

# Git, Quilt and Other Kernel Maintenance Tools

James E.J. Bottomley

3 September 2007

## Abstract

The purpose of this paper is to introduce Git, Quilt and other patch maintenance tools with particular emphasis on how a kernel subsystem maintainer would use them, but also trying to be relevant to how they actually work

## 1 Introduction

Linux Kernel development proceeds, by and large, by the provision of patches from an incredibly diverse community of interested parties. Over the years, it has become apparent that tracking the origin and authorship of those contributions is extremely important. Additionally, when the central collecting point (Linus) was getting overloaded, it became evident that a more scalable collection process was needed[1].

### 1.1 History

Bitkeeper was the initial solution to the Linus scalability issues. It attracted criticism by being proprietary, but it was the first tool to address the needs of the kernel community in terms of a fully scalable, fully distributed tool that was operable disconnected from its central repository. While Bitkeeper was Linus' preferred tool, Andrew Morton demonstrated the fallacy that a source control tool was necessary by maintaining the massive amount of patches in his -mm tree purely by use of a set of patch management scripts which were later packaged up as Quilt.

Following the Bitkeeper dust-up[2], it was necessary to come up with a tool which could replace all of its features (including, most importantly the meta-data tracking for submission history); however, in true open source fashion, some of the drawbacks of Bitkeeper could be corrected; in particular, the problem of actually tracking the linear history of the kernel..

## 2 Developer's Certificate of Origin

In 2004 SCO sued IBM for breach of copyright (amongst other things) claiming that IBM had caused code for which SCO owned the copyright to be incorporated into Linux, the SCO suit (being about the origins of code in the Linux kernel) showed the need not only to maintain an actual tree, but also the need to maintain the submission history of all patches into this tree. From this, the Developer Certificate of Origin[3] (all the Signed-off-by: tags) came into being, and Bitkeeper became an essential repository not just of the kernel code but also of the submission history and patch origin of all of the code flowing into the base Linux kernel.

The object of the Developer's Certificate of Origin is to provide a chain of reasonable provenance from the source control meta-data (via the Signed-off-by: tags) back to the original submitter that gives reasonable assurance to anyone looking of where the code in the actual patch came from. The change it introduces in the kernel development methodology is that a source control tool which can add the Signed-off-by: tags to each commit is no longer a nice to have, it becomes mandatory.

## 3 The Origins of Git

After the release of 2.6.11, and the withdrawal of Bitkeeper, a significant number of maintainers turned their attention to replacing its functionality. Having regarded several promising products, Linus concluded that there was only one thing for it and that was to halt kernel development and roll our own.

At this point, it is only fair to point out that git wasn't the only roll your own source control project that was spawned. In response to Larry McVoy's assertion that Bitkeeper was a superlatively developed tool that had taken two years to get right and would be impossible to recreate in Open Source, Matt Mackall started his own project: Mercurial[4] (often abbreviated to hg, the Chemical symbol for Mercury), the name for which is derived from the personality traits of the person precipitating the controversy in the first place. Mercurial certainly developed surprisingly rapidly into a full featured distributed source control tool, accreting its own group of committed users along the way, in a timescale that almost exactly matched that of git (the actual "who was first" question is slightly controversial and will not be addressed). However, git, by virtue of being created by its eventual user is still the tool of choice for maintaining kernel trees that are fed directly to

Linus. Thus, this paper will not mention Mercurial further (reflecting the author's unfamiliarity with it).

In fact, once git became stable enough to release a kernel with it (which was the 2.6.12 kernel), Linus passed the git maintainership over to Junio Hamano and once more resumed his place at the head of the kernel development tree.

The only other historically significant event with git was the maturity of the merge system. How to do merges had been an ongoing topic on the git mailing list for a while, with several people working on algorithms and scripts. As a result of this effort, on 18 April 2005, the first ever automated merge was made remotely from one git tree to another[5].

### 3.1 The Fundamentals

In the very inception of Git, Linus decreed that it wouldn't be a difference tracking tool, it would be a tree tracking tool. This means that the fundamental entity represented by each node in the history of the project would be a complete source tree. There would be nothing, other than the commit history itself linking these trees. This is a radical departure from most other forms of source control which try to track the differences between the files in the tree as the project evolves. One of the fundamental reasons for tracking differences instead of the actual files is that from one commit to the next, the amount of code changed is usually a tiny fraction of the total source base ... thus tracking complete source trees would necessarily entail a tremendous amount of duplication and a concomitant huge amount of storage space

### 3.2 Content Accessible Objects

The problem of tree duplication was solved in a very elegant way: since most of the files would remain constant between commits, each file was fingerprinted by its SHA1 hash and then filed under this fingerprint in the git object repository. This classification by content (where Content Accessible Objects comes from) means that however many times the same file appears in all the commits, its contents and hence SHA1 hash must be the same, thus all these occurrences point at the same SHA1 object, thus reducing the duplication significantly. Today, if you look at the objects directory of a git tree (under `.git/objects` you will see the initial two digits of the SHA1 fingerprint, so that git has a simple two level look-up to find objects.

### 3.3 Object Packing

It should still be apparent to the reader that although a content based object classification system solves the problem of multiple copies of the same object, it is still an inefficient way of storing objects which are substantially similar but not exactly identical (think of a 10,000 line file which has had only one of those lines changed). This problem is solved by packing: a technique whereby the similar objects in the object repository are identified and compressed together, thus eliminating most of their redundancy. Since packing has no impact at all on the operation of git (it's really just a mechanism for reducing the size of the repository), it won't be discussed further.

### 3.4 Objects in Git

Once it was realised that fingerprinted SHA1 objects would solve the file copy proliferation problem, they were actually pressed into service as the fundamental storage elements for everything. Today, git has five primary objects: commits, trees, blobs and tags. Additionally, every object is referenced by its SHA1 fingerprint.

Blobs are the simplest objects: they represent the actual files stripped of any meta-data information (including the file name: two differently named files in git with the same content would be represented by the same blob).

Tree objects in fact are where the file names are stored. They are very similar to current file-system directory objects: They link names to either blobs or other trees (blobs representing files and trees representing lower level directories) and the linking, naturally being done by the SHA1 fingerprint and type of the relevant object. Note in particular, that there's no other information in a tree object, so the per file comment meta-data that's often associated with other source control systems is absent in git.

Commits represents the fundamental chain of history in the repository. They have a number of predefined fields: a tree (representing the actual source tree state as of this commit), one or more parents (simple SHA1 pointers to other commit objects which logically preceded this one) the author, committer and date information followed by free form text describing the commit.

Tags are really just information pointing to particular commits. Linus uses tags to label the tree when he makes a release. The main point about tags is that they act as invariant points in the tree that are easy to find again.

## 4 Git

Following on from the origins, we will now describe how git can be used as the basis for a full featured, distributed project repository capable of filling all the needs of the Linux Kernel.

### 4.1 Making a Repository

Obviously, the objects aren't by themselves sufficient to define a repository, you also need to know where to begin. This beginning point is called the head commit (or simply head) of the repository. One can use this head to display the entire commit history of the repository. Additionally, one needs some way to check the repository out and to determine if there are any local differences between the head and the checked out repository. This tracking is done by the index file. It simply records the current state of the checked out files by time-stamp (so we don't have to do expensive SHA1 checking unless the timestamps actually differ).

### 4.2 Branches

Since the head tracks the current end point of the repository, it will be no surprise to discover that there can be more than one of these. In fact git does branches (which are forks away from the linear tree) simply by having multiple heads. Each of the heads maintained by git can be seen in the `.git/refs/heads` directory, with the `master` representing the primary head of the tree. Along with this, the `.git/refs/tags` directory represents a cache of the currently interesting tags. Although tags can be reconstructed from scanning the repository, whereas heads might not be.

### 4.3 Merging

When two (or more) branches of a tree come back together, this is termed a merge. As far as git is concerned, a merge is simply a tree with multiple parents (and sometimes there are even more than two: Paul Mackerras once famously did an octopus merge of eight separate branches in a single commit). The key thing to remember here is that git doesn't in the least care *how* the merge is done; all it wants to see is a list of parents and the final merged tree. This, therefore, makes the actual merge algorithms almost infinitely pluggable. Indeed, if you look at the current git merge algorithm, you'll see it consists of at least four separate mechanisms, which it tries sequentially to see what actually produces the best fit.

A substantial amount of work has gone into git to enhance the merge tools (including the contribution of a nice graphical interface by Ted Ts'o).

#### 4.4 Adding, Removing and Renaming files

Historically, tracking these operations has been a real problem for source control systems. However, git really has no such issues: A removed file simply disappears from the tree in the next commit, likewise an added file simply appears. A renamed file, providing it is a pure rename not associated with content changes, simply shows up as two different tree names pointing to the same blob.

#### 4.5 File Histories

Sometimes, it is vitally important not simply to look at the complete history but to look at slices of it that are relevant to individual files or directories. One of the drawbacks of the git “we only track trees” philosophy is that the information about the individual file is present, but hard to extract. What git actually does when asked for this information is to traverse the entire commit tree looking for places where the file or directory being tracked was changed (and prints out only the commits that changed it). This method even works across renames, because the tool can see the same blob with different names across different commits. However, it becomes problematic when the file was not only renamed but had its contents changed. Here, git can do statistical analysis between all files in the tree to see if two are substantially similar, and therefore were possibly a rename with content change. However, this method is only statistical and doesn't give a definitive match. The moral of this is that if you're going to change a file's name and its contents, do it in two separate commits so we can preserve the rename definitively in the meta-data.

This leads to the final observation that all operations that were possible with conventional file difference tracking source control tools are also possible with git ... it just might require more sophisticated analysis and more processing power to derive the information.

## 5 Using Git

This paper isn't going to be a how to guide for git. For most of the commands you can look them up yourself using

```
git help
```

However, there are one or two incredibly useful commands that bear further scrutiny.

## 5.1 git rebase

A rebase simply means to take an existing git patch set and base it off a new tree (altering all of the commit ids in the process). The true advantage of a rebase instead of reapplying patches to the top of the tree is that git can use its set of merge algorithms to perform the rebase and because it knows the prior commit for each patch it is able to perform this merger far more efficiently than a simple diff based patch. For this reason, if a quilt set fails to rebase, it is often easier to import the quilt to git, get git to do the rebase then re-export to quilt (using `git format-patch`) than to try to fix up the quilt rejections.

## 5.2 git bisect

This is one of the classic and most useful debugging tools in Linux Kernel development today. Given the problem of something broke my *xxx* it allows even a novice to find the patch that caused the breakage (provided they're capable of building their own kernel). What git bisect allows you to do is to specify the last working and first broken kernels. You do this by

```
git bisect begin
git bisect good sha1 of last good commit
git bisect bad sha1 of first bad commit
Git will respond with
Bisecting: 159 revisions left to test after this
```

And you go on to build the kernel and try it out. In each case typing `git bisect good` or `git bisect bad` depending on whether your bug shows up. Eventually (in  $O(\log N)$  of the  $N$  commits) git will tell you which the bad commit is and this is the one you should either fix or report.

## 6 Quilt

Quilt is essentially a patch management toolkit. Where it differs from git is that it has no permanent history: The patches themselves can have a description field which usually equates to the commit meta-data in git. Where quilt differs from git is in the anchoring of the patch set (quilt is simply a series of patches above an extracted source base, without any definitive identification of the actual source base being patched). Quilt does contain a

simple series file identifying the patch sequence to be applied to the source base. The simple beauty of quilt over git is that it is incredibly easy to add and remove patches and even to re-order them. For this reason, not just Andrew, but also a growing number of kernel maintainers are adopting quilt series as their preferred method of storing patches.

## References

- [1] Rob Landley *A modest proposal — We need a patch penguin* Email Archive: <http://lkml.org/lkml/2002/1/28/83> 28 January 2002
- [2] Linus Torvalds *Kernel SCM Saga ...* Email Archive: <http://lkml.org/lkml/2005/4/6/121> 6 April 2005
- [3] *Developer Certificate of Origin* Linux Kernel Documentation: [Documentation/SubmittingPatches](#) section 12
- [4] Matt Mackall *Mercurial v0.1 - a minimal scalable distributed SCM* Email Archive: <http://lkml.org/lkml/2005/4/20/45> 20 April 2005
- [5] Linus Torvalds *Merge SCSI tree from James Bottomley*. Commit id `0b2cad2f30d0353f2576b1a2207c0792ba713157` 18 April 2005