TELER: A Streamlined Version Control System

Project Report

Hadley Luker Zachary J. Susag

May 15, 2018

1 Project Overview

During their education, most computer science students must learn to use a version control system (VCS). Although these systems see wide use in research and industry for software development and production, in educational contexts they typically service a narrower use case: saving, submitting, and tracking changes in programming assignments. This limited use case causes many students to become overwhelmed when confronted with the numerous and powerful commands available to modern VCSes such as Git and Mercurial. For example, the typical workflow of saving changes to a remote repository in Git proceeds as follows:

1. `git add .`
2. `git commit -m "Fixed bug."`
3. `git push`

These three commands are practically an idiom that students are habitually trained into, and deviations from this pattern are rare and incite confusion. For example, if the `-m` is omitted from the end of `git commit`, the terminal will run the Vim text editor by default in order for the user to write a commit message. Due to Vim's unintuitive keybindings and minimal user interface, students confronted with this sudden change in environment may make errors in the editor or find themselves unable to exit and subsequently close the entire hosting terminal window.

For these reasons, we decided to develop TELER: a streamlined version control system for the student use case. Our system exposes exactly five commands: repository initialization, pushing changes with a required commit-like summary to a remote, pulling changes from a remote, reverting changes to a specific version, and viewing version history. This interface condenses the common add-commit-push / pull command sequence at the expense of requiring all files to be pushed or pulled at once. However, this has positive effects: it encourages users to push frequently and document important changes as they are made.

Evaluation summary.

2 Design & Implementation

The design of our program is centered around two main commands and three components: the command line interface, the shadow directory which contains all changes recorded by TELER, and the auxiliary data structures and compression that enable TELER to be more time and space efficient.

2.1 Design

The use of textscteler depends mostly on two key commands, `push` and `pull`.

2.1.1 `teler push`

At the heart of any good VCS is the ability to rapidly save checkpoints into the repository with a timestamp and a message without disrupting the user's workflow. To do this, the VCS must find which files have changed, which are new, and which are deleted, save these changes into the shadow directory, and bundle these in a commit to allow for the commit to be referred to at a later time.

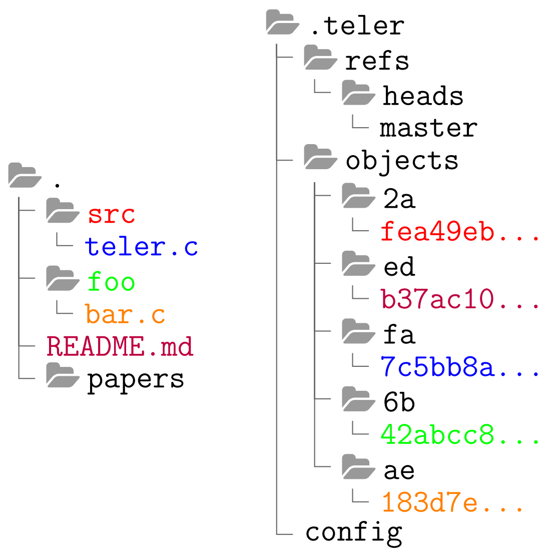
The `teler push` algorithm does precisely this. First, `teler push` reads the latest commit from the `.teler` directory and retrieves from the associated commit object the root of the object tree. From there, our algorithm recursively traverses down this object tree adding the metadata found within the tree objects to a hash table. This hash table will be used to not only store the metadata of files from the previous commit but also any new, or changed, files in the pending commit.

Once the hash table has been fully populated `teler push` will begin computing which files have been changed by traversing the current working directory recursively. Each file within the working directory is added to the directory tree by storing the hash of the file within it. There are two cases.

If the file is a regular file then the file is hashed using `openssl`'s implementation of the SHA-1 hashing algorithm. The hash is then looked up in the hash table to determine whether the file exists already within the shadow directory. If not, then compress the file to the appropriate location in the shadow directory pointed to by the hash using the `zlib` compression algorithm. Once saved, place the hash, file permissions, and human-readable filename (metadata) into the hash table and the hash into the directory tree. If the hash was found, then just add the hash to the directory tree as there is no need to recompress the file as an identical copy already exists (unless there is a hash collision). This is crucial to minimize space consumption as more often than not files do not change with every commit so storing multiple copies of the same file within the shadow directory would result in an egregious waste of space.

If the file is a directory then make a new interior node within the directory tree and recursively add the files within the directory as children to this new node. Once all children have been added, hash them and their metadatas to compute the hash of the tree. Place this information in the hash table.

Once the directory tree has been constructed, `teler push` will write a commit object containing the hash of the root of the directory tree, the previous commit hash (if one exists), the author's name and email, a timestamp, and a message which was given to `teler push` by a prompt to the user. Once committed, the directory tree is traversed whereby all of the tree objects are written to the `.teler/objects` directory. This is simply a matter of for each interior node of the directory tree, iterate through each child, looking up their metadata in the hash table, and writing this information to the tree object. Once written, the tree file is compressed using `zlib`.



2.1.2 `teler pull`

The `teler pull` algorithm is effectively the inverse of the push algorithm in that instead of constructing the directory tree from the working directory the tree is constructed from the shadow directory. The latest commit object is retrieved in a symmetric fashion to `teler push`. The directory tree then is begun to be constructed by adding the root of the object tree as the root of the directory tree.

Each object's hash and metadata inside the tree object is inserted into the hash table. If the current object is another tree object, create a new interior node within the directory tree and recursively add the node's children by parsing the associated tree object. If instead the current object is a blob, then nothing special needs to be done and the blob's hash is simply added to the directory tree.

Before the working directory is populated with the contents of the directory tree, it is crucial that the working directory is cleared of all residual files. If not done then files from the current state will be mixed with those of the latest commit.

Once the directory tree has been fully built, it is then recursively traversed. For each interior node its hash is retrieved which is then used to collect its metadata from the hash table. Using the human-readable filename which is stored in said metadata a new directory with the filename is created. From there, the traversal recurses downward and processes each of the interior node's children. If one of the children is a blob, then decompress the blob from the shadow directory into a file whose name is the human-readable name stored in the hash table.

2.2 Implementation

2.2.1 Shadow Directory

Within every TELER repository is a shadow directory appropriately named `.teler`. The structure of the `.teler` directory is largely influenced by Git.

Within this directory are two subdirectories: `.teler/refs/heads` and `.teler/objects`. The `.teler/refs/heads` directory stores a file for each branch (currently only master) which inside holds the hash of the latest commit. This corresponds to a file within the `.teler/objects` directory.

The `.teler/objects` directory stores the entire history of the repository through snapshots. Directly within this directory are numerous subdirectories whose names are two hexadecimal digits. Within these subdirectories are files made up of thirty-eight hexadecimal digits. Put together, a forty digit hexadecimal number is created which directly corresponds to a SHA-1 hash to uniquely identify it. These files are what are called objects. Objects are broken down into three different types: *trees*, *blobs*, and *commits*.

Trees are the mirror of directories in the working directory in that they store two different types of objects: trees and blobs. Any object within a tree is said to be a *child* of that tree object and so in `teler pull` will be turned into a file within the directory associated with the tree file. These children are referenced by their hash, which corresponds to a file within the `.teler/objects` directory as well as the child's permissions and human-readable name. This information is all compressed via `zlib`.

As trees are to directories, blobs are to regular files. Each blob is a compressed version of the contents of a regular file. The name of a blob is simply the SHA-1 hash of the contents of the file. Blobs are what separates a snapshot-driven implementation of a VCS from a delta-driven implementation as instead of storing deltas, or changelogs, to reconstruct the files we instead store the entire file (albeit in a compressed form).

We opted for a snapshot-driven implementation for `teler` due to its slightly better performance in restoring a working directory to the state it was at a specified commit (i.e. `teler pull`). A downside is we consume more space in the process. As such, minimizing space consumption was one of the driving goals of the design of the shadow directory. We accomplish this through two techniques: compression and uniqueness.

Every file that is stored within the `.teler/objects` directory is compressed using the `zlib` deflation algorithm, aside from those files which are stored within `.teler/refs/heads`.

The other technique is that each blob is only stored once within the shadow directory. Since a blob is identified by the SHA-1 hash of its contents we can uniquely identify copies of files as they will have the same hash. This property is preserved in `teler push` (2.1.1).

It must be noted that we are relying on that we will not have any hash collisions. If a collision happens either a file would be wrongly identified as unchanged and/or portions of a commit would be overwritten. However, since SHA-1 can produce 2^{160} unique hashes the chance of collisions are miniscule, especially in an environment which is likely to be relatively small, such as a repository.

2.2.2 Command-Line Interface

The command-line interface is a simple argument parser made using GNU C's `argp` interface. This interface automatically generates help and syntax information for display on the command line through the assignment of certain global variables and "option vectors", a structure which organizes and configures command line arguments. However, our implementation is slightly modified from the GNU standard of C argument morphology. The GNU practice is to precede all arguments on the command line with a number of hyphens: one for single-character arguments (e.g. `-v`) and two for long arguments (e.g. `--verbose`). We chose to let the first argument be hyphen-less. (For differentation's sake, we refer to this "first argument" as the "command".) This reflects more closely the practices used by Git for their command line interface, which we suspect will be more familiar to students. This decision caused us to parse the first argument through an if–else if chain and instead use `argp`'s parser function to parse arguments given to the command itself.

2.2.3 Data Structures

In order for TELER to be efficient it must be necessary for new or changed files to be rapidly identified as such, as well as the hierarchical structure of the working directory preserved.

2.2.4 Hash Table

A dynamic hash table whose keys are the SHA-1 hashes and values are the associated permissions and human-readable names is used to allow for rapid lookup of metadata information. Due to indexing, lookup can be done in $O(1)$ time. Aside from storing the metadata information for files the hash table is crucial for rapidly identifying whether a file has ben changed.

As described in section 2.1.1, before compressing blobs into the shadow directory `teler push` checks to see whether an object with an identical hash exist, thus coreesponding to a file with exactly the same contents. With the hash table this check can be done in $O(1)$ time instead of having to check to see if a file with the hash exists in the `.teler/objects` directory which would require I/O, and thus incur a performance hit.

2.2.5 Directory Tree

The other crucial aspect of TELER is being able to preserve the hierarchical directory structure of the repository after executing `teler pull`. Since directories are inherently trees we chose to represent both the structure of a commit and the working directory as a general tree in which each node has an arbitrary number of children. Interior nodes correspond to directories, or, equivalently, tree objects while leaves correspond to regular files or blobs. Within each node the hash of the file is stored which is used to retrieve the metadata information of the file from the hash map.

The directory tree is used in both `teler push` (2.1.1) and `teler pull` (2.1.2) for symmetric purposes. During pushing, the directory tree is a representation of the working directory in memory whereas during pulling the directory tree represents the object tree of a commit. Thus, the directory tree is used to both write the commit to the shadow directory while preserving the structure of the directory and to restore the working directory from a specified commit. Both of these operations are simply recursive tree traversals.

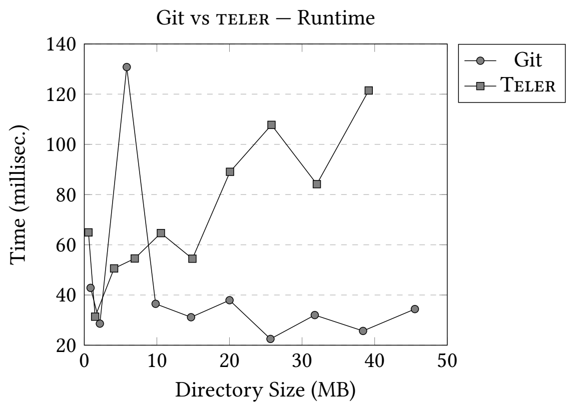
2.2.6 Compression

As TELER saves every version of a file that was added in a change, it becomes important to utilize every means of space-reduction we have available to us. Therefore, all files and commits within a TELER repository are compressed. We utilize the `zlib` library to accomplish this. We chose `zlib` because it is free, portable, will essentially never expand the data, and whose memory footprint is independent on the input data. Specifically, we use the deflate algorithm provided by `zlib` in a manner similar to that of `zpipe`.

3 Evaluation

To evaluate TELER we compared its speed in generating commits through TELER PUSH with Git version 2.17.0. For hardware we used an Intel Core i5-3320M at 3.3GHz with 8GB of RAM with a 320GB spinning hard disk. The workstation we used to evaluate TELER was running Arch Linux with kernal version x86_64 Linux 4.16.8-1-ARCH.We collected the running times of Git and TELER using the `time` command.

To fairly assess each program we generated a random directory with a maximum depth per directory of 3, up to 4 first-level directories, and up to 6 maximum children per directory. We then recorded the time it took for each program to make a commit of the entire directory. This was done ten times with a new subdirectory of the same specifications being made, increasing the size of the total directory each time. During this process we also recorded the total size of the directory in bytes to be able to compare performance based upon size of directory.


Git vs TELER — Runtime
Time (millisec.)
Directory Size (MB)
Git
TELER