Final Project 8 – Godzillasaurus Rex

Andrew Kuhnhausen
Colton Meyers
Landon Gilbert-Bland
Alex Clemmer

Abstract

Typically applications that want to implement `git` functionality link into `libgit.a` in the main `git` directory. This is an unfortunate choice because if something goes wrong, `libgit` literally just calls `die()`. In contrast, `libgit2` is a source-level library that can be compiled and integrated fully into an application.

Our project is to implement as much functionality into `libgit2` as we possibly can in 3 1/2 weeks. Vicent Marti has agreed to mentor us, and we have been in constant contact over both IM and email. Our goal specifically will be to implement `diff` and `merge`.

1 Summary

`Git` is a robust Distributed Version Control System (DCVS) that concentrates mainly on making management of large numbers of distributed, disparate code sources both sane and fast. At the center of this model is fast and cheap merging and branching. Users can pull code from any repository, and merge it with existing code they already have, even if it's *completely unrelated.* They can branch their own projects and merge them back at will which is so uncommon among more "traditional" VCSs (*e.g.*, `cvs` and `svn`) that it is often hard for their longtime users to even understand how dramatically this changes the development model.

This attitude is pervasive and evident throughout git's design: for example, git stores a copy *for every version of every file in the repository*, which stands in stark contrast to systems like `cvs` and `svn`, whose commits are purely a list of changes since the last commit. This sounds expensive, but it isn't: the entire history of the Linux kernel only takes up about 1.5 times the total memory footprint of the kernel itself. And by storing every version of every object in the object store, merges become a *lot* easier. And faster. This is the primary advantage of DVCSs, and it is probably mostly responsible for their advent (*i.e.*, the rise of Mercurial, Bazaar, et al).

One major issue for people who want to integrate git into their applications is that this is not really encouraged by git at a core level. It's true that you can link to `libgit.a` from the main git repository, but when something (*e.g.*, a merge) fails, `libgit.a` literally just calls `die()`. This is obviously a highly undesirable result, and this is particularly because `libgit2` is an *incredibly* useful project. For example, distributed filesystems (*e.g.* Hadoop Filesystem) require the ability to quickly compute and update content in nodes, and a content management engine (which is what git essentially is) would be extremely well suited to this task. This is one of the motivations behind the `libgit2` project.

Our project currently is to implement the diff algorithm. The next step is to implement the merge functionality. The main goal here is speed: our job it so make this as fast as possible, and if possible, faster than git. If this seems a pants-on-head ridiculous, we should mention that we have a secret weapon that is available to us, and not to core git: threads.

There are currently two frontrunners for the diff algorithm: the O(ND) "classical" algorithm implemented as the git core diff algorithm, and Patience Diff, which is the default diff algorithm implemented in Bazaar (a competing DVCS) and JGit. It's not immediately clear which is faster, or whether either can be parallelized to any extent (not likely), but Patience seems to be *much* easier to write. On the other hand the git diff code is more or less a direct rip of `libxdiff`, so we do have reference implementations in both cases.

That's important, because the main challenge here will not be actually writing the diff so much as making sure it both plays nicely with existing code, and doesn't make any stupid design decisions that will set the project back later. In particular, it's important that our internal diff representation is consistent and efficient, as it is this representation alone that makes critical functions like merge even possible. The way we deal with this is simple: we maintain close contact with our adviser, Vicent Marti, who is a core contributor. So far we have talked with him every day since starting the project, so we are confident that this will work out.

To this end, we've dedicated half our team to implementing the structural code around the diff algorithm, and half to actually write the algorithm. Both teams will have to spend a significant amount of time looking through existing code and building into it. This actually, at least in the case of git, represents a fair challenge: git is *highly* optimized for speed, and uses all sorts of wizardry. For example, to add an object to the git index, they use a very fast hash table implemented using an array and a rolling Rabin Checksum as a key. Some of it is truly admirable, and it is pretty clean, but it isn't what we would call extremely readable.

The first week is dedicated to this task alone. The next weeks will be dedicated to implementing various parts of the merge functionality. There are of course lots of *types* of merge used in git–octopus, recursive, fast-forward, *et al.* This is not necessarily as hard as it sounds: most of the git object code is written (*e.g.*, the git object store, index, and so on), so really what we need to do is to figure out the interface and implement the actual functionality.

At this point the rest of the plan is nebulous and will be largely determined by what we find out about the project in during our diff implementation. But in the meantime we have lots of things to do; the one saving grace of the project is that we really like it a lot.

2 Key Features

2.1 Diff algorithm

A hopefully-threaded implementation of either the classical O(ND) diffing algorithm or the Patience algorithm, this will be responsible for many *very* important features in `libgit2`, including merge; it is important not just to get this right, but to make it *fast*, and should include at least a 1000 lines of code, if you count all the core code required to interface with *e.g.* the git index and object store, which is very much required for this to work.

2.2 Diff internal protocol implementation

One important task for diff is to make information available for a lot of really important functions, like merge and pull; building this interface code should require at least a couple thousand lines of code, and will require touching a *ton* of the internal git object code, which already exits in `libgit2`. This will sit around the core diff function and wrap it for other functionality.

2.3 Depending on how much work that ends up being:

We may also implement various parts of **merge**. We hope we can get around to this, but it's hard to know how much diff is going to be before we're done.

3 Team Member Priorities

3.1 Colton Meyers

is mainly responsible for building the protocol code around the core diff algorithm. Probably he will be dealing with the object store code, although it is not immediately obvious that this can be completely decoupled from the other teammate working on protocol, Landon. The object code will be responsible for *e.g.* pulling blobs out of the object store and making them diff-able.

3.2 Landon Gilbert-Bland

is also mainly responsible for the protocol code, but he will probably be dealing with the code around git's index, although there will be a lot of overlap with Colton's responsibilities. This will probably include dealing with the horrifying Rabin Fingerprinting work.

3.3 Andrew Kuhnhausen

is primarily responsible, with Alex, for the core algorithm code. He will probably implement the Patience diff alone. All that this function needs to do is properly diff two objects and then put out the diff information.

3.4 Alex Clemmer

is also primarily responsible for the core diff algorithm, and will probably implement the O(ND) diffing algorithm and most of the structural work that the diffing functions require (*e.g.*, the structs required to hold the deltas. He is also the most experienced with git internals, and will be coordinating a lot of the effort by telling people how things work, where things go, and where to begin looking for solutions. He is also the strongest point of contact with the community.

4 Software Processes

4.1 Version control

We are modding `libgit2`. We are of course using git as a version control system. Since this is OSS, we are using github as our repository, which is really convenient because `libgit2` also uses github.

4.2 Pair Programming

Not something we normally do, pair programming will be incredibly useful here, because we will be operating in a really complicated and dense codebase. We've already done some of this, and we can tell you that it makes us a *lot* more effective.

4.3 Code Review

This is incredibly useful. It makes our code noticeably better, but also we are contributing to OSS. There is no way to *not* do review, because review is required before pull.

5 Appendix A: Use Cases

1:

Developer uses libgit2's diffing mechanism Does not provide all necessary arguments to diff function Return a GIT_ERROR code (which are defined within libgit2)

2:

Developer uses libgit2's diffing mechanism Provides a file that contains a function that was previously in a different file and is diff'ing against the current delta-head of content diff determines which file(s) Returns a data representing each affected file in normal diff format

3:

User uses the binding that calls the 'git merge' function with an invalid git repository passed into the function, and NULL is returned.

4:

User uses the binding that calls the 'git merge' function with an valid git repository and the two branches to merge passed into it (with no merge clonfilcts on those branches) and those branches merged and the files on the local filesystem are changed to the result of this merge.

5:

User uses the binding that calls the 'git merge' function with an valid git repository and the two branches to merge passed into it (with merge clonfilcts on those branches). The branches are not merged and the files on the local filesystem are changed to include both the code from both branches where the conflict occurred.

6:

"Merge" function calls diff to get an internal delta for merging purposes – the diff is returned

7:

User calls "diff" explicitly on two files – the raw diff is returned

8:

Diff is called and git repository is not initialized – return GIT_ERROR.