Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

GIT—A Stupid Content Tracker

Junio C. Hamano
Twin Sun, Inc.
junio@twinsun.com

Abstract

Git was hurriedly hacked together by Linus Torvalds, after the Linux kernel project lost its license to use BitKeeper as its source code management system (SCM). It has since quickly grown to become capable of managing the Linux kernel project source code. Other projects have started to replace their existing SCMs with it.

Among interesting things that it does are:

1. giving a quick whole-tree diff,
2. quick, simple, stupid-but-safe merge,
3. facilitating e-mail based patch exchange workflow, and
4. helping to pin-point the change that caused a particular bug by a bisection search in the development history.

The core git functionality is implemented as a set of programs to allow higher-layer systems (Porcelains) to be built on top of it. Several Porcelains have been built on top of git, to support different workflows and individual taste of users. The primary advantage of this architecture is ease of customization, while keeping the repositories managed by different Porcelains compatible with each other.

The paper gives an overview of how git evolved and discusses the strengths and weaknesses of its design.

1 Low level design

Git is a "stupid content tracker." It is designed to record and compare the whole tree states efficiently. Unlike traditional source code control systems, its data structures are not geared toward recording changes between revisions, but for making it efficient to retrieve the state of individual revisions.

The unit in git storage is an *object*. It records:

- `blob` – the contents of a file (either the contents of a regular file, or the path pointed at by a symbolic link).
- `tree` – the contents of a directory, by recording the mapping from names to objects (either a blob object or a tree object that represents a subdirectory).
- `commit` – A commit associates a tree with meta-information that describes how the tree came into existence. It records:
  - The tree object that describes the project state.

  - The author name and time of creation of the content.
  - The committer name and time of creation of the commit.
  - The parent commits of this commit.
  - The commit log that describes why the project state needs to be changed to the tree contained in this commit from the trees contained in the parent commits.

- `tag` – a tag object names another object and associates arbitrary information with it. A person creating the tag can attest that it points at an authentic object by GPG-signing the tag.

Each object is referred to by taking the SHA1 hash (160 bits) of its internal representation, and the value of this hash is called its object name. A tree object maps a pathname to the object name of the blob (or another tree, for a subdirectory).

By naming a single tree object that represents the top-level directory of a project, the entire directory structure and the contents of any project state can be recreated. In that sense, a tree object is roughly equivalent to a tarball.

A commit object, by tying its tree object with other commit objects in the ancestry chain, gives the specific project state a point in project history. A merge commit ties two or more lines of developments together by recording which commits are its parents. A tag object is used to attach a label to a specific commit object (e.g. a particular release).

These objects are enough to record the project history. A project can have more than one lines of developments, and they are called *branches*. The latest commit in each line of development is called the head of the branch, and a repository keeps track of the heads of currently active branches by recording the commit object names of them.

To keep track of what is being worked on in the user's working tree, another data structure called *index*, is used. It associates pathnames with object names, and is used as the staging area for building the next tree to be committed.

When preparing an index to build the next tree, it also records the `stat(2)` information from the working tree files to optimize common operations. For example, when listing the set of files that are different between the index and the working tree, git does not have to inspect the contents of files whose cached `stat(2)` information match the current working tree.

The core git system consists of many relatively low-level commands (often called Plumbing) and a set of higher level scripts (often called Porcelain) that use Plumbing commands. Each Plumbing command is designed to do a specific task and only that task. The Plumbing commands to move information between the recorded history and the working tree are:

- Files to index. `git-update-index` records the contents of the working tree files to the index; to write out the blob recorded in the index to the working tree files, `git-checkout-index` is used; and `git-diff-files` compares what is recorded in the index and the working tree files. With these, an index is built that records a set of files in the desired state to be committed next.

- Index to recorded history. To write out the contents of the index as a tree object, `git-write-index` is used; `git-commit-tree` takes a tree object, zero or more commit objects as its parents, and the commit log message, and creates a commit object. `git-diff-index`

compares what is recorded in a tree object and the index, to serve as a preview of what is going to be committed.

- Recorded history to index. The directory structure recorded in a tree object is read by `git-read-tree` into the index.
- Index to files. `git-checkout-index` writes the blobs recorded in the index to the working tree files.

By tying these low-level commands together, Porcelain commands give usability to the whole system for the end users. For example, `git-commit` provides a UI to ask for the commit log message, create a new commit object (using `git-write-tree` and `git-commit-tree`), and record the object name of that commit as the updated topmost commit in the current line of development.

2 Design Goals

From the beginning, git was designed specifically to support the workflow of the Linux kernel project, and its design was heavily influenced by the common types of operations in the kernel project. The statistics quoted below are for the 10-month period between the beginning of May 2005 and the end of February 2006.

- The source tree is fairly large. It has approximately 19,000 files spread across 1,100 directories, and it is growing.
- The project is very active. Approximately 20,000 changes were made during the 10-month period. At the end of the examined period, around 75% of the lines are from the version from the beginning of the period, and the rest are additions and modifications.
- The development process is highly distributed. The development history led to v2.6.16 since v2.6.12-rc2 contains changes by more than 1,800 authors that were committed by a few dozen people.
- The workflow involves many patch exchanges through the mailing list. Among 20,000 changes, 16,000 were committed by somebody other than the original author of the change.
- Each change tends to touch only a handful files. The source tree is highly modular and a change is often very contained to a small part of the tree. A change touches only three files on average, and modifies about 160 lines.
- The tree reorganization by addition and deletion is not so uncommon, but often happens over time, not as a single rename with some modifications. 4,500 files were added or deleted, but less than 600 were renamed.
- The workflow involves frequent merges between subsystem trees and the mainline. About 1,500 changes are merges (7%).
- A merge tends to be straightforward. The median number of paths involved in the 1,500 merges was 185, and among them, only 10 required manual inspection of content-level merges.

Initial design guidelines came from the above project characteristics.

- A few dozen people playing the integrator role have to handle work by 2,000 contributors, and it is paramount to make it efficient form them to perform common operations, such as patch acceptance and merging.

- Although the entire project is large, individual changes tend to be localized. Supporting patch application and merging with a working tree with local modifications, as long as such local modifications do not interfere with the change being processed, makes the integrators' job more efficient.

- The application of an e-mailed patch must be very fast. It is not uncommon to feed more than 1,000 changes at once during a sync from the `-mm` tree to the mainline.

- When existing contents are moved around in the project tree, renaming of an entire file (with or without modification at the same time) is not a majority. Other content movements happen more often, such as consolidating parts of multiple files to one new file or splitting an existing files into multiple new files. Recording file renames and treating them specially does not help much.

- Although the merge plays an important role in building the history of the project, clever merge algorithms do not make much practical difference, because majority of the merges are trivial; nontrivial cases need to be examined carefully by humans anyway, and the maintainer can always respond, "This does not apply, please rework it based on the latest version and resubmit." Faster merge is more important, as long as it does not silently merge things incorrectly.

- Frequent and repeated merges are the norm. It is important to record what has already been merged in order to avoid having to resolve the same merge conflicts over and over again.

- Two revisions close together tend to have many common directories unchanged between them. Tree comparison can take advantage of this to avoid descending into subdirectories that are represented by the same tree object while examining changes.

3 Evolution

The very initial version of git, released by Linus Torvalds on April 7, 2005, had only a handful of commands to:

- initialize the repository;
- update the index to prepare for the next tree;
- create a tree object out of the current index contents;
- create a commit object that points at its tree object and its parent commits;
- print the contents of an object, given its object name;
- read a tree object into the current index;
- show the difference between the index and the working tree.

Even with this only limited set of commands, it was capable of hosting itself. It needed scripting around it even for "power users."

By the end of the second week, the Plumbing level already had many of the fundamental data structure of today's git, and the initial commit of the modern Linux kernel history hosted on git (v2.6.12-rc2) was created with this version. It had commands to:

- read more than one tree object to process a merge in index;

- perform content-level merges by iterating over an unmerged index;
- list the commit ancestry and find the most recent common commit between two lines of development;
- show differences between two tree objects, in the raw format;
- fetch from a remote repository over rsync and merge the results.

When two lines of development meet, git uses the index to match corresponding files from the common ancestor (merge base) and the tips of the two branches. If one side changed a file while the other side didn't, which often happens in a big project, the merge algorithm can take the updated version without looking at the contents of the file itself. The only case that needs the content-level merge is when both side changed the same file. This tree merge optimization is one of the foundations of today's git, and it was already present there. The first ever true git merge in the Linux kernel repository was made with this version on April 17, 2005.

By mid May, 2005, it had commands to:

- fetch objects from remote repositories over HTTP;
- create tags that point at other objects;
- show differences between the index and a tree, working tree files and a tree, in addition to the original two tree comparison commands—both raw and patch format output were supported;
- show the commit ancestry along with the list of changed paths.

By the time Linus handed the project over to the current maintainer in late July 2005, the core part was more or less complete. Added during this period were:

- packed archive for efficient storage, access, and transfer;
- the git "native" transfer protocol, and the `git-daemon` server;
- exporting commits into the patch format for easier e-mail submission.
- application of e-mailed patches.
- rename detection by diff commands.
- more "user friendliness" layer commands, such as `git-add` and `git-diff` wrappers.

The evolution of git up to this point primarily concentrated on supporting the people in the integrator role better. Support for individual developers who feed patches to integrators was there, but providing developers with more pleasant user experiences was left to third-party Porcelains, most notably Cogito and StGIT.

4 Features and Strengths

This section discusses a few examples of how the implementation achieves the design goals stated earlier.

4.1 Patch flows

There are two things git does to help developers with the patch-based workflow.

Generating a patch out of a git-managed history is done by using the `git-diff-tree` command, which knows how to look at only subtrees that are actually different in two trees for efficient patch generation.

The diff commands in git can optionally be told to detect file renames. When a file is renamed and modified at the same time, with this option, the change is expressed as a diff between the file under the old name in the original tree and the file under the new name in the updated tree. Here is an example taken from the kernel project:

```
5e7b83ffc67e15791d9bf8b2a18e4f5fd0eb69b8
diff --git a/arch/um/kernel/sys_call_table....
similarity index 99%
rename from arch/um/kernel/sys_call_table.c
rename to arch/um/sys-x86_64/sys_call_table.c
index b671a31..3f5efbf 100644
--- a/arch/um/kernel/sys_call_table.c
+++ b/arch/um/sys-x86_64/sys_call_table.c
@@ -16,2 +16,8 @@ #include "kern_util.h"

+#ifdef CONFIG_NFSD
+#define NFSSERVCTL sys_nfsservctl
+#else
+#define NFSSERVCTL sys_ni_syscall
+#endif
+
 #define LAST_GENERIC_SYSCALL __NR_keyctl
```

This is done to help reviewing such a change by making it easier than expressing it as a deletion and a creation of two unrelated files.

The committer (typically the subsystem maintainer) keeps the tip of the development branch checked out, and applies e-mailed patches to it with `git-apply` command. The command checks to make sure the patch applies cleanly to the working tree, paths affected by the patch in the working tree are unmodified, and the index does not have modification from the tip of the branch. These checks ensure that after applying the patch to the working tree and the index, the index is ready to be committed, even when there are unrelated changes in the working tree. This allows the subsystem maintainer to be in the middle of doing his own work and still accept patches from outside.

Because the workflow git supports should not require all participants to use git, it understands both patches generated by git and traditional diff in unified format. It does not matter how the change was prepared, and does not negatively affect contributors who manage their own patches using other tools, such as Andrew Morton's patch-scripts, or quilt.

4.2 Frequent merges

A highly distributed development process involves frequent merges between different branches. Git uses its commit ancestry relation to find common ancestors of the branches being merged, and uses a three-way merge algorithm at two levels to resolve them. Because merges tend to happen often, and the subprojects are highly modular, most of the merges tend to deal with cases where only one branch modifies paths that are left intact by the other branch. This common case is resolved within the index, without even having to look at the contents of files. Only paths that need content-level merges are given to an external three-way merge program (e.g. "merge" from the RCS suite) to be processed. Similarly to the patch-application process, the merge can be done as long as the index does not have modification from the tip of the branch and there is no change to the working tree files that are involved in the merge, to allow the integrator to have local changes in the working tree.

While merging, if one branch renamed a file from the common ancestor while the other branch kept it at the same location (or renamed it to a different location), the merge algorithm

notices the rename between the common ancestor and the tips of the branches, and applies three-way merge algorithm to merge the renames (i.e. if one branch renamed but the other kept it the same, the file is renamed). The experiences by users of this "merging renamed paths" feature is mixed. When merges are frequently done, it is more likely that the differences in the contents between the common ancestor and the tip of the branch is small enough that automated rename detector notices it.

When two branches are merged frequently with each other, there can be more than one closest common ancestor, and depending on which ancestor is picked, the three-way merge is known to produce different results. The merge algorithms git uses notice this case and try to be safe. The faster "resolve" algorithm leaves the resolution to the end-user, while the more careful "recursive" algorithm first attempts to merge the common ancestors (recursively—hence its name) and then uses the result as the merge base of the three-way merge.

4.3 Following changes

The project history is represented as a parent-child ancestry relation of commit objects, and the Plumbing command `git-rev-list` is used to traverse it. Because detecting subdirectory changes between two trees is a very cheap operation in git, it can be told to ignore commits that do not touch certain parts of the directory hierarchy by giving it optional pathnames. This allows the higher-level Porcelain commands to efficiently inspect only "interesting" commits more closely.

The traversal of the commit ancestry graph is also done while finding a regression, and is used by the `git-bisect` command. This traversal can also be told to omit commits that do not touch a particular area of the project directory; this speeds up the bug-hunting process when the source of the regression is known to be in a particular area.

4.4 Interoperating with other SCM

A working tree checked out from a foreign SCM system can be made into a git repository. This allows an individual participant of a project whose primary SCM system is not git to manage his own changes with git. Typically this is done by using two git branches per the upstream branch, one to track the foreign SCM's progress, another to hold his own changes based on that. When changes are ready, they are fed back by the project's preferred means, be it committing into the foreign SCM system or sending out a series of patches via e-mail, without affecting the workflow of the other participants of the projects. This allows not just distributed development but distributed choice of SCM. In addition, there are commands to import commits from other SCM systems (as of this writing, supported systems are: GNU arch, CVS, and Subversion), which helps to make this process smoother.

There also is an emulator that makes a git repository appear as if it is a CVS repository to remote CVS clients that come over the `pserver` protocol. This allows people more familiar with CVS to keep using it while others work in the same project that is hosted on git.

4.5 Interoperating among git users

Due to the clear separation of the Plumbing and Porcelain layers, it is easy to implement higher-level commands to support different workflows on top of the core git Plumbing commands. The Porcelain layer that comes with the core git requires the user to be fairly familiar with how the tools work internally, especially how the index is used. In order to make effective use of the

tool, the users need to be aware that there are three levels of entities: the histories recorded in commits, the index, and the working tree files.

An alternative Porcelain, Cogito, takes a different approach by hiding the existence of the index from the users, to give them a more traditional two-level world model: recorded histories and the working tree files. This may fall down at times, especially for people playing the integrator role during merges, but gives a more familiar feel to new users who are used to other SCM systems.

Another popular tool based on git, StGIT, is designed to help a workflow that depends more heavily on exchange of patches. While the primary way to integrate changes from different tracks of development is to make merges in the workflow git and Cogito primarily targets, StGIT supports the workflow to build up piles of patches to be fed upstream, and re-sync with the upstream when some or all of the patches are accepted by rebuilding the patch queue.

While different Porcelains can be used by different people with different work habits, the development history recorded by different Porcelains are eventually made by the common Plumbing commands in the same underlying format, and therefore are compatible with each other. This allows people with different workflow and choice of tools to cooperate on the same project.

5 Weakness and Future Works

There are various missing features and unfinished parts in the current system that require further improvements. Note that the system is still evolving at a rapid pace and some of the issues listed here may have already been addressed when this paper is published.

5.1 Partial history

The system operates on commit ancestry chains to perform many of the interesting things it does, and most of the time it only needs to look at the commits near the tip of the branches. An obvious example is to look at recent development histories. Merging branches need access to the commits on the ancestry chain down to the latest common ancestor commit, and no earlier history is required. One thing that is often desired but not currently supported is to make a "shallow" clone of a repository that records only the recent history, and later deepen it by retrieving older commits.

Synchronizing two git repositories is done by comparing the heads of branches on both repositories, finding common ancestors and copying the commits and their associated tree and blob objects that are missing from one end to the other. This operation relies on an invariant that all history behind commits that are recorded as the heads of branches are already in the repository. Making a shallow clone that has only commits near the tip of the branch violates this invariant, and a later attempt to download older history would become a no-operation. To support "shallow" cloning, this invariant needs to be conditionally lifted during "history deepening" operation.

5.2 Subprojects

Multiple projects overlayed in a single directory are not supported. Different repositories can be stored along with their associated working trees in separate subdirectories, but currently there is no support to tie the versions from the different subprojects together.

There have been discussions on this topic and two alternative approaches were proposed, but

there were not enough interest to cause either approach to materialize in the form of concrete code yet.

5.3 Implications of not recording renames

In an early stage of the development, we decided not to record rename information in trees nor commits. This was both practical and philosophical.

Files are not renamed that often, and it was observed that moving file contents around without moving the file itself happened just as often. Not having to record renames specially, but always recording the state of the whole tree in each revision, was easier to implement from a practical point of view.

When examining the project history, the question "*where did this function come from, and how did it get into the current form?*" is far more interesting than "*where did this file come from?*" and when the former question is answered properly (i.e. "*it started in this shape in file* X*, but later assumed that shape and migrated to file* Y"), the latter becomes a narrow special case, and we did not want to only support the special case. Instead, we wanted to solve the former problem in a way general enough to make the latter a non-issue. However, deliberately not recording renames often contradicts people's expectations.

Currently we have a merge strategy that looks at the common ancestor and two branch heads being merged to detect file renames and try to merge the contents accordingly. If the modifications made to the file in question across renames is too big, the rename detection logic would not notice that they are related. It is possible for the merge algorithm to inspect all commits along the ancestry chain to make the rename detection more precise, but this would make merges more expensive.

6 Conclusion

Git started as a necessity to have a minimally usable system, and during its brief development history, it has quickly become capable of hosting one of the most important free software projects, the Linux kernel. It is now used by projects other than the kernel (to name a few: Cairo, Gnumeric, Wine, xmms2, the X.org X server). Its simple model and tool-based approach allow it to be enhanced to support different workflows by scripting around it and still be compatible with other people who use it. The development community is active and is growing (about 1500 postings are made to the mailing list every month).