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 to 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 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 File Syncing and Version Control Systems

1.1.1 File Syncing Systems

File syncing systems have become popular over the past few years. Systems like Dropbox and Google Drive are simple systems that allow users to backup their files and share files with other users.

These systems are often characterized by how easy it is for users to learn and use them effectively. Some, such as Dropbox, are popular for situations in which concurrent work is not being done because Dropbox does not merge conflicting edits to a file. Google Drive does allow concurrent editing on their online app, but they do so in a way that can drastically change the file's composition. Sometimes, Google Drive will create new lines and extra white space to accomodate the conflict.

File syncing services also allow files to be shared from user to user. Dropbox allows users to share folders with other users, thus making that folder a "shared folder". This shared folder can exist on a user's computer, allowing them to make edits that the original owner will see. These edits can be made offline, but each user must connect to a network in order to send and receive those edits. Sharing files in Dropbox is only possible by sharing a link that allows another user to edit that file online. This online editing is down through the browser, so the file cannot be edited offline. Google Drive allows similar functionality, though they do allow users to share and then edit individual files offline.

1.1.2 Version Control Systems

Version Control Systems (VCSs) are also common today in certain industries. Systems such as Git and Mercurial are often used by software developers to maintain their

projects and share code.

Like file syncing systems, VCSs provide data backup and data sharing. However, they also provide more functionality to help with project management and efficient development. VCSs allow users to create parallel lines of development that they can later merge, and they allow users to label changes and project milestones. VCSs can be centralized systems and allow users to collaborate through a shared, central repository; or, they can be decentralized, allowing users to collaborate with each other directly and work offline. VCSs can also support complex workflows. These workflows allow users to distribute work among themsleves, hide portions of the project from certain users, and manage contributions to the main project by vetting them first.

The extra benefits of VCSs come at a cost. VCSs are often difficult for non-technical users to learn; even some expert software developers resign themselves to using a few basic commands and do not explore the full extent of these systems.

1.1.3 Issues with VCSs and File Syncing Systems

Due to their overlapping nature, this paper explores both file syncing systems and VCSs, finding benefits and flaws within each set. Those flaws come in two major categories.

The first issue these systems have is they are too narrow in their domain. Rarely do software developers use Dropbox to collaborate on software. Dropbox does not have the functionality required by most complex software projects. The ability to work on independent lines, merge those independent lines, and perform tasks such as grouping and labeling changes are not possible.

It is even rarer that a non-technical industry would use a system like Git for sharing files. Git is a complex system with a huge learning curve. It's advanced functionality, like branching, would not be useful for a non-technical team trying to share files. Unfortunately, it is difficult to use Git at a basic level without somewhat exposing yourself to the complexities of its more advanced functions.

In both cases, the system was designed for a specific user group, not for the underlying purposes the users needed the system to fullfill. Technical systems are

built for technical people, with a steep learning curve only conquerable by power users. More basic systems are too simple and lack 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. Novices, especially, find some of Git's design choices confusing such as its inability to allow a committing of an empty directory [3]. Supposedly simpler file syncing systems like Dropbox have similar design issues. The way that Dropbox's shared folder model operates has left many users confused[6].

This paper presents a new VCS that can also serve as a capable file syncing system, Snapstore. We believe that all of the issues associated with current VCSs and file syncing systems can be solved by focusing on the design of our VCS at the conceptual level. This system leverages the best from the technical and non-technical systems available today. It is designed using essential purposes and concepts in order to make user startup 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 entire system 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 brings together all of these past, disjointed ideas. 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 needs an unspoken medium of communication between the designer and the user. Conceptual design gives that medium a language.

A given concept is accompanied by a motivating purpose, its reason for existing. A purpose is a desired result. It is not a piece of code, a design detail, or a way to achieve a desired result. The purpose behind the trash can, for example, 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.

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 backup. Then, if desired, users can explore more advanced features of the system.

Snapstore operates within a specially designated Snapstore folder which is created upon opening the application for the first time. It is similar to the Dropbox folder; Snapstore only looks at files that are inside of it. Snapstore watches the user's filesystem for changes in order to respond with certain actions like creating snapshots (section 2.1.1). This allows users to use any editor with Snapstore. Also, the Snapstore folder that Snapstore creates does not need to remain the only folder used by the application. Snapstore can be opened using any folder as the Snapstore folder.

2.1 Basic Features

The basic features of Snapstore allow a single user to use the application like they would a file syncing system.

2.1.1 Snapshots

Snapshots allow a user to persistently store all of their edits to a file over time. Whenever a file is saved to disk, a snapshot is created with the contents of that file and stored in the local repository. The type of each snapshot corresponds to the action that created it. Snapshots can be the result of a create, update, rename, delete, merge, or conflicting merge of a file. Users can retreive an old state of a file by finding the appropriate snapshot in the file's history.

Snapshots are created when a file is created, updtaed, deleted or renamed. Snapshots are also created when files are merged together. These snapshots are either the result of a successful merge or of a conflicting merge. Snapshots, then, can be one of six types: create, update, delete, rename, merge, or conflict.

When creating snapshots for a given file, Snapstore will add the snapshot to that file's *snapshot graph*. This graph represents the 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 only way a snapshot could have more than one parent is if it's a product of a merge of multiple snapshots. The first snapshot in a graph is called the *root*, and the last snapshot in the graph is called the *head*.

To show a sequence of snapshots from a file's history, navigate to that file within Snapstore's interface. Then, click on the file's name. The snapshot graph will appear at the top of the window. Each node in this graph is a snapshot, and the node on the far right is the head. By clicking on these nodes, the content of the snapshot will appear on the bottom-right of the window. After the correct snapshot is found, it's possible to revert to that snapshot by clicking "Revert", a button above the snapshot content. Reverting to a previous snapshot will create a new snapshot with the same

content at the head position and alter the file's content to match the snapshot's content.

2.1.2 Upstreams

Upstreams are used to synchronize the data of collaborators on a project. Whenever a user makes any changes to their Snapstore system, those changes are sent to the upstream and out to all other users who are collaborating with that user.

If two users are working together on a project, the upstream will synchronize their snapshots as they make them. In the case of multiple snapshots coming in to the upstream at the same time, the upstream will resolve the conflict and push the same ordering of snapshots to both users.

By default, the upstream is the Snapstore server, but users might want to change their upstream so their data passes through a known location. To do so, follow these 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

2.1.3 Local Repository

The *local repository* allows users to work without an active Internet connection. Whenever a user makes any changes to their Snapstore system that data is saved in the local repository. When connection is restored, Snapstore will push all new changes to the upstream.

2.2 Advanced Features

Snapstore's advanced features allow users to access the more powerful components of a version control system. They provide additional functionality that users might

want when working on projects with complex version control requirements.

2.2.1 Groups

Groups allow users to designate a collection of snapshots as related. When a user has a collection of snapshots they believe are related, even if they exist across multiple files, they can place them in a group. The user can then give this group a name.

A software team collaborating on a project might want to fix a syntax bug in their program. If one user makes a few snapshots in this process, they can then place them in a group and title it "Fixed syntax bug". Now, these snapshots and this group have all been shared with the other team members through the upstream. Team members can easily locate this group and inspect its snapshots to see how the bug was fixed.

2.2.2 Tags

Tags allow users to designate a group as a coherent point in development. The exact nature of a coherent point will differ from project to project, but in general this means a point in which the project is ready for further development. When a group of snapshots is particularly significant, such as project completion or a release of a piece of software, users can tag that group.

Tags can only be given to groups who have at most once snapshot per file. This is so that a the user can revert the state of their files using the tag. When this is done, every file that is in the group with that tag will be reverted to the state described by the snapshots in that group.

For a software release, a user can tag the group of head snapshots "Version 1.3" to signify that the project is in a stable release state. Later, after more snapshots have been created, users can utilize this tag to revert the project to the group of snapshots tagged by "Version 1.3".

2.2.3 Branches

Branches allow users to separate parallel lines of work. Whenever any data is created, it is saved within the user's current branch. Switching to a different branch will load all data associated with that branch, changing the user's filesystem as necessary. Branches can be merged together to combine parallel lines of work.

Snapstore provides a default branch called "master" on startup. A user can then create and use new 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 [3].

A branch can also be created by cloning an existing branch. Creating a clone involves choosing a branch to clone and selecting snapshots inside the original branch to bring over to the clone. For all snapshots that are cloned into the new branch, any groups associated those snapshots, and any tags associated with those groups, will be copied to the cloned branch.

Sharing Branches

When a branch is shared with another user, that branch's data is copied on that user's local repository. On a shared branch, when any user makes changes, those changes are immediately sent to the other user(s) associated with that shared branch. For example, one user might create a new snapshot on a file in the shared branch. That new snapshot is propagated to all collaborators on that branch and reflected in their snapshot graphs and on their filesystem.

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

Figure 2-1 illustrates an example of two user working on a shared branch. When two snapshots are created at the same time, one will reach the upstream first. When

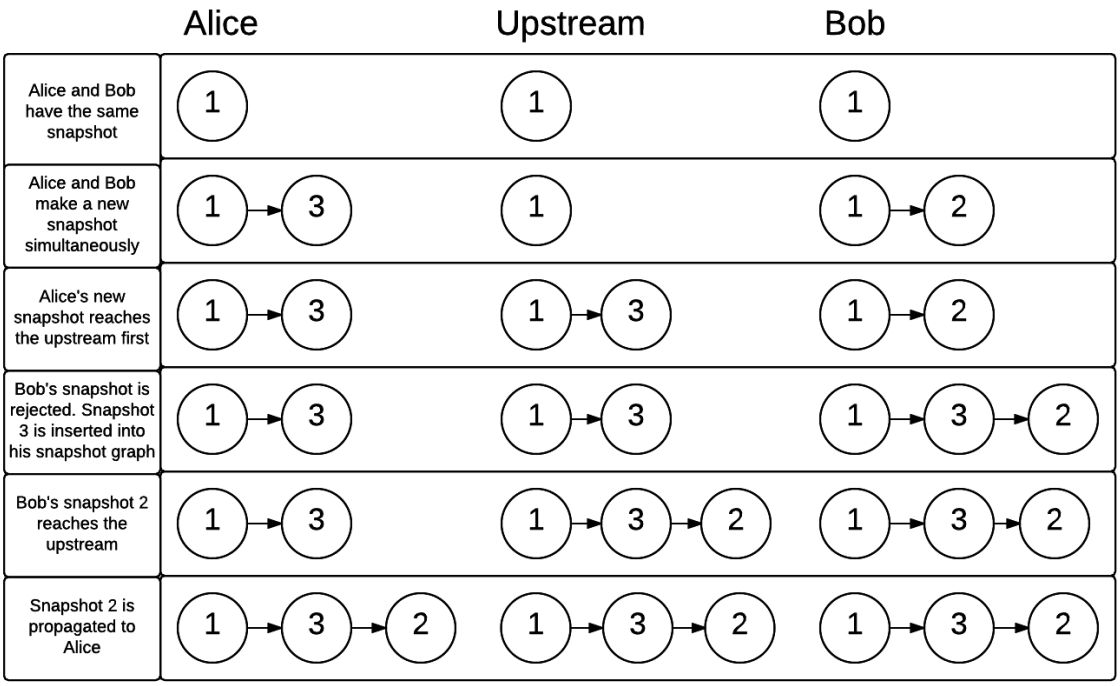

Figure 2-1: Conflicts being resolved on a shared branch.

this happens, that snapshot is confirmed and cannot be changed. When the second snapshot reaches the upstream, it will be rejected. Any snapshots that caused the rejection will be returned with the rejection to be inserted into the user's snapshot graph. The snapshot is then sent again and, if it successfully added to the upstream, it will be sent to all collaborators on that branch.

Merging Branches

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 three way merge and their common ancestor. This will result in a merge snapshot with as many parents as there are snapshots being merged. If there is a merge conflict, then the resulting snapshot will be a conflict snapshot. Like in Git, a conflict snapshot's content will show where the conflict needs to be resolved. Unlike in Git, 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

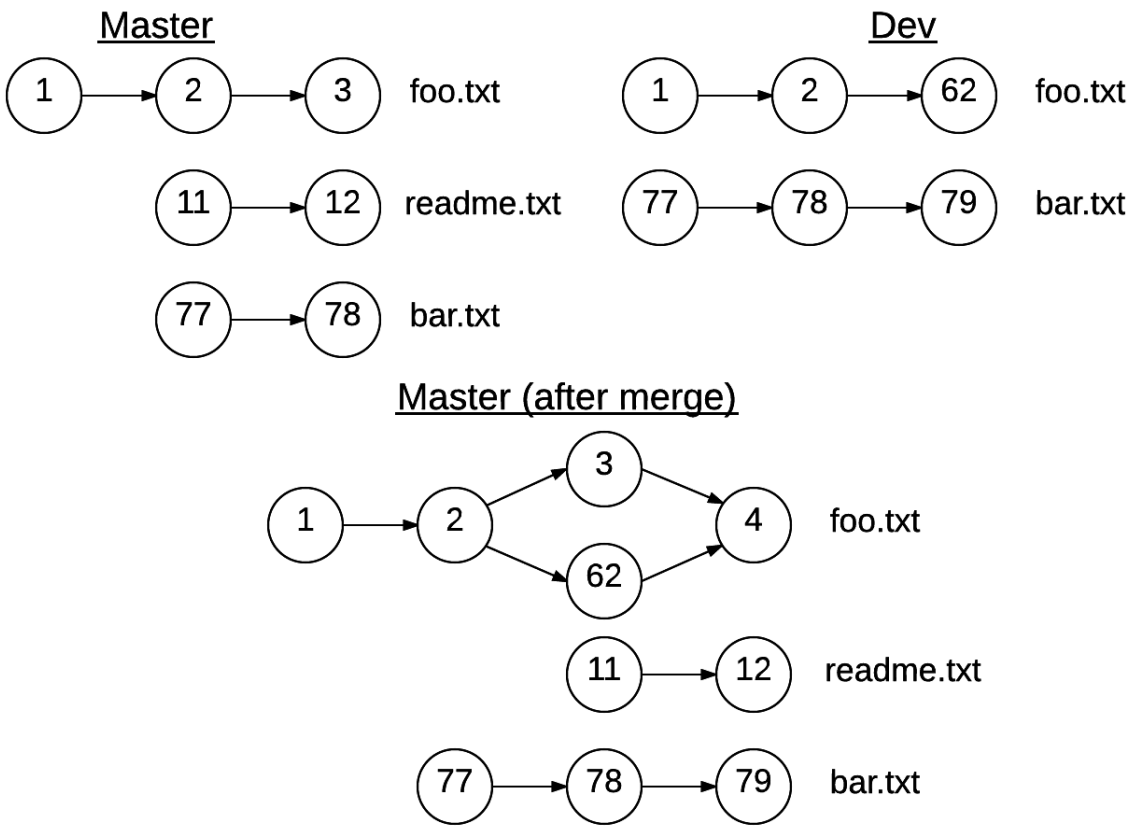

Figure 2-2: Merging two branches.

reflects the fix. Merging two branches will keep all of the group and tag data from both branches.

Figure 2-2 shows an overview of branch merging in Snapstore. In this image, the file id is shown next to the head snapshot for each snapshot graph. Once snapshot graphs are identified as corresponding to the same file id, their head snapshots are merged into a new snapshot, whose parents are the old head snapshots. Any snapshot graph that does not have a counterpart in the other branch will just be copied over to the merged branch.

2.2.4 Use Cases

Single User on Multiple Computers

Git users can sympathize with the hassle of having to commit, push and then pull whenever changing computers. Snapstore solves this using upstreams and local repositories.

Imagine the a user has two computers, A and B. Both have a file called "foo.txt" on them. The user works on computer A, making edits to "foo.txt". These changes are saved to their local repository. When they open Snapstore on computer B, computer A's local repository will push those changes to computer B through the upstream. Now, computer B has an updated version of "foo.txt" that has the exact same content as "foo.txt" on computer A. There is no push/pull model in Snapstore. Data is automatically propagated to every local repository that has access to the data.

Project Partitioning and Re-Assembly

Snapstore uses branches to allow users to partition their projects; they can they distribute these partitions to other users. This helps manage projects, and it can produce powerful workflows. Snapstore can achieve a workflow similar to that used by the Linux project and its system of trusted lieutenants who vet incoming contributions before passing them on to the project owner[2].

Imagine that a website is be developed called "Iweb". The team creating this website has a project manager, a back-end developer, and a front-end developer. There is a master branch, "Iweb-master", that the project manager has access to. In order to partition this project, the project manager makes two clones of this branch, "Iweb-front" and "Iweb-back". She then shares "Iweb-back" with the back-end developer and "Iweb-front" with both the front and back-end developer.

A graphical representation of these branches, along with who has access to them, is included in figure 2-3.

When edits are made by the developers, they are immediately seen by the project manager because they share the same branch. When the project manager decides

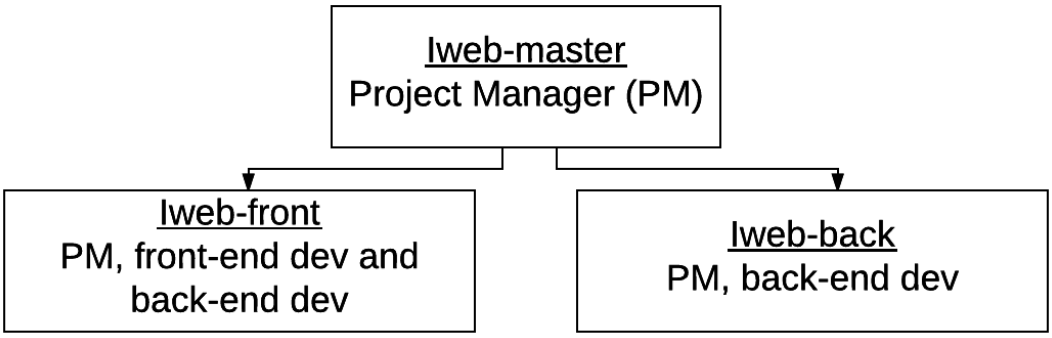

Figure 2-3: Icorp's branch hierarchy.

that a certain branch, like "Iweb-front", is in a stable state, they can merge it with "Iweb-master".

This hierarchy of branches accomplishes two things. First, it allows the project manager to vet edits to the cloned branches before merging them onto the master branch. Second, it allows some work to be hidden from some employees, achieving effective access control. The front-end developer does not have access to any of the files in "Iweb-back".

This hierarchy of branches can be expanded, allowing Snapstore to achieve a workflow similar to that of very large software projects, such as Linux.

Chapter 3

Design

The design of Snapstore had two steps. The first was to identify the purposes of a VCS. The second was to create a conceptual model composed of concepts that fulfilled the purposes of a VCS.

3.1 Purposes of Version Control

The six purposes of a VCS identified in [3] were used in the design of Snapstore. A purpose graph is included in Figure 3-1, where each subpurpose points to a purpose. There are five classes of VCS purposes: data management, change management, collaboration, parallel development, and disconnected development.

Data Management deals with the notion of backup. In case of failure, these backups need to be stored persistently and be able to be retreived. This purposes allays risks associated with development such as accidental deletion and incorrect saves, not just machine failure. The ability to track/untrack files to track types of edits to files provides more granular persistent saves for the files.

The second class, **change management**, deals with managing these edits. Grouping changes allows the user to divide the history of a file or files in logical segments. Groups like current file state help users accurately model the state of their project. Tagging these groups as coherent points in development aid in administrative and managerial tasks associated with long or large projects. These coherent points in the

Figure 3-1: Purpose Graph of Version Control Systems.

project can then be returned to by reverting the project to reflect the versions of files in that collection.

Collaboration concerns a system shared by multiple users. Synchronizing the changes of collaborators on that system allows their edits to be amalgamated in such a way that conflicts are avoided when possible and made explicit when they cannot be avoided.

Parallel Development, the fourth class, mainly deals with supporting parallel lines of development. Switching between parallel lines, as well as merging parallel lines are needed to fully support parallel development. This purpose allows users more flexibility to isolate parts of their work from others, and to develop without affecting the main line.

The final class, **Disconnected Development**, allows operations to be performed in a disconnected mode. This motivates any work that a user can do in an offline setting or in a setting where a user is willingly disconnected from their collaborators.

3.2 Conceptual Model

A conceptual model of Snapstore was then created composed of single concepts. These concepts were invented in a way that is aligned with conceptual design theory [4]. They 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. Table 3.2 shows each Snapstore concept and its operational principle. A graphical representation of the conceptual model, using the notation for extended entity-relationship diagrams as used in [4], is shown in Figure 3-2.

3.2.1 Data Storage — Snapshot

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

Concept	**Motivating Purpose**
Snapshot	Make a set of changes to a file persistent
Create, Update, Rename, and Delete Snapshot	Track various types of changes to a file
Snapstore Folder	Provide a platform for users to edit files
Tracked File	Mark files whose changes should be saved
Untracked File	Mark files whose changes should be ignored
Group	Group logically related changes together
Head Snapshot	Easy way of naming the current snapshot
Tag	Represent and record coherent points in history
Upstream Repository	Synchronize changes of collaborators
Branch	Support parallel lines of work
Conflict Snapshot	Mark snapshots with unresolved conflicts
Merge Snapshot	Indicate that a snapshot is the result of a merge
Local Repository	Do work in disconnected mode

Table 3.1: Concepts of Snapstore and their motivating purposes.

Concept	Operational Principle
Snapshot	Whenever a file is saved to disk, a snapshot containing that file's contents is created
Create, Update, Rename, and Delete Snapshot	Whenever a specific type of action on a file results in a disk save, the same type of snapshot is created
Snapstore Folder	If a user edits any tracked files within the Snapstore folder, Snapstore will create a snapshot for that file
Tracked File	When a user tracks a file within the Snapstore folder, snapshots will be created for that file
Untracked File	When a user untracks a file within the Snapstore folder, no snapshots will be created for that file
Group	When a user places a set of snapshots in a group, they can be found later with the group's name
Head Snapshot	Whenever a new snapshot is created for a file, it becomes the head snapshot for that file
Tag	When a user places a tag on a group, that group can be found using the tag's name, and every file within that group can be reverted to it's associated snapshot content within that group at the same time
Upstream Repository	Any change to the local repository made by a user is distributed by the upstream to all collaborators of that user
Branch	When the user switches branches, Snapstore hides all of the data associated with the previous branch and shows them all of the data associated with the current branch
Conflict Snapshot	If there is a conflict when merging two head snapshots, the result is a conflict snapshot, which shows where the conflict is
Merge Snapshot	If there is no conflict when merging two head snapshots, the result is a merge snapshot
Local Repository	Whenever the user is offline, any changes to Snapstore are saved persistently in the local repository

Table 3.2: Concepts of Snapstore and their operational principles.

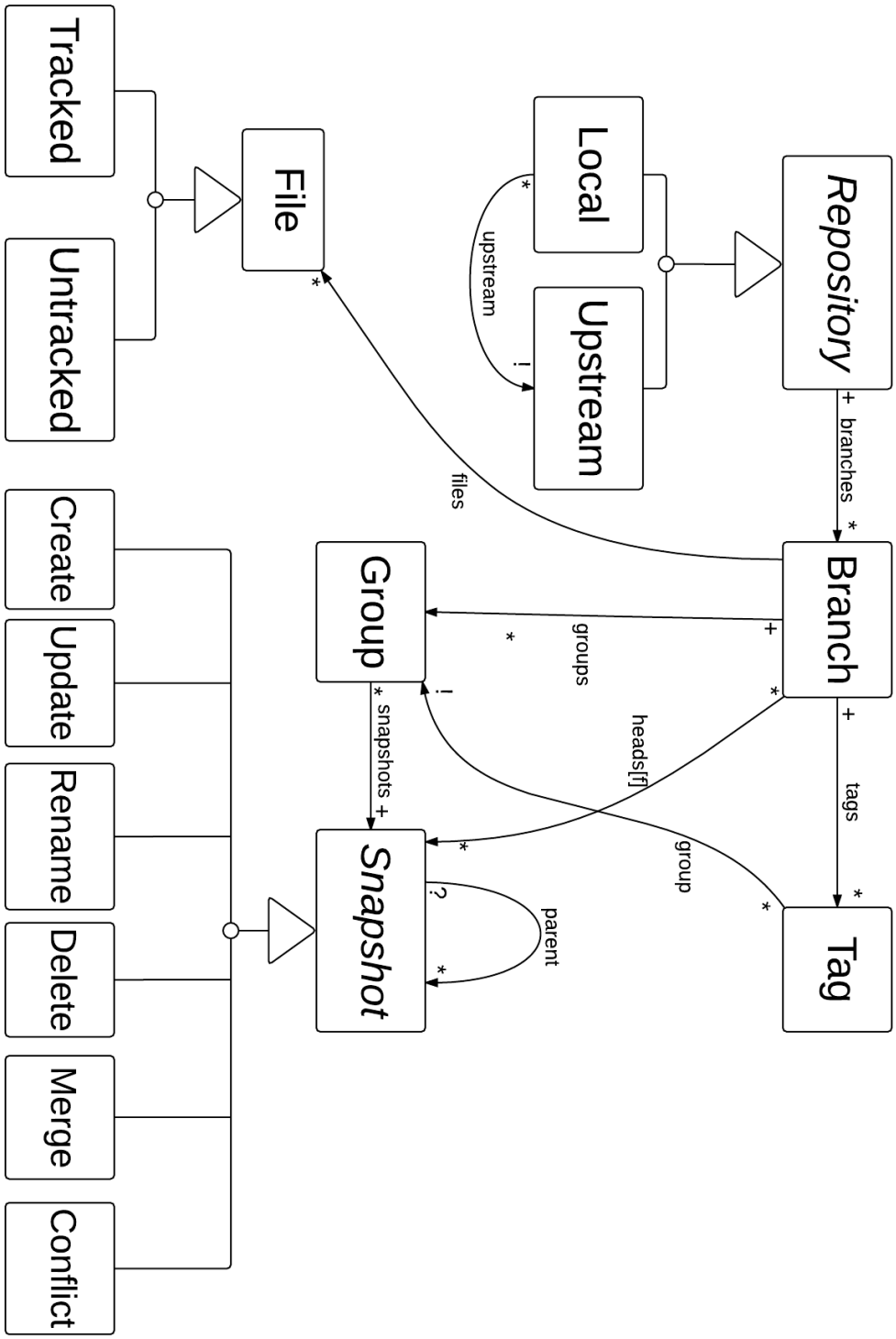

Figure 3-2: Concept Model of Snapstore.

machine.

This type of snapshot dictates the values of that snapshot's attributes. 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, though they still have a parent and can therefore be placed in the snapshot graph. 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 section 2.1.1. 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.

The snapshot graph is guaranteed to be an in-order description of snapshots a specific client has made to a file in a branch. There is no operation on the client that distorts the ordering of snapshots in the graph. The only operation that can alter the graph occurs when a snapshot graph is being shared and a collaborator inserts their snapshots somewhere in the graph. However, even if snapshots are inserted into a user's snapshot graph, the user's ordering of snapshots made by them stays intact.

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.

3.2.2 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 two group contain the same snapshots across different branches, those groups are different because they exist on

different lines of development. Any group can be given a name for identification for the user.

3.2.3 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[3].

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.

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

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.

3.2.5 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.

3.2.6 Disconnected Work — Local Repository

The ability to leverage the benefits of a VCS 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.

3.2.7 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 VCSs 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 VCS. 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[3].

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]. We used web sockets[2] for networking. 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. A data model representing each data structure and its attributes can be found in figure 4-1.

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

[1] http://electron.atom.io/
[2] www.socket.io

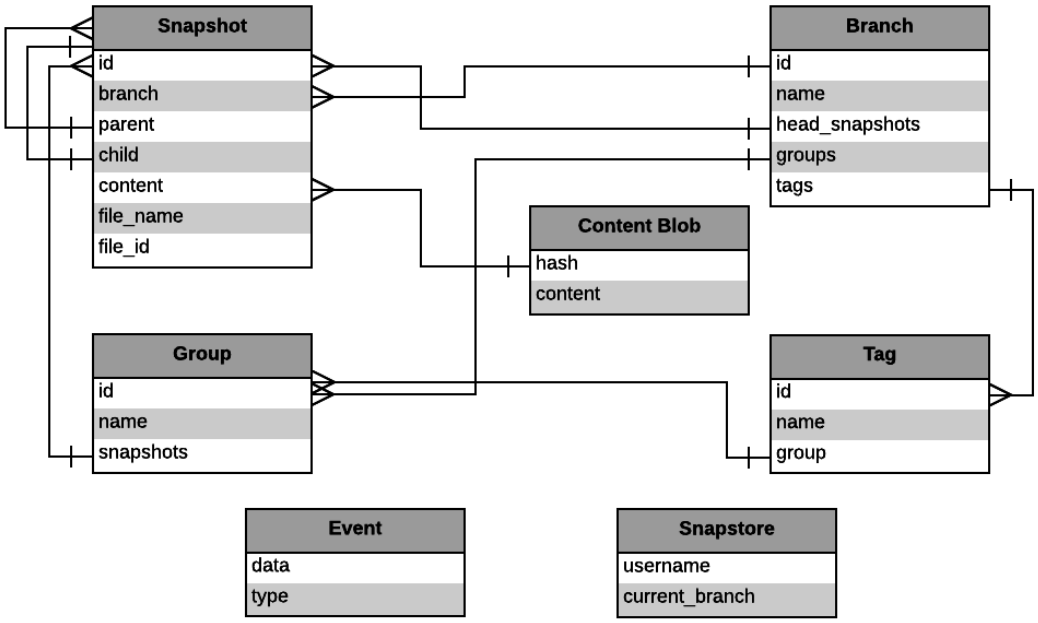

Figure 4-1: Snapstore client data model.

dictates how it will populate its data fields. A create snapshot, for example, has no parent snapshot. When any 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, while removing 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 binary content.

When a new branch is created, it is added to the database with only a name. If, however, it was cloned from another branch, the cloned branch will point to all head snapshots, groups, and tags from the original branch. When switching to a branch, the heads snapshots for the target branch are read and applied to the Snapstore folder. To read the history of a file, the head snapshot of the file is located, and the rest is found by searching backwards through the snapshot graph.

The event collection stores all of the unconfirmed snapshots, groups, and tag events on the client. As these events are confirmed, they are erased from the collection.

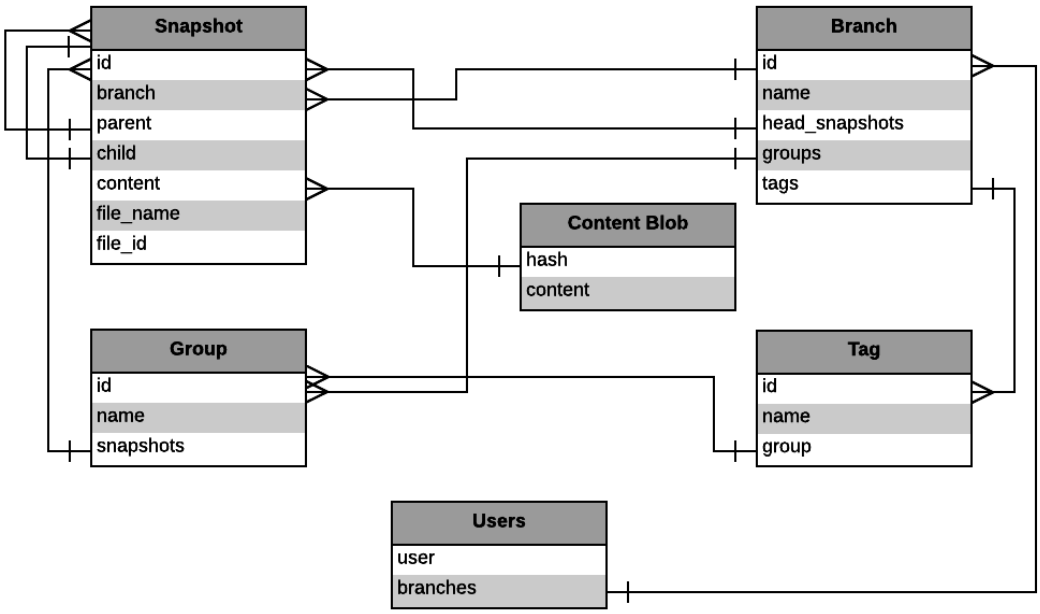

Figure 4-2: Snapstore server data model.

4.1.2 Upstream

On the upstream server, the snapshot, branch, group, tag, and content 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. A model of the data structures on the upstream is show in figure 4-2.

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 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 Keeping Data in Sync

4.3.1 Shared Branches

For Snapstore, we wanted users to be able to work on a shared branch. As described in section 2.2.3, 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.

We have opted to use a last-write-wins appraoch when dealing with conflicts on a shared branch because it is an easier paradigm for non-technical users to understand compared with merging. Plus, with the potential amount of conflicts on a shared branch, the number of merges would be very high. Because of this, merging is only done between branches on the local repository.

This approach can result in a snapshot being very far removed from their original parent. For example, say Alice and Bob share a branch with a single snapshot. Alice goes offline and makes one snapshot of her own. Bob, still online, makes 10 snapshots that are immediately confirmed by the server. When Alice returns to the network, her snapshot would be placed after Bob's 10 confirmed snapshots, far from its original parent.

Despite this, we believe this approach 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 in section 4.3.1 can be difficult to maintain. Multiple users can be making multiple edits at the same time, increasing concurrency issues. If two snapshots are made at the same time, there needs to be a way to resolve the

snapshot graph and propagate it to all collaborators on a branch. 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.

We take the approach that any data that reaches the upstream server and is confirmed should be regarded as fact; they should not be undone. With this invariant, we designed a protocol algorithm for this process, called Distributed Event Sychronization Queue (DESQ).

4.3.3 DESQ

Each client will have their own ordering of events. These events are any database operation, and they are stored in a client queue until they are confirmed by the server. The DESQ algorithm seeks to reach eventual consistency between these client queues so that every client has the same data.

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 server. Otherwise, the server will try to create a group containing a snapshot that doesn't exist.

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 the client is working with another client on the same branch, there may be concurrent events being sent to the server, resulting in ordering issues.

This algorithm tries to push these events to the server repository. In Git terms, if there is a conflict, the branch rebases and tries to push that event again. This

Algorithm 1 DESQ

```
1: procedure CLIENT-DESQ
2:     events ← collection of Events
3:     socket ← server socket connection
4:     while events not empty do
5:         socket.send(events(0))
6:     loop on socket.reponse(response, message):
7:         Save response
8:         if message == "Confirmed" or "Duplicate" then
9:             events.remove(response)
1: procedure SERVER-DESQ
2:     db ← MongoDB
3:     socket ← client socket connection
4:     loop on socket.receive(event):
5:         if event.data.id in db then
6:             return "Duplicate Event".
7:         if event.type ! = snapshot then
8:             event.confirmed ← True
9:             socket.send(event, "Confirmed")
10:            socket.room.send(event, "New Event")
11:        else
12:            if event.data.parent is head snapshot then
13:                event.confirmed ← True
14:                socket.send(event, "Confirmed")
15:                socket.room.send(event, "New Event")
16:            else
17:                conflictSnapshots ← All snapshots between event and event.parent
18:                socket.send(conflictSnapshots,"Reject Event")
```

continues until all events are successfully pushed. The rebasing keeps the snapshot graph and the workflow for the branch linear. However, while Git creates new commits during a rebase, Snapstore uses existing snapshots in the resulting snapshot graph.

DESQ 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, it will pull in those changes. 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

Events are always confirmed unless they are a snapshot event whose parent is not a head snapshot on the server. Lines 9 and 14 of the pseudo-code for Server-DESQ show a particular event being confirmed by the server. On the client machine, the event is registered as confirmed and removed from the event queue.

Receiving Events From the Server

When a client sends an event to the server and it is confirmed, the event must be propagated to all other collaborators. The server will find all clients that have access to that event's branch and send it to them. Because this is a confirmed event coming from the server, the other clients can apply this event to their local repository.

This process can be happening while a client is offline. As stated above, when the client returns to the network, it will be pushed all of these events from the server so that it can update its queue.

Rejected Snapshots

DESQ's last-write-wins approach only applies to snapshots because they are the only type of event that can cause a conflict, due to their inclusion in a snapshot graph. The server, for each snapshot event it receives, will always verify whether it has seen a different snapshot event from another client in the meantime.

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 the server's snapshot graph from the rejected snapshot's parent to the head. These additional snapshots are inserted at the end of the client's snapshot graph, but 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.

This protocol allows the system to handle consecutive rejections. This can occur when other clients are sending snapshot events to the server while another client's event is being rejected.

Duplicate Events

In the case of network outages, it could be the case that 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 retry to send that event. Because the ID of that event's data 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

Interface Space

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.

File and Snapshot Tree Navigation

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.

Branch and Snapshot Manipulation

Manipulating branches and snapshots within those branches is extremely simple. Branch creation and switching between branches are both one-click operations. Because the file system changes with respect to the current branch, there is never any confusion about which branch's file the user is working on.

Also, because snapshots are automatically taken, there's no need to worry about creating commits to save your work. Reverting to a previous snapshot, once it is found, is another one-click operation, and it edits the user's filesystem so that their computer and Snapstore are always in agreement about the contents of files.

These manipulation of both of these concepts have been optimized for ease and speed due to their conceptual importance in Snapstore.

5.1.2 Issues

Snapshot Excess

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.

Cloned Branch Snapshot Selection

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.

Operations for Each File in Interface

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 Other Applications of the Theory of Conceptual Design

In [3], 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.

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.

In both of these case studies, the theory of conceptual design is applied to a specific system for analysis. The design of Snapstore, on the other hand, involved using the theory of conceptual design to design a new version control system from scratch.

We do use the purposes of version control enumerated in [3] to guide the design of snapstore. These purposes cover all of the existing concepts in version control

systems. They provide the benefit of allowing the designer to simplify some concepts and remove others; this makes the entire system easier to model cognitively.

6.2 Version Control and File Syncing

Before the 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 [5]. 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. Snapstore uses the local repository to track every edit made within the Snapstore folder. With this, Snapstore can track renames and maintain file history.

File syncing systems have limited offline support. Google Drive allows minimal

offline capabilities on Chromebooks, and Dropbox does not monitor offline changes. Snapstore uses the local repository to track offline edits exactly the same way it tracks online edits. This allows Snapstore to push those edits once a network connection is restored.

6.2.2 Version Control Systems

Version control systems tend to 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. One potential interface problem we forsee is choosing which snapshots to clone over to the new branch. Users will be given the option to choose any subset of snapshots from every file to clone. Choosing these snapshots, from 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 func-

tionality 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.

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] George M. Dafermos. Management and virtual decentralised networks: The linux project. *First Monday*, 2001.

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

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

[5] Yi Xu, Chengzheng Sun, and Mo Li. Achieving convergence in operational transformation: Conditions, mechanisms, and systems. Technical report, Nanyang Technological University, 2014.

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