project that you may find at <https://github.com/felipec/git-remote-bzr>. To install it, you just have to download the file git-remote-bzr in a folder contained in your \$PATH:

```
$ wget https://raw.githubusercontent.com/felipec/git-remote-bzr/master/git-remote-bzr -O ~/bin/git-remote-bzr
$ chmod +x ~/bin/git-remote-bzr
```

You also need to have Bazaar installed. That's all!

### Create a Git repository from a Bazaar repository

It is simple to use. It is enough to clone a Bazaar repository prefixing it by bzr:. Since Git and Bazaar both do full clones to your machine, it's possible to attach a Git clone to your local Bazaar clone, but it isn't recommended. It's much easier to attach your Git clone directly to the same place your Bazaar clone is attached to — the central repository.

Let's suppose that you worked with a remote repository which is at address bzr+ssh://developer@mybazaarserver:myproject. Then you must clone it in the following way:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git
$ cd myProject-Git
```

At this point, your Git repository is created but it is not compacted for optimal disk use. That's why you should also clean and compact your Git repository, especially if it is a big one:

```
$ git gc --aggressive
```

### Bazaar branches

Bazaar only allows you to clone branches, but a repository may contain several branches, and git-remote-bzr can clone both. For example, to clone a branch:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

And to clone the whole repository:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

The second command clones all the branches contained in the emacs repository; nevertheless, it is possible to point out some branches:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Some remote repositories don't allow you to list their branches, in which case you have to manually specify them, and even though you could specify the configuration in the cloning command, you may find this easier:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

### Ignore what is ignored with .bzrignore

Since you are working on a project managed with Bazaar, you shouldn't create a .gitignore file because you *may* accidentally set it under version control and the other people working with Bazaar would be disturbed. The solution is to create the .git/info/exclude file either as a symbolic link or as a regular file. We'll see later on how to solve this question.

Bazaar uses the same model as Git to ignore files, but also has two features which don't have an equivalent into Git. The complete description may be found in [the documentation](#). The two features are:

1. "!!" allows you to ignore certain file patterns even if they're specified using a "!" rule.
2. "RE:" at the beginning of a line allows you to specify a [Python regular expression](#) (Git only allows shell globs).

As a consequence, there are two different situations to consider:

1. If the .bzrignore file does not contain any of these two specific prefixes, then you can simply make a symbolic link to it in the repository: `ln -s .bzrignore .git/info/exclude`.
2. Otherwise, you must create the .git/info/exclude file and adapt it to ignore exactly the same files in .bzrignore.

Whatever the case is, you will have to remain vigilant against any change of .bzrignore to make sure that the .git/info/exclude file always reflects .bzrignore. Indeed, if the .bzrignore file were to change and contained one or more lines starting with "!!" or "RE:", Git not being able to interpret these lines, you'll have to adapt your .git/info/exclude file to ignore the same files as the ones ignored with .bzrignore. Moreover, if the .git/info/exclude file was a symbolic link, you'll have to first delete the symbolic link, copy .bzrignore to .git/info/exclude and then adapt the latter. However, be careful with its creation because with Git it is impossible to re-include a file if a parent directory of that file is excluded.

### Fetch the changes of the remote repository

To fetch the changes of the remote, you pull changes as usually, using Git commands. Supposing that your changes are on the master branch, you merge or rebase your work on the origin/master branch:

```
$ git pull --rebase origin
```

### Push your work on the remote repository

Because Bazaar also has the concept of merge commits, there will be no problem if you push a merge commit. So you can work on a branch, merge the changes into master and push your work. Then, you create your branches, you test and commit your work as usual. You finally push your work to the Bazaar repository:

```
$ git push origin master
```

## Caveats

Git's remote-helpers framework has some limitations that apply. In particular, these commands don't work:

- `git push origin :branch-to-delete` (Bazaar can't accept ref deletions in this way)
- `git push origin old:new` (it will push `old`)
- `git push --dry-run origin branch` (it will push)

## Summary

Since Git's and Bazaar's models are similar, there isn't a lot of resistance when working across the boundary. As long as you watch out for the limitations, and are always aware that the remote repository isn't natively Git, you'll be fine.

## Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is `git-p4`, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

## Git Fusion

Perforce provides a product called Git Fusion (available at <http://www.perforce.com/git-fusion>), which synchronizes a Perforce server with Git repositories on the server side.

### Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from <http://www.perforce.com/downloads/Perforce/20-User>, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:

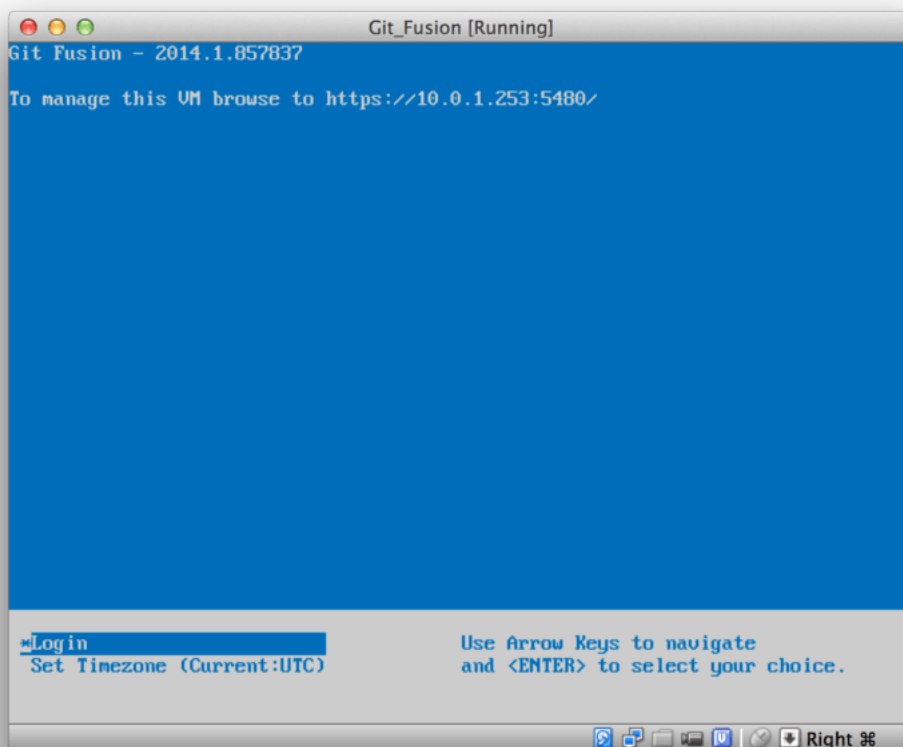


Figure 145. The Git Fusion virtual machine boot screen

You should take note of the IP address that's shown here, we'll be using it later on. Next, we'll create a Perforce user. Select the "Login" option at the bottom and press enter (or SSH to the machine), and log in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing :wq and hitting enter. The second one will prompt you to enter a password twice. That's all we need to do with a shell prompt, so exit out of the session.

The next thing you'll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it's for a domain that won't match your virtual machine's IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce Git Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we're cloning over HTTPS, with the john user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

#### Fusion Configuration

Once you've got Git Fusion installed, you'll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `//git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
├── p4gf_config
├── repos
│   ├── Talkhouse
│   │   └── p4gf_config
├── users
│   └── p4gf_usermap
```

498 directories, 287 files

The objects directory is used internally by Git Fusion to map Perforce objects to Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file, much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ..
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/commmitter field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

#### Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local masterbranch that tracks origin/master. Let's do a bit of work, and create a couple of new commits:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
 d254865..6afeb15 master -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

It looks like someone was! You wouldn't know it from this view, but the 6afeb15 commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```
$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
```

```
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
6afeb15..89cba2b master -> master
```

Git thinks it worked. Let's take a look at the history of the README file from Perforce's point of view, using the revision graph feature of p4v:

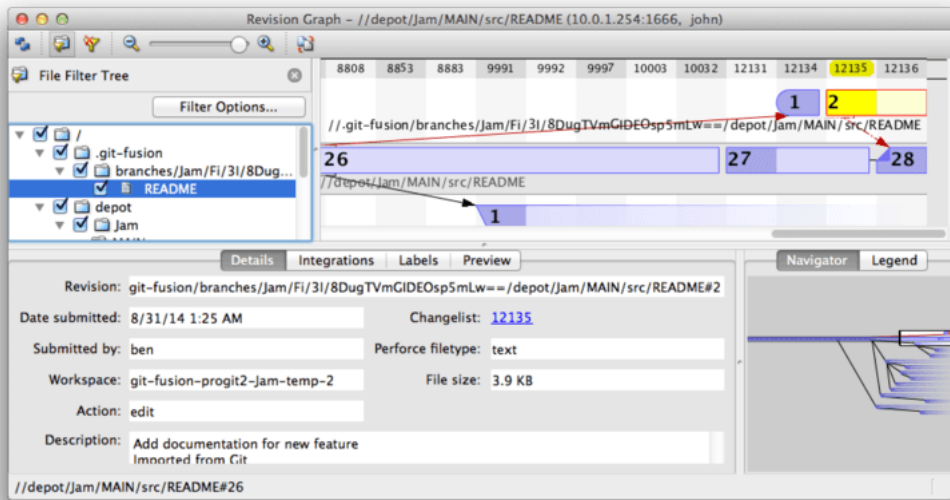


Figure 146. Perforce revision graph resulting from Git push

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the README file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (2 in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the 1 and 2 commits, so it made an "anonymous" branch in the .git-fusion directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

#### Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

#### Git-p4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you won't need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn't as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you'd want to do without being invasive to the server environment.

Note You'll need the p4 tool somewhere in your PATH to work with git-p4. As of this writing, it is freely available at <http://www.perforce.com/downloads/Perforce/20-User>.

#### Setting Up

For example purposes, we'll be running the Perforce server from the Git Fusion OVA as shown above, but we'll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the p4 command-line client (which git-p4 depends on), you'll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

#### Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into refs/remotes/p4/master
```

This creates what in Git terms is a "shallow" clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn't designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it's not enough.

Once it's finished, we have a fully-functional Git repository:

```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from the state at revision #head
```

Note how there's a "p4" remote for the Perforce server, but everything else looks like a standard clone. Actually, that's a bit misleading; there isn't actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to `git log`, but they're not managed by Git itself, and you can't push to them.

#### Workflow

Okay, let's do some work. Let's assume you've made some progress on a very important feature, and you're ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at revision #head
```

We've made two new commits that we're ready to submit to the Perforce server. Let's check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Looks like they were, and `master` and `p4/master` have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but `git p4 rebase` is a shortcut for `git p4 sync` followed by `git rebase p4/master`. It's a bit smarter than that, especially when working with multiple branches, but this is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The `git p4 submit` command will try to create a new Perforce revision for every Git commit between `p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```
# A Perforce Change Specification.
#
# Change:      The change number. 'new' on a new changelist.
# Date:        The date this specification was last modified.
# Client:      The client on which the changelist was created. Read-only.
# User:        The user who created the changelist.
# Status:      Either 'pending' or 'submitted'. Read-only.
# Type:        Either 'public' or 'restricted'. Default is 'public'.
# Description: Comments about the changelist. Required.
# Jobs:        What opened jobs are to be closed by this changelist.
#              You may delete jobs from this list. (New changelists only.)
# Files:       What opened files from the default changelist are to be added
#              to this changelist. You may delete files from this list.
#              (New changelists only.)

Change: new

Client: john_bens-mbp_8487

User: john

Status: new

Description:
    Update link

Files:
    //depot/www/live/index.html  # edit

##### git author ben@straub.cc does not match your p4 account.
##### Use option --preserve-user to modify authorship.
##### Variable git-p4.skipUserNameCheck hides this message.
##### everything below this line is just the diff #####
--- //depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html  2014-08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
<a href="http://www.perforce.com/jam/jam.html">
<a href="jam.html">
```

```
Jam/MR</a>,  
a software build tool.  
</td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which git-p4 has helpfully included. Git-p4 tries to honor your Git and Perforce settings individually when it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they wrote it (and not you).

Git-p4 has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit  
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/  
Synchronizing p4 checkout...  
... - file(s) up-to-date.  
Applying dbac45b Update link  
//depot/www/live/index.html#4 - opened for edit  
Change 12143 created with 1 open file(s).  
Submitting change 12143.  
Locking 1 files ...  
edit //depot/www/live/index.html#5  
Change 12143 submitted.  
Applying 905ec6a Change page title  
//depot/www/live/index.html#5 - opened for edit  
Change 12144 created with 1 open file(s).  
Submitting change 12144.  
Locking 1 files ...  
edit //depot/www/live/index.html#6  
Change 12144 submitted.  
All commits applied!  
Performing incremental import into refs/remotes/p4/master git branch  
Depot paths: //depot/www/live/  
Import destination: refs/remotes/p4/master  
Importing revision 12144 (100%)  
Rebasing the current branch onto remotes/p4/master  
First, rewinding head to replay your work on top of it...  
$ git log --oneline --all --graph --decorate  
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title  
* 05f1ade Update link  
* 75cd059 Update copyright  
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running `git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```
$ git log -1  
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145  
Author: John Doe <john@example.com>  
Date: Sun Aug 31 10:31:44 2014 -0800  
  
Change page title  
  
[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```
$ git log --oneline --all --graph --decorate  
* 3be6fd8 (HEAD, master) Correct email address  
* 1dcbf21 Merge remote-tracking branch 'p4/master'  
|\   
| * c4689fc (p4/master, p4/HEAD) Grammar fix  
* | cbacd0a Table borders: yes please  
* | b4959b6 Trademark  
|/  
* 775a46f Change page title  
* 05f1ade Update link  
* 75cd059 Update copyright  
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```
$ git p4 submit -n  
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/  
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/  
Would apply  
b4959b6 Trademark  
cbacd0a Table borders: yes please  
3be6fd8 Correct email address
```

The `-n` flag is short for `--dry-run`, which tries to report what would happen if the submit command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit  
[...]  
$ git log --oneline --all --graph --decorate  
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address  
* 1b79a80 Table borders: yes please  
* 0097235 Trademark  
* c4689fc Grammar fix  
* 775a46f Change page title  
* 05f1ade Update link  
* 75cd059 Update copyright  
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

## Branching

If your Perforce project has multiple branches, you're not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let's say your Perforce depot is laid out like this:

```
//depot
└─ project
   └─ main
      └─ dev
```

And let's say you have a dev branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
  Importing new branch project/dev

  Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/...
|/
* 2b83451 Project init
```

Note the “@all” specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git's concept of a clone, but if you're working on a project with a long history, it could take a while.

The --detect-branches flag tells git-p4 to use Perforce's branch specs to map the branches to Git refs. If these mappings aren't present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```
$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .
```

Setting the git-p4.branchList configuration variable to main:dev tells git-p4 that “main” and “dev” are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can't mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you'll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you'll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

## Git and Perforce Summary

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it's pretty good at it. However, it's important to remember that Perforce is in charge of the source, and you're only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other people use, don't push any commits that haven't already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

[prev](#) | [next](#)