Snapstore: A Version Control System for Everyone

by

Alex Chumbley

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2016

Author . . . .
Department of Electrical Engineering and Computer Science
August 16, 2016

Certified by . . . .
Daniel Jackson
Professor
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Snapstore: A Version Control System for Everyone

by

Alex Chumbley

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

Version control systems have, for many years, been applications that are developed and maintained with software engineering in mind. However, other less technical industries and endeavors can benefit immensely from the functionality these systems provide. Existing systems such as Git and SVN have too steep a learning curve to make quick adoption feasible. Less complex file syncing applications such as Dropbox and Google Drive do not offer the same level of power as their software-minded counterparts. Even novice programmers are left using less than ideal systems to avoid sinking time into learning unnecessarily complex systems. In this thesis, we provide two main contributions, both in the context of Snapstore, the proposed system. We outline the steps needed to design a simpler, more powerful version control system by following the theory of conceptual design. We then describe the proprietary technologies and algorithms we created to fulfill the vision of a universal, simple version control system. Snapstore is the result of all the goals and ideals described by this paper.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

Thank you to Daniel Jackson and Santiago Perez de Rosso for guiding me through this process.

Thank you to my family, for you love and your support.

Contents

Chapter 1

Introduction

In the winter spanning the years of 2014 and 2015, I worked as a teaching assistant for an introductory course in Python programming. The course was short, only four weeks long. I met with multiple groups of students as they worked their way towards their final project. Immediately the issue of collaboration came up. How can they divide work in a way that makes sense? How would they share code? Can two team members work on the same file at the same time?

The main question, of course, was how should they share their code? In such as short course, learing about Git was out of the question. We were more worried about teaching students what a function was than showing them the finer points of branches and merging. Dropbox was an easy alternative, but after a half-dozen hours of trying to set up a useful, collaborative folder, they realized that the complexities of sharing in Dropbox were not worth the trouble. Dropbox would not let them efficiently share portions of their codebase, disallowing things like nested shared folders. The timeline of this project was very short, and parallel development with Dropbox was very difficult with everyone writing code at once.

In the end, they decided to use email to share code. This decision was made in order to reduce general administrative overhead and the possibilities of conflicts. But it came at the cost of speed and efficiency.

Version control shouldn't be confined to a small subset of power users in the software industry, as is the case with Git. File sharing shouldn't be obscured with confusing design concepts, as is the case with Dropbox's shared folder model. Users from any discipline should be able to start an application and intuitively share files and utilize version control in minutes. That is the vision of this paper.

1.1 Version Control Systems

Version control systems (VCSs) are ubiquitous today, though they come in different forms. Software developers have been using systems like SVN and Git for years for version control. However, there are also file sharing systems such as Dropbox and Google Docs that perform simpler tasks and are geared towards less technical users. While version control systems and file sharing systems are very different, they do fulfill some of the same needs: data storage and data sharing. The version control system, then, is a file sharing system that contains even more power and functionality.

Due to their overlapping nature, this paper compares the two sets of systems, finding benefits and flaws within each set. Each system tends to have their faults, and those faults come in two major categories.

The first issue these systems have is they are too narrow in their domain. Rarely are software developers using Dropbox to collaborate on software. It is even rarer that a non-technical industry, such as law, would use a system like Git for sharing files. In both cases, the system was designed for a specific user group instead of underlying purposes. Technical systems are built for technical people, with a steep learning curve only conquerable by power users. More basic systems are too simple, lacking the functionality to support the requirements of more complex projects.

The second main issue these systems have is their design. Even Git, perhaps the most popular, prototypical VCS suffers from a lack of robustness in its design. Users are often frustrated by Git's complicated and opaque design. They donâĂŹt understand so many of its functions that they often resign themselves to using a few basic commands. Novices, especially, find some of Git's design choices confusing such as its inability to allow a committing of an empty directory [2]. Even supposedly simpler systems like Dropbox have similar conceptual issues. Dropbox's shared folder model has left many users confused [6].

We believe that both of these categories of issues can be solved by focusing on the design of a VCS at the conceptual level. A new system is proposed, one that leverages the best from the technical and non-technical systems available today. It is robustly designed using essential purposes and concepts in order to make the user learnability as fast and as easy as possible. It promotes the idea of "opt-in complexity". This ensures that basic users can effectively use the system on day 1 while advanced users can learn the system more in-depth to unlock its full set of features. We hope this system can bridge the gap between technical and non-technical industries.

1.2 Conceptual Design

In the field of software design, there is little agreement concerning how a designer should structure the software they build. Notions of conceptual integrity and concept-based design are nothing new. Leaders in the field of user interface have noted the important of the connection between the mental model that a user has of a piece of software with its underlying software-based concepts.

Conceptual design is a design theory that reduces all of these past, disjointed theories. It calls for a conceptual model, designed to fulfill a set of purposes [4]. Within the conceptual model is a set of concepts. These concepts represent essential ideas that a system deals with, and their creation and refinement are the central activity of software design.

The designer should be designing the system with these concepts as their vocabulary. Then, the user can use the system with this conceptual model as their mental model of the system. This shared model connects the designer with the user, making it easier to understand. Any system built needs an unspoken medium of communication between the designer and the user. Conceptual design gives that medium a singular language.

A given concept is accompanied by a motivating purpose, its reason for existing. A purpose is a desired result. It is not piece of code, a design detail, or a way to achieve a desired result. The purpose behind the trash can on your computer's operating system is to be able to undo file deletions.

There are four properties that concepts must have in order to be strong and viable. Concepts should have **motivation**, meaning they fulfill an articulated purpose. No two concepts should be **redundant**, or fulfill the same purpose. Concepts should not be **overloaded** and fullfill more than one purpose. Finally, concepts should be **uniform** and, when possible, variant concepts should behave similarly.

In [2], 6 purposes associated with VCSs are enumerated.

1. Making a set of changes to a file persistent
2. Represent and record coherent points in history
3. Group logically related changes together
4. Synchronize changes of collaborators
5. Support parallel lines of work
6. Do work in disconnected mode

Conceptual design was used in this paper to help guide the design of Snapstore from the beginning. We believe that its use will make for an easier mental model of the system for the user. One of the goals of this system, after all, is to be accessible to non-technical users. Conceptual design simplifies this task by pruning unnecessary, complicating concepts and creating simple, purpose-driven ones.

Chapter 2

An Overview of Snapstore

Snapstore was created with two distinct groups of users in mind, non-technical users and technical users. In order to be an attractive system to technical users who use systems like Git, Snapstore needed to support the functionality of powerful version control systems. However, Snapstore also needed to have a smoother learning curve to promote quick startup and attract users who feel overwhelmed by complicated VCSs.

We decided to create Snapstore with an opt-in strategy concerning complexity. Users can download Snapstore and get started right away with simple actions like file storage. Then, if desired, users can explore more advanced features of the system, without fear that they are necessary for their current project needs.

2.1 Basic Features

The basic features of Snapstore allow a single user to use the application for file storage and backup.

Snapstore uses a simple user-based identification system. Once a user downloads Snapstore, they create an account with a username. With this, they can immediately begin to use Snapstore to whatever capacity they wish.

Snapstore operates within a specially designated Snapstore folder which is created upon opening the application for the first time. It operates very similarly to the Dropbox folder; Snapstore only looks at files that are inside of it and nothing else. Snapstore watches the user's filesystem for changes in order to trigger certain functionality. This allows users to use any editor with Snapstore. The Snapstore folder that Snapstore uses does not need to remain the default folder. Snapstore can be opened using any folder as the Snapstore folder, allowing users to have more than one folder that is to be monitored by Snapstore.

2.1.1 Snapshots

Once the user is logged in and starts making edits to files in their Snapstore folder, Snapstore will begin saving their data. By default, any edits to a file made in the Snapstore folder will result in the creation of a *snapshot*. The snapshot is essentially a copy of a file at a specific moment in time. Snapshots are also created when a file is created, deleted or renamed. This allows users to see the true history of a file, even if it has been moved, renamed, deleted, or reverted.

Snapshots can be one of six types: create, update, rename, delete, merge, and conflict. The different types of snapshots will show up as different colors in the snapshot graph so that the user can see the difference.

When creating snapshots for a given file, Snapstore will add the snapshot to that file's *snapshot graph*. This graph represents a history of the file and it shows each snapshot that was taken, along with its relationship to other snapshots of that file. A snapshot has one or more parents (the snapshot(s) taken before it), and it has a child (the snapshot taken after it). The first snapshot in a graph is called the *root*, and the last snapshot in the graph (i.e. the current snapshot) is called the *head*.

2.1.2 Upstreams

When connected to a network, all changes and edits are sent to an *upstream* automatically. An upstream is a remote repository that allows Snapstore data to be stored persistently, in case of local system crashes. The default upstream is Snapstore's server, and it is available right away, when the application is opened for the first time.

Snapstore, at its most basic, can be used for personal backup. Once Snapstore is downloaded and connected to a network, all snapshots are sent to an upstream for persistent storage. Any user can use this setup for personal backup. They can even decide where to send these snapshots (described below in section 2.2.4); perhaps a machine in their home is all they need.

One benefit of the upstream model that any Git user can sympathize with is how it handles the case of a single user using multiple machines. A Git user needs to pull and push from every machine they use in order to get work they have commited from other machines. Snapstore eliminates that by having all snapshots automatically saved on whatever server the user wants. Then, when they log in to another machine, all changes are automtically pulled in from the server and reflected on the user's machine.

2.1.3 Local Repository

All changes made on a user's local computer are saved in the *local repository*. This local repository is a collection of branches and their associated data. A user can have multiple local repositories on one or more machines to shield certain edits from other repositories. The local repository allows users to work in a disconnected setting. While offline, Snapstore will still watch the filesystem and save changes. When connection is restored, Snapstore will push all new changes to the server.

2.2 Advanced Features

Snapstore's advanced features allow users to access the more powerful components of a version control system. None of the advanced features are needed to use the system. Rather, they provide additional functionality that users might want when working on their projects.

2.2.1 Groups

Snapshots can also be grouped for projects and files where this is necessary. Groups of snapshots do not necessarily need to contain all snapshots from the same file. For example, a user might want to group the head snapshots for every file in a branch. Or, they might collect multiple snapshots from the same snapshot graph where they fixed a bug.

2.2.2 Tags

Groups of snapshots can be given tags. These tags represent a coherent development point. For example, the group of head snapshots for a branch might be tagged "Version 1.3" to signify that the project is in a stable release state. Only groups that have at most one snapshot per file (vertical groups) can be tagged. Tags help administrative duties such as setting deadlines, labeling work, and marking milestones.

2.2.3 Branches

Snapstore uses the *branch* to differentiate separate lines of development. Branches hold files (represented by their snapshot graphs) and their group and tag data. Branches can be shared between users so that they can collaborate on files within them. When Snapstore is opened for the first time, a master branch is created.

New branches can be created for multiple reasons. One reason might be to simplify the Snapstore folder for an individual's use. Keeping every single file in the master branch would become excessive, and new branches can clean up a user's workspace. Another reason might be for a user to hide certain files from others users they are sharing a branch with. A user might keep private files in their master branch and instead create a new branch called "public" that is shared with other people. A user can also create and use branches to maintain multiple versions/releases of the same project, keep the development of major features isolated, and to give users the ability to try out experimental changes without affecting the main line [2].

A branch can also be created by cloning an existing branch. Creating a clone involves choosing a branch to clone, and then selecting the files inside the original branch to bring over to the clone. This allows the new branch to have common snapshot ancestors with the original branch, for the purpose of later merging. Branches that are cloned will inherit all of the related tags and groups from its parent branch. This means that for all snapshots that are cloned into the new branch, any groups associated those snapshots, and any tags associated with those groups, will be in the cloned branch.

Collaborating on Branches

When a branch is shared with another user, that branch's data is sent to that user and it becomes a shared branch. When the new user switches to that branch, it changes their file system within their Snapstore folder to match the branch's head snapshots. On a shared branch, when any user makes changes, those changes are immediately sent to the other user(s) associated with that shared branch, offering immediate collaboration.

When conflicts arise due to multiple users working on the same branch at the same time, Snapstore uses a last-write wins methodology. The last snapshot to reach the server will become the head snapshot for the file. However, no snapshots are lost in the conflict, and the user can revert to a passed over snapshot easily.

An important use case is the one mentioned in the introduction of this paper. If a small software team in school needs to be able to share code and develop in parallel, they can work on a shared branched in Snapstore. Snapstore is especially attractive if they do not have the time or desire to learn a system as complex as Git. In either case, Snapstore allows them to share work and maintain versioning.

Snapstore is designed with every industry in mind, and these features also benefit the non-technical user. A legal team working on an array documents might use Snapstore to maintain versions of those documents and to manage parallel work on the same document. Team leaders can easily keep track of progress, and those working on the documents can see the snapshots from other team members immediately.

Merging two branches is possible with Snapstore and facilitates parallel development in teams. Merging two branches compares the snapshot graphs in those two branches. If two snapshot graphs correspond to the same file, then a merge is performed using their common ancestor. This will result in a snapshot with as many parents as there are snapshots being merged. This new, “merge” snapshot will merge the file contents from the head snapshots to produce a new snapshot. If there is a merge conflict, then the resulting snapshot will be a conflict snapshot. Much like a merge conflict in Git, the file will show where the conflict needs to be resolved. Unlike Git, however, this “merge” snapshot is already on the server and saved, no conflict resolution is needed to continue working. By simply fixing the conflict and saving, a new snapshot is created that reflects the fix. Merging two branches will keep all of the group and tag data from both branches.

With these more advanced collaboration tools, Snapstore offers many project management solutions. By adding branches and clones of branches, a project manager can shield certain parts of the project from certain workers. For example, imagine that a web application project has a project manager, a front-end developer, and a graphic artist. The project manager can create two clones of his master branch from the code folder and the images folder of the project. She can then share these cloned branches with the developer and the artist. This shields each worker from the other’s work. The project manager can keep an eye on each branch’s progress because she has access to both cloned branches; each snapshot one of her employees makes shows up on her machine. Then, the project manager can merge both branches back onto the main branch and have a completed project.

Snapstore branches can also be used for projects with different workflows. The Linux project uses trusted lieutenants to review patches in sections of the Linux Kernel before sending them on to the project owner, Linus. Many open source developers send their code directly to these lieutenants for review. This hierarchy allows a huge project like Linux to function efficiently. Snapstore can achieve a similar hierarchy by having the project owner clone certain aspects of the full project and share those clones with the lieutenants. Then, they can share these clones with the world and vet incoming snapshots.

2.2.4 Changing Upstreams

The location of which upstream repository a local repository is connected to can be changed according to a user's needs. A user can designate another machine as their backup by following a few easy steps.

1. Download the Snapstore server program onto the machine
2. Run the Snapstore server on that machine
3. Point the Snapstore client to that machine

This allows Snapstore to be a viable use case for sensitive information and even for data backup in the home.

Chapter 3

Design

The design of Snapstore had two steps. The first was to identify the purposes of a version control system. The six purposes of a version control system identified in [2] were used.

The next step was to create a conceptual model composed of concepts that fulfilled the purposes of a version control system. These concepts were invented in a way that is aligned with conceptual design theory [4]. That is, they were each had a motivating purposes, they were not redundant or overloaded, and they were uniform. A mapping of each Snapstore concept to its motivating purpose is shown in Table 3.1. The $\mapsto$ symbol stands for "is a subpurpose of". A graphical representation of the conceptual model, using the notation for extended entity-relationship as used in [4], is shown in Figure 3-1.

3.1 Data Storage - Snapshot

Persistent data storage in Snapstore is achieved with the notion of a *snapshot*. The snapshot is a saved state of a file. Snapshots record updates, renames, moves, deletes, along with merges and conflicts. Records of the many operations enable all VCS features of Snapstore. The *head* snapshot for any given file is the most recent snapshot made for that file and the current content of that file.

Concept	**Motivating Purpose**
Snapshot	Making a set of changes to a file persistent
Create, Update, Rename, Delete, Merge, and Conflict Snapshot	Track various types of changes to a file $\mapsto$ making a set of changes to a file persistent
Snapstore Folder	Create, read, and update files $\mapsto$ making a set of changes to a file persistent
Tracked File	Mark files whose changes should be saved $\mapsto$ making a set of changes to a file persistent
Untracked File	Mark files whose changes should be ignored $\mapsto$ making a set of changes to a file persistent
Group	Grouping logically related changes together
Tag	Represent and record coherent points in history
Upstream Repository	Synchronize changes of collaborators
Branch	Support parallel lines of work
Local Repository	Do work in disconnected mode

Table 3.1: VCS purpose to concept mapping for Snapstore

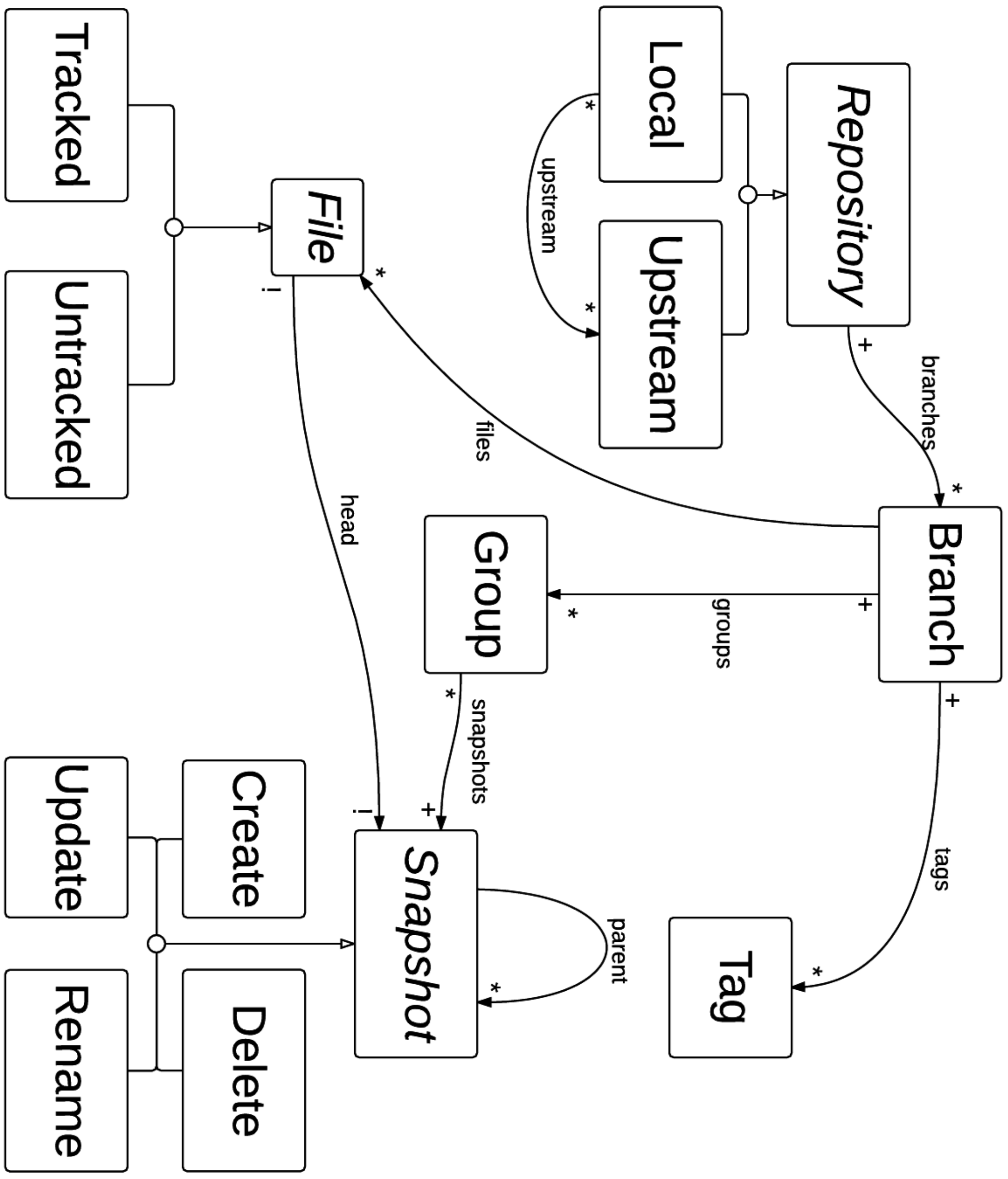

Figure 3-1: Concept Model of Snapstore.

A snapshot is either a create, an update, a rename, a delete, a merge, or a conflict snapshot. This distinction dictates the values of the snapshot fields. Create snapshots have no parent. Update snapshots have a parent, a child, and content. Rename snapshots have a different filename than their parent. Delete snapshots have no content. Merge snapshots have more than one parent, and conflict snapshots are merge snapshots that have conflict markers in their data.

All snapshots have parent snapshots and child snapshots. The snapshots of a file are related by the graph they create with these parent/child relationships. This ordering forms the snapshot graph described in the tutorial section. The snapshot graph is guaranteed to be an in-order description of snapshots a specific client has made to a file in a branch. Each unique (branch, file) tuple is represented by its own snapshot graph. A snapshot can have multiple parents if it is the result of a merge operation.

Any file, identified by its snapshot graph, can either be tracked or untracked. Untracked files will not create snapshots and will not affect the local or upstream repositories.

3.2 Grouping Changes - Group

A *group* is an assembly of related snapshots. A group must contain at least one snapshot, but there are no restrictions on what kinds of snapshots can be in the group or what their relationship must be. The same snapshot can exist in more than one group. It is up to the user to decide what makes a group of snapshots logically related. This allows flexibility in projects and development strategy.

Groups are an attribute of a specific branch. Even if a group contains the same snapshots across different branches, they are treated as different concepts because they exist on independent lines of development. They can be given names for identification for the user.

3.3 Recording Coherent Points - Tag

The notion of a *tag* allows users to label logical milestones in their work. They describe groups but have an added function over a group's name: they describe the status of the group as representing a coherent point. Here, coherent means that the project is in a state that is ready for further development or work, though the definition will differ from team to team[2].

Tags will always describe groups that are perfectly vertical. That is, at most one snapshot from any file is in the group. An example of this is tagging every head snapshot in a branch at a given time with the tag "Submitted to Scientific Journal" or "Version 1.0".

Tags are also an attribute of the branch. This means that they must be created inside a of an independent line of development. They can be copied across branches when merging and cloning, but they stay a fundamental attribute of the branch concept.

3.4 Support Parallel Lines - Branch

In Snapstore, the concept of a *branch* supports parallel and independent lines of development. These branches are completely separate from each other. They facilitate the appropriate partitioning of data.

The branch houses three of the other main concepts in Snapstore: snapshots, groups, and tags. All three of those concepts exist within the confines of a branch. All of these things together consitute a line of development, and so the branch is the conceptual representation of that line.

Branches make up a repository, whether that repository is local or upstream. Switching between them constitutes changing the line of development, project, folder, or anything that delineates the user's work. Switching between two branches on a single local repository that are stored on different upstreams has no averse effects due to their independence.

Branches can be merged together, synchronizing the parallel development. This simply involves combining each branch's individual snapshot, group, and tag data together as explained in chapter 2.2.3.

3.5 Synchronize Changes of Collaborators - Upstream Repository

Snapstore uses a centralized data storage system that holds all connected branch data, called an *upstream repository*, or upstream for short. While users do not necessarily have to use an upstream for their local repository, it is the only way to collaborate with other users on any branch in that local repository.

Every local repository can have multiple upstreams. This allows the user to distribute where their snapshots, groups, and tags are saved and backed up. These upstreams can then be shared with other Snapstore users, provided that the upstream is connected to those users.

All changes that occur at the branch level (branches, snapshots, groups, tags) are reflected in any connected upstream. There, the upstream can see if any other users have access to that branch and it can push the changes down to them.

3.6 Disconnected Work - Local Repository

The ability to leverage the benefits of a version control system without needing an internet or network connection is handled by the *local repository*. The local repository affords all of the same relationships to other concepts as the upstream. That is, it is a collection of branches, which are in turn collections of snapshots, groups, and tags.

The local repository has one other important relationship, it can have zero or more upstreams attached to it. These upstreams are mirroring the data housed by the local repository.

3.6.1 Discussion

During the design process, there were many decisions made that had lasting tradeoffs for Snapstore. The main tradeoffs are explored below.

Granularity of a Snapshot

The decision of what a snapshot would represent was the first design decision we encountered. Either a snapshot could represent a file, or it could represent every file in a branch. In many VCSs such as Git and Mercurial, saving changes couples together the act of saving changes with the act of grouping changes, resulting in an overloaded concept [4]. We separated the saving of changes with grouping those changes, so the snapshot represents a single file.

Another reason the snapshot describes only a single file was that it was more intuitive to a typical user. If a user was to save a file and create a snapshot, they would expect that snapshot to relate to the object they just interacted with, that file. They would not expect it to relate to every file in the branch.

One downside of this approach is the additional computation needed to compute the current state of a branch. In Git, the current state is the head commit. In Snapstore, we need to calculate this by grabbing all of the head snapshots for a branch. This is typically trivial, so the tradeoff is beneficial.

The Upstream

Different version control systems and file sharing systems have their own ways of handling storage. Git, for example, use a decentralized storage system. Dropbox and SVN, on the other hand, use a centralized system. When deciding which model to use for our upstream, we looked at both models' pros and cons.

Snapstore uses a hybrid centralized/decentralized upstream model. On the surface, it is centralzied because all collaboration must take place via the upstream repository. That is, any data a user wants to collaborate on with another user must first go through a centralized repository.

But, Snapstore is also decentralized because users have a local repository, where actions can be made without requiring an active connection to the upstream. Snapstore users each have an entire history of their branch on their own system, just like in a decentralized version control system. Because of this, users can work offline, without checking out a central repository.

There are downsides to this hybrid model. Initially, there is only one upstream server, and so the location of the data is held by a single entity. If users want to set up their own server, they must use a machine to do so, and there is the overhead of setting up that server. Because there is no push/pull model in Snapstore, this machine must always be online in order to facilitate collaboration. Further, users cannot share directly with subsets of users on a given branch. They can, however, work around this limitation by creating new branches with new sets of users.

We used our goal of opt-in complexity to create a balance between the simplicity of a client/server, centralized model with some of the power of a decentralized system. The centralized model is easier for the majority of users to use, and it has a faster learning curve [1]. In Snapstore, the data on the upstream is indeed "blessed", and so it is centralized. However, by offering a local repository alongisde and potentially independent from the upstream, Snapstore has some characteristics of a decentralized system.

File Names

Whether or not to make the file name a property of a file or the identifier for a file was an important decision for Snapstore. Systems like Git use the file name as an identifier for a file. Because of this decision, renames to a file are sometimes processed as deleting that file and creating a new file, causing much consernation among users (especially novices)[2].

We wanted to be able to fully support renaming files in Snapstore, so file names in Snapstore are simply a property of the file. Each snapshot in a given snapshot graph will have the same file id. When branch merging occurs, only snapshot graphs with the same file id will be merged. The merge operation will look back over a file's two snapshot graphs for a common ancestor and perform a standard three-way merge. This allows Snapstore to accurately handle renames and merges.

Chapter 4

Implementation

This chapter details the implementation of Snapstore. That includes relevant implementation decisions, the current status of Snapstore, and important algorithms.

Snapstore was built to be cross-platform using Electron[1]. To offer consistency, we used web sockets[2]. Once a client is connected over a socket, Snapstore can be constantly pulling in and pushing out new snapshots that come in from that user and from other users. Sockets provide the additional benefit of allowing us to group users. We used this functonality to make “rooms” of users that have read and write access to a specific branch. Pushing changes on that branch out to those users is simple with sockets.

4.1 Data Structures

4.1.1 Client

Each local repository on the client has its own mongo database. Each database has a collection of snapshots, branches, groups, tags, and events. It also has one snapstore document and a binary large object (blob) collection.

The snapshot collection holds every snapshot in the local repository. Each snapshot knows the branch it belongs to, its parent and its child, the hash of its content

[1]http://electron.atom.io/

[2]www.socket.io

(which is stored in the blob database), and the name and id of the file associated to it.

When a file is saved, the resulting snapshot must first decide what kind of snapshot it is. Whether it is a create, update, rename, delete, merge, or conflict snapshot dictates how it will populate its data fields. A create snapshot, for example, has no parent snapshot. When the snapshot is created, it is added to the collection of snapshots, and the branch updates its list of head snapshots to include this new snapshot, clobbering its parent.

If two snapshots have the same content, they point to the same blob data in the blob collection to save space. This blob collection holds hashes of the content along with the actual binary content. Further, duplicate snapshots can be flagged when connecting to the server by looking at their ids.

The branch data structure has a name, a list of its head snapshots, and a list of groups and of tags in it. When a new branch is created, it is added to the local repository with only a name. If it was cloned from another branch, the cloned branch will point to all head snapshots, groups, and tags from the original branch. Subsequent actions such as snapshot creation populate the branch. When switching to a branch, the heads snapshots for the target branch are read and applied to the Snapstore folder. To show the history of a file in the UI, the head snapshot of the file is first found; the rest of the history is found by searching backwards through the snapshot graph.

The group data structure only references a set of one or more snapshots and has a name. The tag data structure, in turn, references a group and has a name.

The event collection stores all of the unconfirmed snapshots, groups, and tags on the client.

The snapstore metadata structure holds all necessary metadata for the application. This includes the user's username along with their password for identification purposes with the server, and it holds the value of the current branch. Snapstore uses the current branch to pull up the most recently used branch on startup.

4.1.2 Upstream

On the upstream server, the snapshot, branch, group and tag data structures are the exact same as those on the client. They are kept consistent with each local repository when a socket connection is open.

The user model on the server is a mapping of users to branches to which they have access. It uses this mapping to add users to appropriate socket rooms when they first connect to the Snapstore server. Once users are in those rooms, they can be updated with changes to that branch.

4.2 User Interface

Pictures will go in here when the UI is done.

4.3 DESQ Algorithm

4.3.1 Shared Branches

For Snapstore, we wanted users to be able to work on a shared branch. As described earlier, a shared branch is a line of development where a change from one user is propagated to all other users on that line as soon as a network connection is available. If there are multiple connected users on a shared branch, a change made by one of them should result in changes to the filesystems of all other users, so as to keep all local working directories consistent.

If there are multiple users on a shared branch, they should each be able to work independently, confident that any changes they make will not be lost. These changes, whether they deal with snapshots, groups, or tags, should be persistent. New changes can come in through the network while a user is working, but it should not affect their ability to send their own changes.

We have opted to use a last-write-wins methodology when collaborating between users on the same branch because it is an easier paradigm for non-technical users to understand. Plus, with the potential amount of conflicts between snapshots on a shared branch, the number of merges would be too high. So, merging is only done between branches on the local repository.

While this methodology might cause some snapshots created to be very far removed from their original parent, we believe that it is appropriate for two reasons. First, in the current highly connected environment of today's computing, making that many offline edits is typically done by choice. Second, if offline edits are indeed an issue, Snapstore allows users to create a separate branch for highly disconnected development.

4.3.2 Network Issues

The workflow described above can be difficult to maintain. Multiple users can be making multiple edits at the same time, increasing concerns of concurrency. Furthermore, network concerns and partitions increase the difficulty and uncertainty of this problem.

Imagine a user goes offline and makes multiple snapshots and groups, all while their shared file is being written to by other, online users. Snapstore should be able to handle their reintroduction to the network without destroying the branch history. Simply trying to push these changes would be pushing an incorrect snapshot graph structure to the server.

We take the approach that any data that reaches the upstream server and is accepted should be regarded as fact. Any events that are confirmed by the server should not be undone. With this invariant, we can more adequately reason about how to propose a protocol algorithm for this process, an algorithm we call Distributed Event Sychronization Queue (DESQ).

4.3.3 DESQ

The DESQ algorithm seeks to synchronize all event queues - the server queue and all client queues - and to reach eventual consistency in the ordering of their events. The events in Snapstore are the creation, update, or deletion of any data structure. The queue of events is the collection of events on the client.

For Snapstore, it is important to maintain a consistent snapshot graph between users, but it is also important to maintain the ordering of events created by a single user. If a user creates a snapshot and then creates a group with that snapshot it in, it is necessary to send those two events in order to the upstream.

This algorithm tries to push these events to the upstream repository. In Git terms, if there is a conflict, the branch rebases and tries to push that event again. This continues until all events are successfully pushed. The rebasing keeps the snapshot graph and the workflow for the branch linear. While Git creates new commits during a rebase, Snapstore uses existing snapshots in the resulting snapshot graph.

These events describe a set of database actions and therefore describe an ordering that all parties can agree upon for system-wide accordance. When a new action in the database is triggered, that event data is saved to the client event queue. This queue, by itself, is a guaranteed in-order sequence of all database actions by the client. Each event in the queue is related to its parent and its child by a pointer, and these pointers are used to detect inconsistencies. If the client is working by themselves, in their own branch, this queue will simply be mirrored by the server when the network in connected. If, however, the client is working with another client on the same branch, there may be concurrent events being sent to the server. This can result in issues of ordering across the system.

DESQ is an algorithm that begins when an inconsistency is detected in the system. This can happen in two ways. First, if there is an event in the client's event queue, the algorithm will try to get that event confirmed by the server and shared with all appropriate users. Second, if a client connects to the server and detects that changes have been made to the server's queue, it will pull in those changes to make the queues consistent. Once the algorithm begins, it will not stop until the inconsistencies are resovled.

Note that this protocol can proceed only when network connections between the server and client are open. If they are closed, the events are queued in the client until the network is available. Then, they are processed in the same way.

Confirmed Events

In the most basic case, a single client is creating events in their own queue, with no other users having access to that queue. For example, if the client creates a snapshot, the snapshot will be wrapped in an event and added to the event queue. The event will be sent to the server, and the server will see that no additional events have been added since the parent of this new event (the parent of the snapshot). The server will add this snapshot to its version of the queue, and send a response back to the client. On the client machine, the event is registered as confirmed and removed from the event collection and queue.

Receiving Events From the Server

When sharing a particular branch, more than one client will have read and write access to it. If a certain client sends an event to the server and it is confirmed, then that event must be propagated to all other involved clients. When an event is confirmed, the server will find all clients that have access to that event's branch. It will then push that event to those clients. Because this is a confirmed event coming from the server, the other clients can apply this event to their local repository, knowing that all queues are still consistent in the system.

When an event comes in from the server to the client, if it is a snapshot event, its parent snapshot should be the end of the client's queue. This is because the rejected snapshot protocol (which can be happening simultaneously) will already be including the snapshot with its response, so there's no need to apply it in this case.

Rejected Snapshots

To combat the concurrency issues, DESQ takes an approach that results in a last-write-wins methodology. This only applies to snapshots because they are the only type of event that can cause a conflict. When a snapshot event is logged to the client's queue, it is sent to the server to be verified. The server then verifies whether or not it has seen a different snapshot event from another client. This is done by checking the parent of the incoming snapshot. If the parent of the incoming snapshot matches the last known confirmed snapshot on the server, it is allowed in. If another client has already sent a snapshot event that has been confirmed, then that snapshot will not match the incoming snapshot's parent.

If the server has seen other snapshots, making the server's history inconsistent with the client's history, it rejects the client's snapshot event. The rejected snapshot then goes back to the client, along with the snapshot(s) that caused the rejection. The snapshot(s) that caused the rejection are found by traversing child pointers. These additional snapshots are inserted at the end of the client's snapshot graph (before the rejected snapshot). The rejected snapshot is sent to the server again for confirmation, restarting the algorithm.

Because the rejected snapshot is kept at the front of the event queue to be sent to the server, this process can continue without disrupting the inherent correct ordering of events for a single client. So, if a client has made multiple offline events, only the first of those could trigger the rejection.

Furthermore, this algorithm allows the system to handle consecutive rejections. This can occur in the case where other clients are sending snapshot events to the server while another clients' events are beign rejected.

Duplicate Events

In the case of network outages, it could be the case the client goes down before the server can respond that it has received an event. In this case, when the client comes back online, it will simply retry to send that event. Because the ID of that event's data structure already exists on the server, it will simply respond that it has already received the event. This will allow the client to confirm the event.

Chapter 5

Evaluation

A report on our personal experiences using Snapstore for the past few months is included below. Ideas for future evaluations can be found in chapter 7.2.

5.1 Usability

5.1.1 Conveniences

Snapstore is able to fit all of the concepts into the application with little wasted space and an overall small interface. This can be attributed to the bare bones conceptual design of Snapstore and the methodical placement of these concepts in the interface itself.

Snapstore's built-in file navigation makes it simple to search through all of the files within the Snapstore folder. This makes it easy to locate snapshot graphs and see the snapshots inside them — though searching through these snapshots is difficult, as explored below.

Creating branches and snapshots within those branches is extremely simple. Branch creation is a one-click operation. Also, because snapshots are automatically taken, there's no need to worry about creating commits to save your work. These operations have been optimized for ease and speed due to their conceptual importance in Snapstore.

5.1.2 Issues

The user should be able to change the frequency with which snapshots are taken. However, even with this ability, snapshots will tend to be taken too often. This will result in a cluttered database as well as a cluttered interface. The snapshot graph will lose meaning if there are too many snapshots to navigate it effectively. This is especially true because the only identifying traits of a snapshot in the graph are its color (what kind of snapshot it is) and the content that shows up in the interface when you click on it (the content of that snapshot). This makes searching through the snapshot graph very inefficient.

The act of cloning a branch involves selecting which snapshots to copy over to the new, cloned branch. This process will suffer similar issues as the cluttered snapshot tree, at a greater scale. For each file (snapshot graph) that a user wishes to bring over to the cloned branch, the user must select each snapshot they want to copy. Selecting every snapshot or no snapshots from a graph is easy, but picking and choosing subsets is very difficult.

The operations that a user can perform on a file in the UI are not clear. For each file, there is both a file icon and a file name. The file icon opens up that file in an editor of the user's choosing. However, the file name performs a function that is not obivious to the user, it opens the snapshot graph for that file. Clearly, this is an important function, but there are no affordances for the user to know about this functionality.

Chapter 6

Related Work

6.1 Design Case Studies

6.1.1 Applications of the Theory of Conceptual Design

In [2], the authors used Conceptual Design Theory and applied it to version control systems. The authors studied Git to understand why it seems to fall short of users' expectations. The authors analyzed those issues by using Conceptual Design to explain Git's design issues as operational misfits of its underlying concepts. They then fix those operational misfits by constructing a new conceptual model and system, called *Gitless*, built on top of Git. The design of Snapstore involved using the theory of conceptual design the design a new version control system from scratch.

This study [2] goes on to enumerate a set of purposes for version control systems. These purposes cover all of the existing concepts in every version control system and were used in the creation of Snapstore. They provide the benefit of allowing the designer to simplify some concepts and remove others; this makes the entire system easier to model cognitively.

In [6], the author performed another conceptual analysis case study, this time focusing on Dropbox. The author first researched and polled users for the areas of Dropbox they find most confusing. They then used Conceptual Design Theory to find the operational misfits that caused this confusion. The result was a remade conceptual model of Dropbox, one that was cleaner and easier to understand. Again, this study focuses on one file sharing system, while Snapstore's creation focused on designing a version control system from scratch using the theory of conceptual design.

6.1.2 Application of Other Design Theories

Other design theories and tools exist. One such is the cognitive dimensions framework [3]. This framework asseses a design and shows cognitive consequences for various design decisions. It is used as tool throughout the design process to evaluate design decisions that have been made. Conceptual theory, employed by Snapstore, is not a tool used to look back at decisions or systems for evaluation [4], it is a tool that should be used from the very beginning preemptively to guide decisions.

Heuristics for user interface design have been used often, such as in [5]. These are general rules of thumb, not specific guidelines. Conceptual theory, on the other hand, has strong rules that cannot be broken, and it can be used in all dimensions of software design. Snapstore benefits from conceptual design guiding both its back-end and front-end development.

6.2 Version Control and File Syncing

Before the conceptual design of Snapstore, we studied the the version control and file syncing software spaces. Systems such as Dropbox, Google Drive, Git, Mercurial and more were studied from a user's perspective to see how they accomplished various tasks that Snapstore would cover.

6.2.1 File Syncing Tools

The acts of grouping changes and recording coherent points in development is not well supported by file syncing tools. Continuous saving is supported in Google Drive and Dropbox, but they do not allow a user to group changes or record coherent points. These users can leave comments on changes but cannot do so across multiple files.

Snapstore accomplishes this with the group and tag concept.

The act of merging is not well supported by many file syncing tools. Google Drive preserves every edit made on a shared document through a process called operational transformation. When a conflict arises, a newline is inserted. This is fine for some text documents, but it can break code. Dropbox does not allow files to be merged at all. With these tools, any merging must be done manually. Snapstore simplifies the shared document by keeping the line of development perfectly linear, so as not to adversely affect white space sensitive documents. It also allows file merging between branches to give it the power of a version control system.

The file concept is prohibitive in many file syncing tools because the filename is a unique identifier. In Dropbox, for example, renaming a file offline and reconnecting to the network will upload an entirely new file with a new history. These systems cannot track renames because their offline support is limited. Snapstore uses the local repository to track offline edits exactly the same way it tracks online edits. With this, Snapstore can track renames and maintain file history.

6.2.2 Version Control Systems

Many version control systems also couple together the motivating purposes of grouping changes together and recording coherent points. Git, for instance, combines the commit with the act of labeling the commit. Snapstore decouples these purposes by allowing users the granularity of a single snapshot and by allowing them to later group snapshots and tag those groups.

Creating branches and copies of files varies greatly between version control systems and file syncing systems. Git allows a user to create a branch, a fork, or a clone. This array of options is simplified to just a branch in Snapstore.

The file model of these systems is typically file name dependent. In Git, if a user renames a file without using the "git" command, Git will see that as deleting an old file and creating a new one. It is similarly handled in Gitless. This limits the history of commits and history of the file. Snapstore achieves a seemless history by watching the user's filesystem for name changes. It is able to log renames to files as rename snapshots and keep the snapshot history consistent.

Finally, version control systems typically cannot save or pull in changes without a direct command from the user. Git and Gitless require the explicit “commit” command to do so. The push/pull model of Git is unnecessary. Snapstore automatically pushes out and pulls in changes to a branch.

Chapter 7

Future Work

Snapstore is currently a minimal viable product. However, there is more work that needs to be done before it is fit for version control and for distribution. These improvements fall into two general categories: implementation and user study.

Implementation improvements include completing the implementation of the conceptual design and functionality goals of Snapstore, as well as improving the user interface. A user study needs to be done with real users to help improve the user interface and provide a quantitative evaluation of Snapstore's features, comparing them to features on existing systems.

7.1 Implementation

7.1.1 Functionality

The first area of funtionality that needs to be addressed is the grouping and tagging of snapshots. This area is pivotal for more managerial and administrative parts of version control. Beyond their data structure creation, they will need to be maintained by their branch in the same way snapshots are. Any changes must be propagated to all users with access to their branch. Front end components that interact with these structures will also be needed.

The next feature that needs to be implemented is the ability to clone and merge branches. This development is necessary for powerful parallel development. These cloned and merged branches are the same as regular branches, but the ability to merge and to clone them will need to have hooks in the front end. These front end hooks will need to be easy to use; one potential interface problem we forsee is choosing which snapshots to clone over to the new branch. Choosing these snapshots, for multiple cluttered snapshot graphs, might prove difficult.

The user should also be able to change the frequency with which snapshots are taken. This will help reduce the total amount of snapshots taken and clean up the snapshot graphs, making them easier to navigate.

A final feature is the ability to change the user's upstream location. This location is currently just a static value, pointing to a specific IP address, in the application, so it can be easily modified. However, this process will take work on the part of the end user to set up their own Snapstore server.

7.1.2 User Interface

The user interface, up until the writing of this paper, has been designed with function in mind, not aesthetics. Components have been aligned, and they are well labeled. However, no principles of good interface design have been consciously applied, and it overall has not been a strong focus. A redesign of the front end is necessary to garner a significant user base.

User interface design and conceptual design are related. Conceptual design dictates what concepts will appear in the interface and how they will interact with each other. User interface principles can be applied to simplify the interaction with these concepts.

However, user interface principles can also be applied in a way that confuses users. Concept overload (having one concept fulfill more than one purpose) might be applied to simplify the interface at the expense of confusing the user. This is the case with Dropbox's shared folder deletion [6]. It is important that future interface work not obscure the conceptual design that it is trying to show the user. It is the goal, after all, that this conceptual design becomes to user's mental model of the system.

Future iterations of the user interface should also minimize the amount of functionality accessible through only the Snapstore application. Like Dropbox, Snapstore functionality should be accessible from native file managers like the Mac OS Finder. This accessibility should include file-specific functionality like reversion and accessing a file's snapshot graph.

Snapstore would also benefit from another Dropbox feature, a presence in the menubar. The Snapstore menubar icon would allow users to easily see recent changes to a branch, elect to work offline, and perform other network-related tasks.

7.2 User Study

A user study will of course test how well participants like the functionality provided by Snapstore and the interface used to provide it. An interesting result of these studies will be the differences and overlap between the high tech and low tech participants.

Thus far, we have evaluated Snapstore from a conceptual design theory perspective and from a personal, subjective perspective. However, in these user studies, we can gather more objective and quantitative data along with the subjective opinions of the study participants. We can measure the time it takes for participants to perform actions they are used to performing with their file sharing or version control system. With those numbers, we can quantify the benefits that Snapstore brings the end user.

Chapter 8

Conclusion

Snapstore is a version control system that tries to bridge the gap between version control systems and file syncing systems to being the best of both worlds to users. Its motivating purpose is to be simple enough for every user and powerful enough for every user. By following the theory of conceptual design, we believe Snapstore's design has acheived that.

Snapstore still has implementation-specific needs and flaws which will need to be addressed before it is widely used. By open-sourcing this project, we hope it development will continue and its full design is realized.

Bibliography

[1] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? Technical report, Oregon State University, 2014.

[2] Santiago Perez de Rosso. Purposes, concepts, and misfits, in git. August, December 2016.

[3] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7:131–174, January 1996. This is a full ARTICLE entry.

[4] Daniel Jackson. Towards a theory of conceptual design for software. Technical report, MIT, 2015.

[5] Jakob Nielsen. 10 usability heuristics for user interface design. Technical report, Nielsen Norman Group, 1995.

[6] Xiao Zhang. A conceptual design analysis of dropbox. August, December 2014.