Snapstore: A Version Control System for Everyone

by

Alexander Chumbley

B.S., Massachusetts Institute of Technology (2015)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2016

Author ....
Department of Electrical Engineering and Computer Science
August 26, 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

Alexander Chumbley

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

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. File syncing applications such as Dropbox and Google Drive, while easy to learn and understand, 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. 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 system we created to fulfill the vision of a universal, simple version control system. The system, Snapstore, is the result of all the goals and ideas described by this paper.

Thesis Supervisor: Daniel Jackson
Title: Professor

Acknowledgments

Firstly, I'd like to thank my advisor, Prof. Daniel Jackson, for allowing me to work with him as an undergraduate and graduate student. It has been a wonderful experience.

I'd like to also thank Santiago Perez de Rosso. His help and collaboration made this project possible.

Finally, I'd like to thank my friends and family for their love and 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, learning 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 code base, 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[1] and Google Drive[2] 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, do not allow concurrent editing of files and are unable to 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 accommodate the conflict, which can be unwanted in some situations.

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 done 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]http://dropbox.com
[2]http://drive.google.com

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 themselves, hide portions of the project from certain users, and manage contributions to the main project by vetting them first. In Git, for example, the vetting process is done via pull requests.

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. Its 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.

It seems like each system was designed for a specific user group, not for the underlying purposes the users needed the system to fulfill. 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 [8]. 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 [10].

This paper presents Snapstore, a new VCS that can also serve as a capable file syncing system. We believe that many of the issues associated with current VCSs and file syncing systems can be solved by focusing on the design of the VCS at the conceptual level. Snapstore aims to leverage 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 conceptbased design are nothing new [2]. Leaders in the field of user interface have noted the importance of the connection between the mental model that a user has of a piece of software with its underlying, software-based concepts [6].

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 the system 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 fulfill more than one purpose. Finally, concepts should be **uniform** and, when possible, variant concepts should behave similarly.

A concept is defined by an *operational principle*, which is a scenario in which the concept fulfills its motivating purpose. The operational principle is not a formal model or a description of everything a concept does. It is a plain text description that focuses on the aspect of the concept which fulfills its motivating purpose.

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.

Snapstore's vision is that of 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.

2.1 Basic Concepts

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. Snapshots can be the result of a create, update, rename, delete, merge, or conflicting merge of a file. Users can retrieve an old state of a file by finding the appropriate snapshot in the file's history.

Snapshots are created when a file is created, updated, 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). A Snapshot can have more than one parent if it is a product of a merge of multiple snapshots. The first snapshot in a graph is called the *root*. The last snapshot in the graph is called the *head*. The contents of a file's head snapshot will always match the contents of the file on disk.

Figure 2-1 shows how to navigate a file's snapshot graph in Snapstore. You first find that file within Snapstore's interface. In this example we choose "foo.txt". Clicking on the file's name makes the snapshot graph 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. When you click on these nodes, the content of that snapshot will appear on the bottom-right of the window. In this example we select the second snapshot, 1. To revert to this snapshot, click "Revert", a button above the snapshot content. Reverting to a previous snapshot creates the new snapshot, 4, with the same content as snapshot 0 and alters the file's content on disk to match the new head snapshot.

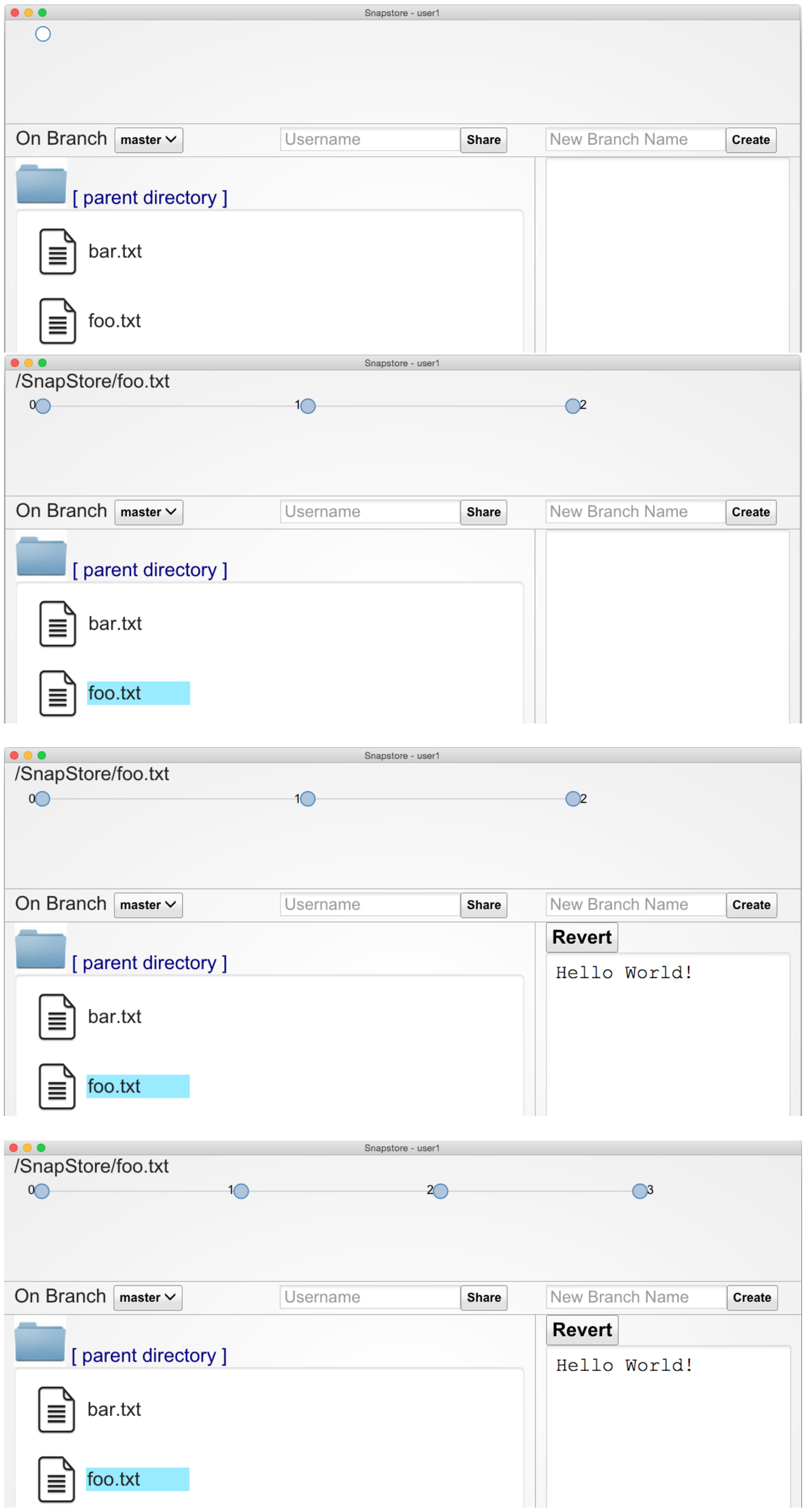


Figure 2-1: Locating a file's snapshot graph, choosing a snapshot, and reverting to it.

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, groups, and tags 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 all users.

Git users can sympathize with the hassle of having to commit, push and then pull data whenever changing computers. Snapstore solves this issue using upstreams. Imagine that a user has a desktop and a laptop computer. Both computers have a file called "foo.txt" on them. The user works on their desktop, making edits to "foo.txt" in the form of snapshots. When they open Snapstore on their laptop, the upstream pushes all of the snapshots made on the desktop to the laptop. Both computers now have the same version of "foo.txt". There is no push/pull model in Snapstore, data is automatically propagated.

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 computer.

2. Run the Snapstore server on that computer.

3. Point the Snapstore client to the desired computer by inputting its IP address into the code.

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 first saved in the local repository, whether there is an Internet connection or not.

If the local repository is connected to an upstream and there is an Internet connection, Snapstore will copy all of the data in the local repository to the upstream repository. If there is not an Internet connection, Snapstore will wait and push all new changes to the upstream repository when connection is restored.

The local repository does need to be connected to an upstream, however. You can use Snapstore locally, without an Internet connection, with only the local repository.

2.2 Advanced Concepts

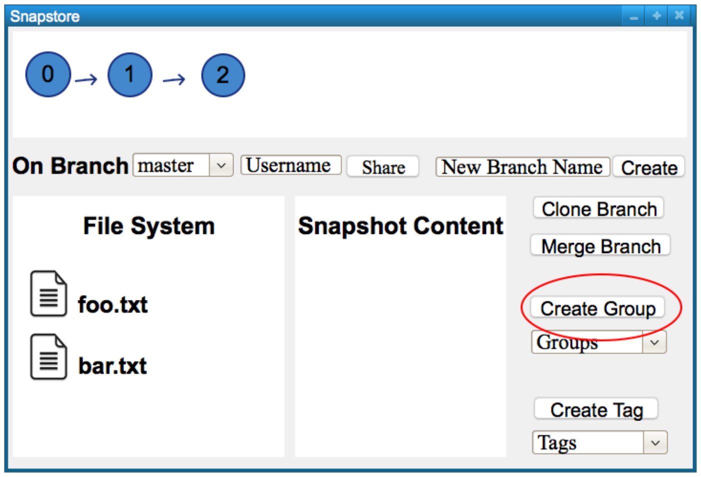
Snapstore's advanced features allow users to access the more powerful components of a version control system. They provide additional functionality that users might want when working on projects with complex version control requirements.

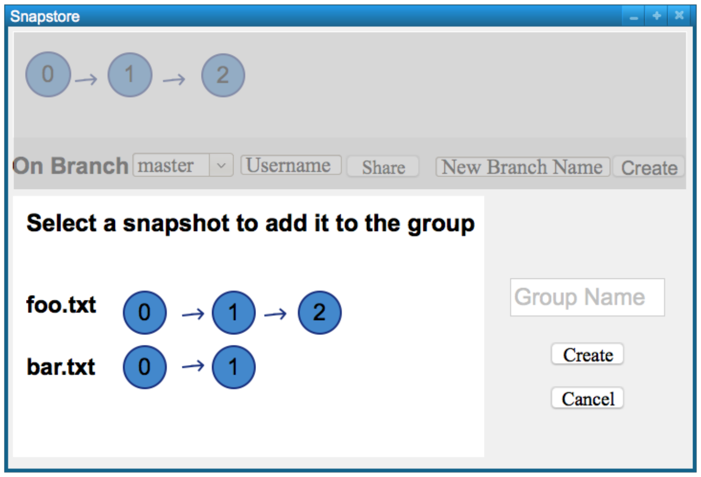
2.2.1 Groups

Groups allow users to designate a collection of snapshots as related. Users can place any number of snapshots in a group, even if they exist across multiple files. The user can then give this group a name and use this name to find the group later.

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". Once created, 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.

Figure 2-2 shows the process to create a group. Because this functionality is not yet implemented, it is a mock-up of the planned interface. First, click the "Create Group" button on the right. The snapshot graphs for each file in this branch (section 2.2.3) will then appear. Choose the snapshots you'd like to include in this group. Then, give the group a name. Finally, click "Create Group" to finish. Your group name will now appear in the "Groups" dropdown menu.





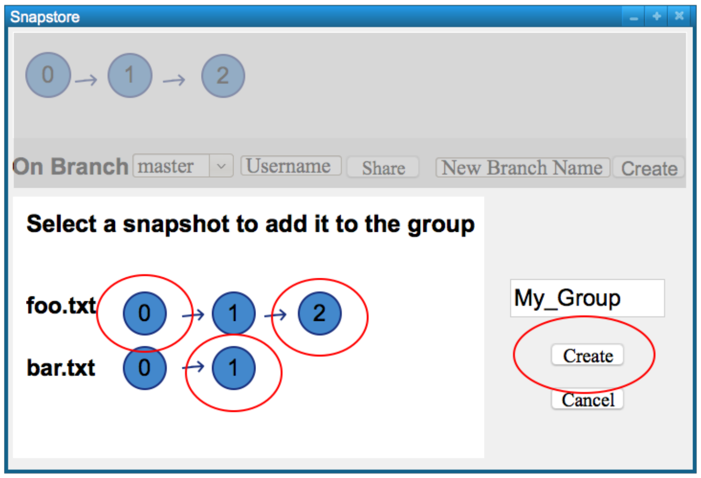


Figure 2-2: Creating a Group.

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 one snapshot per file. This is so that 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 their snapshot in that group.

One example of using tags is to describe a release in a software project. At the time of the 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 "Version 1.3".

2.2.3 Branches

Branches allow users to separate independent 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. The user can then begin adding data to this branch.

Branches are a collection of snapshots, groups, and tags. The collection of these concepts constitutes an independent line of development. Branches can be shared with other users, making them collaborators on that line of development. Branches can also be merged together in the local repository to combine lines of development.

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 collaborators the ability to try out experimental changes without affecting the main line [8].

Figure 2-3 shows how to create and switch between branches in Snapstore. To

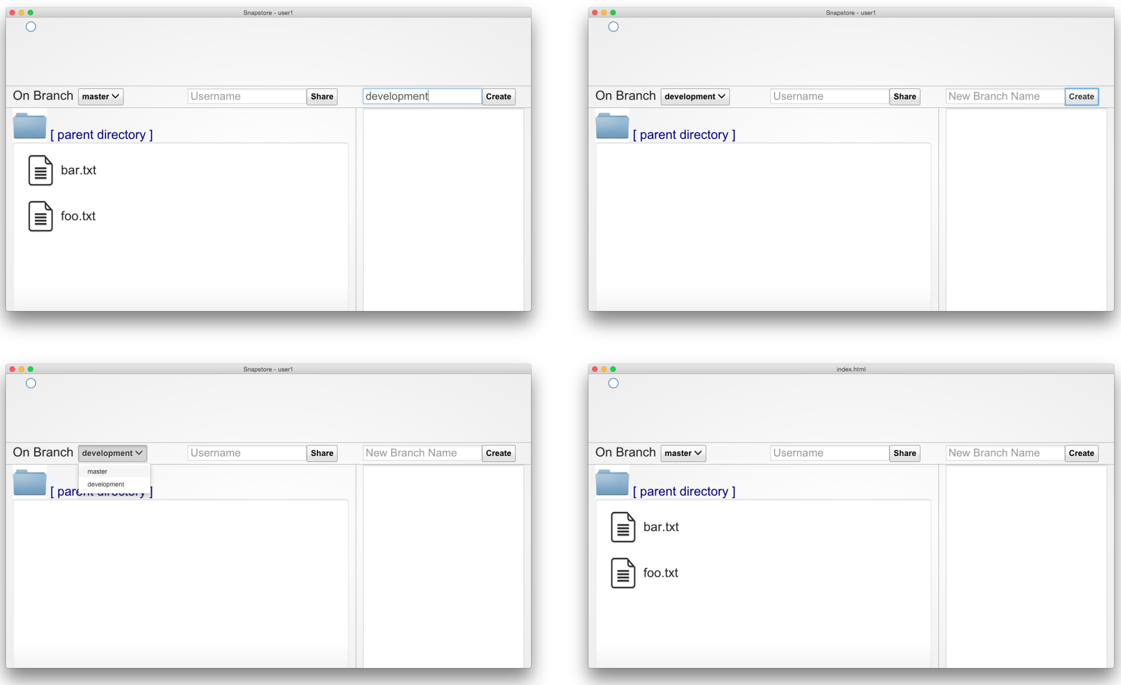


Figure 2-3: Creating a new branch "development" and switching back to the "master" branch.

create a branch, insert its name into the text box that says "New Branch Name". Here we make a new branch called "development". Then click create. This will switch you to the new branch, and you can start making snapshots, groups, and tags. To switch back to the "master" branch, click the dropdown branch name next to "On Branch". This will show you all of your branches. Click the branch you'd like to switch to, in this case "master", and you will be placed back on that branch.

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 with 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 to that user's local repository. On a shared branch, when any user makes changes, those changes are

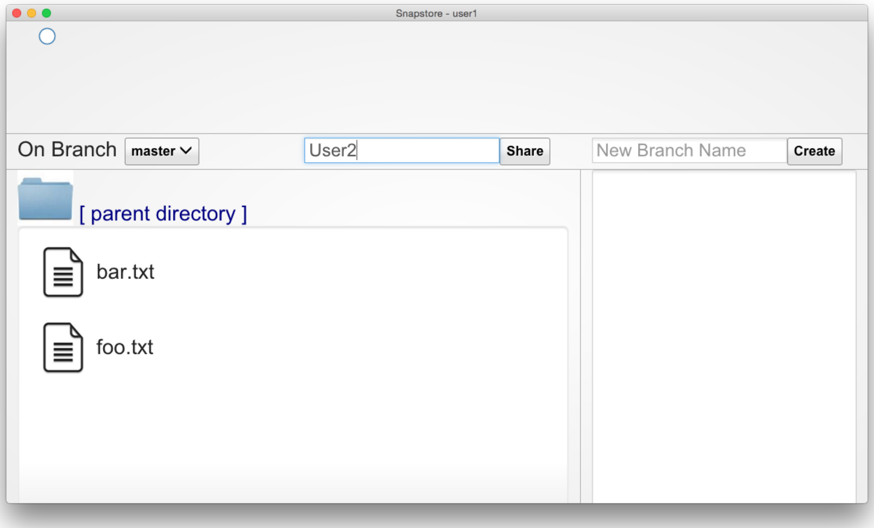


Figure 2-4: Sharing the "master" branch with user "User2".

immediately sent to the other user(s) who are collaborators on 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 a 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 2-4 shows how to share a branch with another user in Snapstore. Navigate to the branch you'd like to share, in this case we're sharing the "master" branch. Input the username of the user you'd like to share that branch with in the text box labeled "Username" next to the "Share" button. Then, click "Share".

Figure 2-5 illustrates an example of two users working on a shared branch. When two snapshots are created at the same time, one will reach the upstream first. When 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

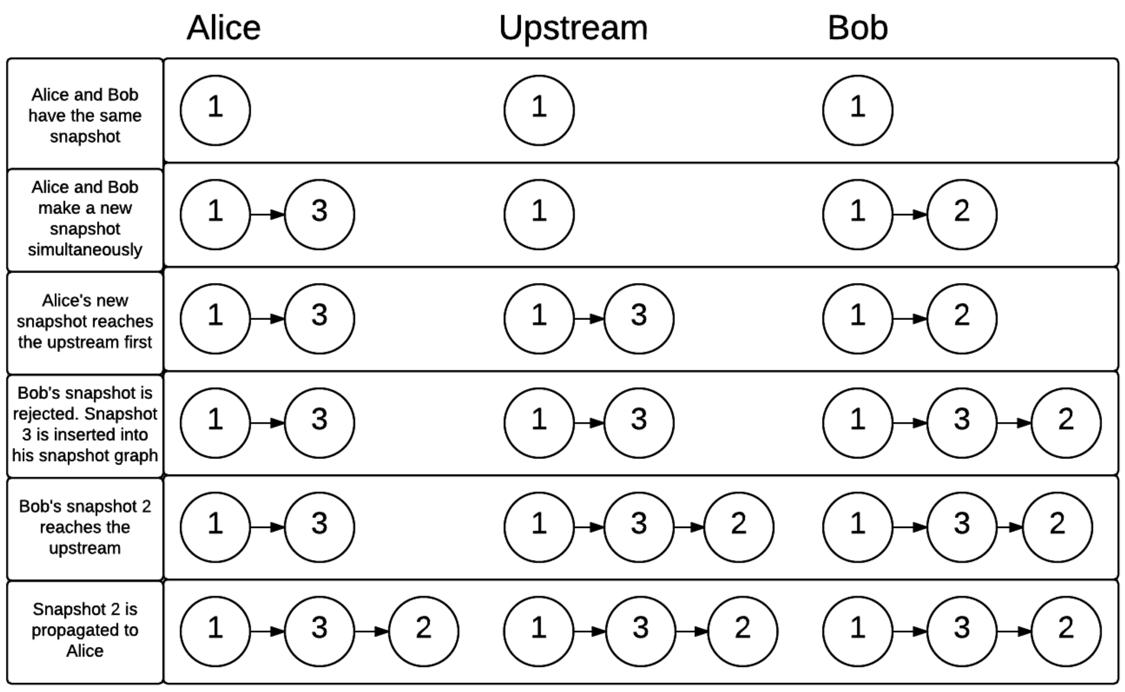

Alice
Upstream
Bob
Alice and Bob have the same snapshot
Alice and Bob make a new snapshot simultaneously
Alice's new snapshot reaches the upstream first
Bob's snapshot is rejected. Snapshot 3 is inserted into his snapshot graph
Bob's snapshot 2 reaches the upstream
Snapshot 2 is propagated to Alice
1
3
2

Figure 2-5: Conflicts being resolved on a shared branch.

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 is 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, 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 conflict snapshot is already saved in the local repository and any connected upstream repository; 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.

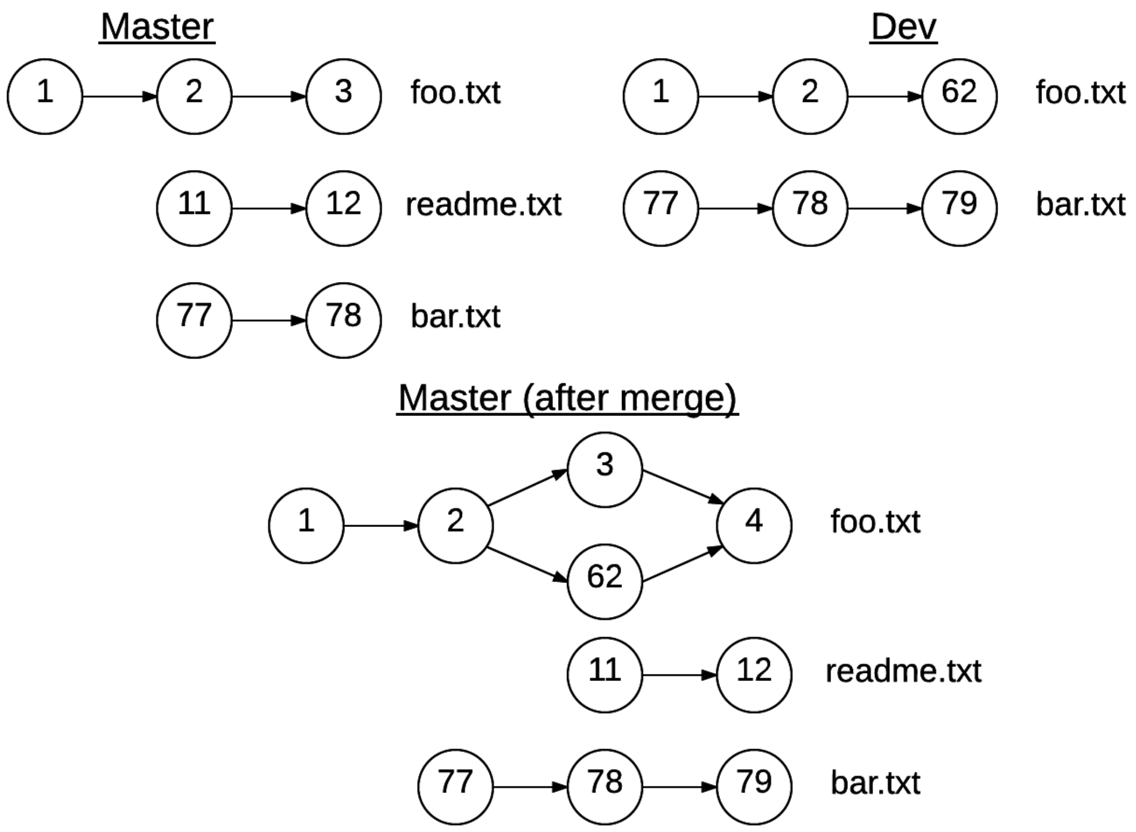


Figure 2-6: Merging two branches.

Figure 2-6 shows an overview of branch merging in Snapstore. Once snapshot graphs are identified as corresponding to the same file, their head snapshots are merged into a new snapshot, whose parents are the old head snapshots. In the figure, the head snapshots of "foo.txt", 3 and 62, are merged into a new merge snapshot, 4. The snapshot graph of any file that does not have a counterpart in the other branch, like "readme.txt", will be copied over to the merged branch. If there is a file where a fast forward is needed instead of a merge, like "bar.txt", the snapshots will be copied and the head will move forward, but there will be no merge.

Using Branches to Support Complex Workflows

Snapstore uses branches to allow users to partition their projects; they can then 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

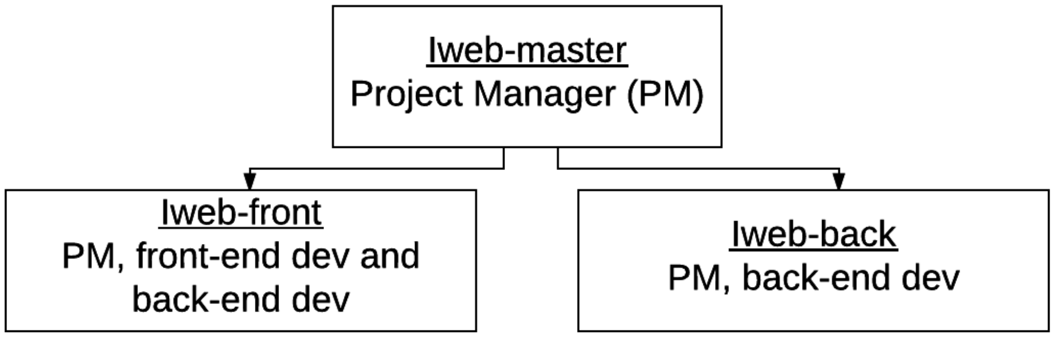


Figure 2-7: Iweb's branch arrangement.

the Linux project and its system of trusted lieutenants who vet incoming contributions before passing them on to the project owner [3].

Imagine that a website is being 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. To partition this project, the project manager makes two clones of this branch, "Iweb-front" and "Iweb-back". "Iweb-front" contains all of the front-end code files, and "Iweb-back" contains all of the back-end code files. She shares "Iweb-back" with the back-end developer, and she shares "Iweb-front" with both the front and back-end developer because the back-end developer needs access to some of the front-end files.

A graphical representation of these branches, along with who has access to them, is included in Figure 2-7.

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

This arrangement 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”.

Note that this arrangement could be further expanded. The front-end developer, for example, could clone parts of the “Iweb-front” branch into more branches. Using this approach, Snapstore uses can achieve complex hierarchical workflows.

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 [8] were used in the design of Snapstore. A purpose graph showing these purposes in the context of Snapstore is included in Figure 3-1, where each sub-purpose points to its parent purpose. A purpose is a sub-purpose of another purpose if it supplements it. The purposes are classified into five categories: data management, change management, collaboration, parallel development, and disconnected operation.

Data Management deals with the notion of backup. In case of failure, data needs to be stored persistently and be able to be retrieved. This purpose class addresses risks associated with development such as accidental deletion and incorrect saves, in addition to machine failure. The ability to track and untrack files gives the user control over what files will be persistently stored.

The second class, **change management**, deals with managing edits. Grouping changes allows the user to divide the history of a file or files in logical segments. Groups allow users to create segments in their projects and file histories. Tagging

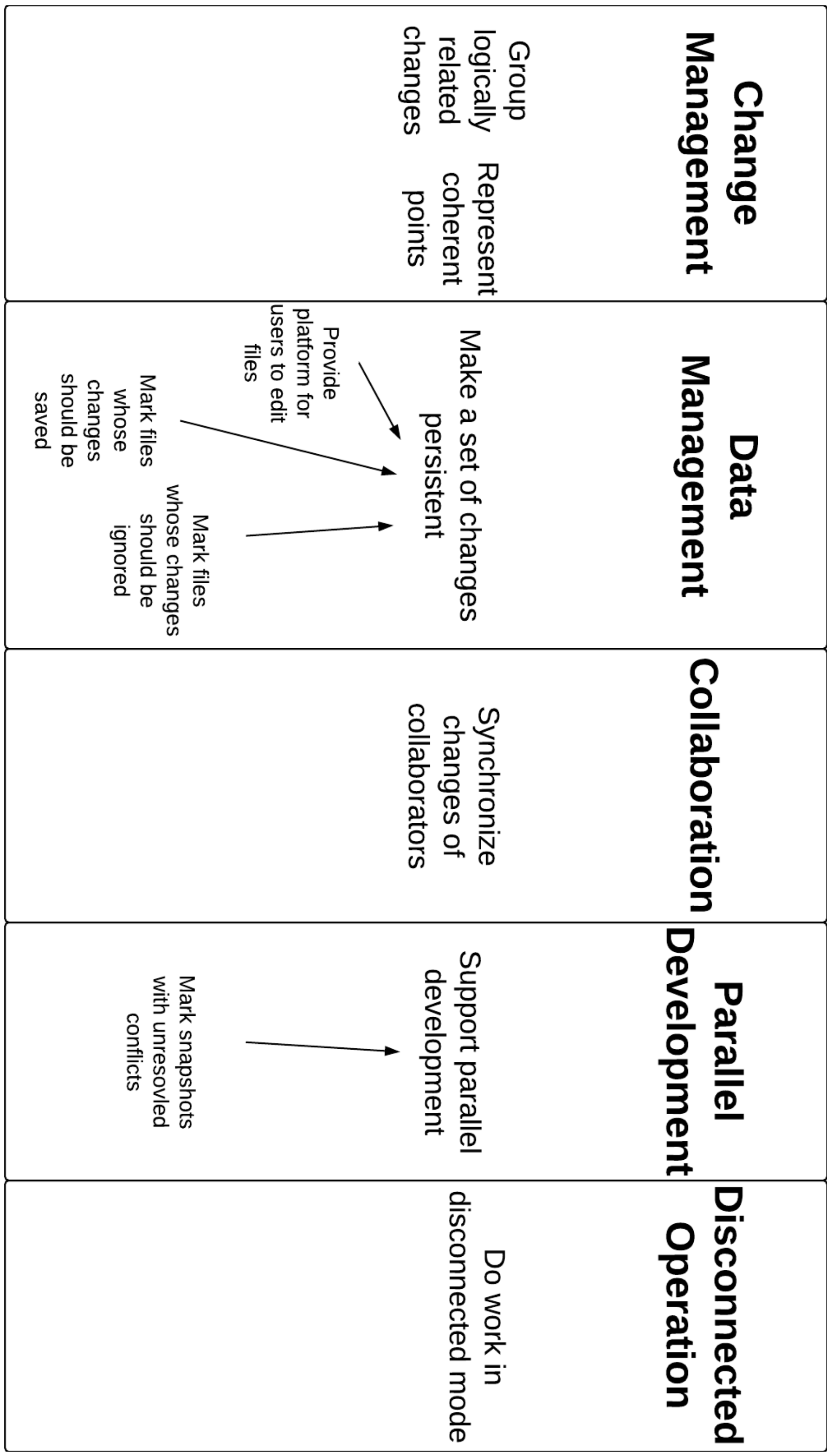

Change Management
Group logically related changes
Represent coherent points
Data Management
Make a set of changes persistent
Provide platform for users to edit files
Mark files whose changes should be saved
Mark files whose changes should be ignored
Collaboration
Synchronize changes of collaborators
Parallel Development
Support parallel development
Mark snapshots with unresovled conflicts
Disconnected Operation
Do work in disconnected mode

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

these groups as coherent points in development aid in administrative and managerial tasks associated with complex projects. These coherent points in the project can then be returned to by reverting the project to reflect the versions of files in that group.

Collaboration concerns a project shared by multiple users. Synchronizing the changes of collaborators on that system amalgamates their edits and distributes them in such a way that every collaborator can agree on the nature and ordering of the changes.

Parallel Development, the fourth class, deals with supporting parallel lines of development. Switching between parallel lines, as well as merging parallel lines are both needed to fully support parallel development. Merging must be done in a way that prevents conflicts when possible and makes them explicit when they cannot be avoided. 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 Operation**, allows operations to be performed in a disconnected mode. Any work that a user can do in the VCS should be possible in an offline setting or in a setting where the user has willingly disconnected from their collaborators.

3.2 Conceptual Model

Once the purposes of VCSs were defined, a conceptual model that addressed those purposes and that was aligned with conceptual design theory [4] was created. 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 [7], 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,

Purpose Class	**Concept**	**Motivating Purpose**
Data Management	Snapshot	Make a set of changes to a file persistent
	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
Change Management	Group	Group logically related changes together
	Tag	Represent and record coherent points in history
Collaboration	Upstream Repository	Synchronize changes of collaborators
Support Parallel Lines	Branch	Support parallel lines of work
	Conflict Snapshot	Mark snapshots with unresolved conflicts
Disconnected Operation	Local Repository	Perform operations in disconnected mode

Table 3.1: Concepts of Snapstore and their motivating purposes.

Purpose Class	**Concept**	**Operational Principle**
Data Management	Snapshot	Whenever a user saves a tracked file to disk from within the Snapstore Folder, a snapshot containing that file's contents is created and permanently stored in the snapshot graph for that file. The user can revert the file to a previous snapshot by navigating this snapshot graph and finding the correct snapshot. When the user reverts the file to this snapshot, the file's contents are modified to match the contents of the file at the point in time in which the snapshot was taken.
	Snapstore Folder	The user can see every file accessible to Snapstore by looking in the Snapstore folder. There, users can change the tracking status of those files and edit those files to create new snapshots.
	Tracked File	By making a file tracked, the user causes snapshots to be made of that file whenever edits of that file are saved to disk. These snapshots are stored in the local and upstream repository, and collaborators on the branch in which those snapshots were created can see them.
	Untracked File	The user can cause snapshots to stop being created for a file by making it an untracked file. Untracked files will not be saved in the local or upstream repository, and collaborators cannot see changes made to untracked files.
Change Management	Group	A user can place logically related snapshots in a group to increase the organization of their branch. The user can view this group by searching for its group name, and all collaborators on that branch can do the same.
	Tag	When a user places a tag on a group, it becomes findable by that tag's name in addition to the group name. This tag is also shared with all collaborators of that user. When the user reverts this tagged group, every file in that group is reverted to their snapshot in that group.
Collaboration	Upstream Repository	Whenever a user makes a change on a branch that other users have access to, that change is propagated by the upstream to those other users. Upstreams synchronize the changes made by collaborators so that each local repository has the same data.
Support Parallel Lines	Branch	When the user switches branches, Snapstore hides the old branch's data, shows them the current branch's data, and allows them to start adding data to the current branch. Branches separate data on independent lines of development for the user.
	Conflict Snapshot	When the merge of two snapshots results in a conflict, a conflict snapshot is created. This conflict snapshot shows the user where the conflict exists using conflict markers. Fixing the conflict creates a new snapshot.
Disconnected Operation	Local Repository	All saved changes made by the user are first stored persistently in the local repository, allowing them to work offline. When a network connection is restored, the local repository will push any data created while offline to the connected upstream.

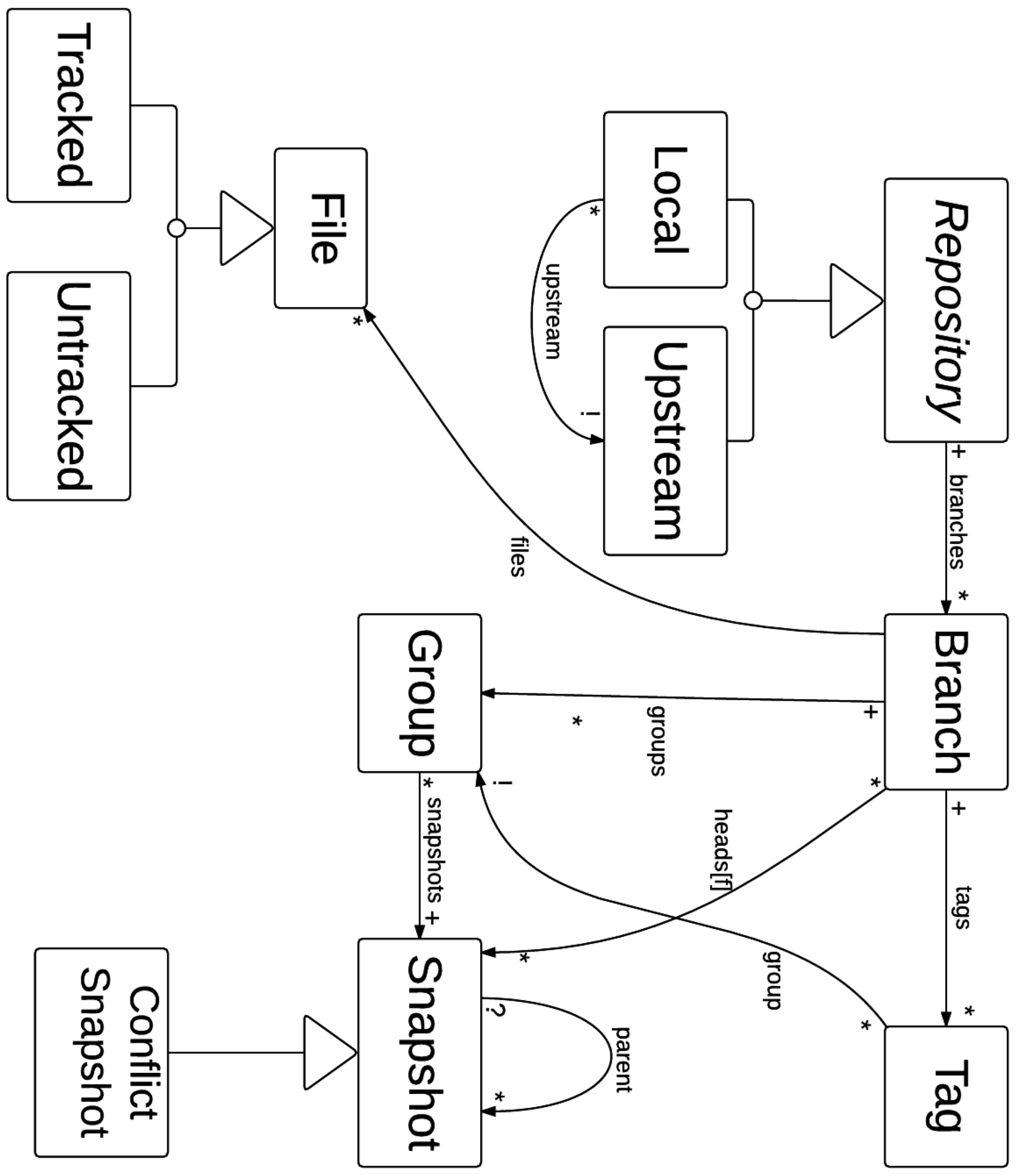


Figure 3-2: Concept Model of Snapstore.

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 disk.

The type of snapshot dictates the values of that snapshot's attributes. Create snapshots[1] have no parent. Update snapshots have a parent, a child, and content. Rename snapshots have a different file name 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.

The snapshots of a file are related by the graph they create with their 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.

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 another client's snapshot is inserted in the graph. This can occur when two users are on a shared branch and they need to share snapshots. However, even if snapshots are inserted into a user's snapshot graph, the user's ordering of locally made snapshots stays intact.

Any file, identified by its snapshot graph, can either be tracked or untracked. Edits made to untracked files will not result in the creation of new snapshots.

3.2.2 Grouping Changes — Group

A *group* is a collection of logically related snapshots, identified by a group name. 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

[1]Create snapshots are a type of snapshot. The identifier "create" is not an action. Rather, it refers to the fact that this snapshot was made when a file was first created. Create snapshots represent the *root* of a file's snapshot graph.

development strategy.

Groups are an attribute of a specific branch. Even if two groups contain the same snapshots across different branches, those groups are different because they exist on different lines of development.

3.2.3 Recording Coherent Points — Tag

The notion of a *tag* allows users to label logical milestones in their work. They describe a group 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 this definition may differ from project to project [8].

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

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

3.2.4 Support Parallel Lines — Branch

In Snapstore, the *branch* supports parallel and independent lines of development. These branches are completely separate from each other and facilitate the partitioning of data. The branch houses three of the other main concepts in Snapstore: snapshots, groups, and tags. These three concepts together constitute a line of development, and so the branch is the conceptual representation of that line.

Branches can be shared between multiple users. Users on a shared branch can make changes to the snapshots, groups, and tags of that branch, and the other collaborators will receive those changes. We have opted to use a last-write-wins approach 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 its 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 highly connected environment of today's computing, making that many offline edits is typically done by choice. Second, if the user knows that offline edits will be an issue, Snapstore allows them to create a separate branch for highly disconnected development. Users can create a separate branch they can work on offline, and they can merge back into the main branch when they're finished.

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

3.2.5 Synchronize Changes of Collaborators — Upstream Repository

Snapstore uses a centralized data storage system called an *upstream repository*, or upstream, to synchronize the data of collaborators. When multiple users share a branch, their data is shared via the upstream, synchronizing all local repositories.

When one collaborator makes a change at the branch level (branches, snapshots, groups, tags), it is reflected in the upstream. The upstream then searches for other collaborators on that branch and pushes the change down to them.

Every local repository can have one upstream, though it does not have to be the default Snapstore upstream. This allows the user to choose the location through which their data passes.

3.2.6 Disconnected Operation — Local Repository

The ability to leverage the benefits of a VCS without needing an Internet or network connection is enabled by the *local repository.* The local repository holds all of the branch, snapshot, group, and tag data for a user.

When any data is saved by the user, it is first saved to the local repository, whether or not there is network connection. This allows users to operate Snapstore offline, with all functionality except sharing. When a network connection is restored after a period of disconnected development, all of the local data created in the interim is pushed to the upstream.

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 should represent was the first design decision we encountered. Either a snapshot could represent the state of a file, or it could represent the state of every file in a branch.

The first reason we decided to make the snapshot describe the state of 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.

Another reason for this decision was the necessity of the group concept. In many VCSs, such as Git, saving changes couples together the act of saving changes with the act of grouping changes (a git commit), resulting in an overloaded concept [4]. If a snapshot represented the state of every file in a branch, then it could also be a group. We designed the snapshot to represent the state of a single file to conceptually separate the saving of changes with the grouping of those changes.

The Upstream

Different VCSs and file syncing systems have different storage models. Git, for example, uses 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. Centralized VCSs are easier to learn and use [1], but they require an Internet connection for all operations other than file edits and don't allow users to share directly with each other. Decentralized VCSs do allow users to work offline and share with each other, but they also have more space requirements because the version history for every file exists on every machine.

Snapstore uses a hybrid centralized/decentralized upstream model. On one hand, it is centralized because all collaboration takes place via the upstream repository. Any data a user wants to collaborate on with another user must go through the upstream first. On the other hand, Snapstore is also decentralized because users have a local repository, where actions can be made without a network connection. Snapstore users have all of their data on their own machine, just like in a decentralized VCS. Users can work offline, without checking out a central repository.

There are downsides to this hybrid model. Because Snapstore local repositories can only have one upstream, shared data will always be controlled by a single, centralized entity. And, this upstream must always be online in order to facilitate collaboration. Users also cannot share directly with a subset of collaborators on a shared branch. They can, however, work around this limitation by creating a new branch with that subset of users.

Despite these downsides, we believe this hybrid model is a good balance between the centralized and decentralized models. The centralized characteristics make it easier to learn and use, and the decentralized characteristics make it more powerful.

File Names

Whether to make the file name a property of the file or the identifier for the file was an important decision for Snapstore. Git, for example, uses the file's name as its identifier. Because of this decision, renames to a file are sometimes processed as deleting that file and creating a new file, causing much consternation among users, especially novices [8].

We wanted to robustly 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 search the file's two snapshot graphs for a common ancestor and perform a three-way merge. This allows Snapstore to accurately handle renames and merges.

Chapter 4

Implementation

This chapter details the implementation of Snapstore. This includes data structures and the synchronization algorithm.

Snapstore was built to be cross-platform using Electron[1], and we used web sockets[2] for networking. Once a client is connected over a socket, Snapstore can constantly pull in and push out new events that come in from that user and from other users. Sockets provide the additional benefit of allowing us to group users. We used this functionality to make “rooms” of users that have read and write access to a specific branch, allowing us to push out changes on that branch.

4.1 Data Structures

4.1.1 Client

Each local repository on the client has its own Mongo[3] 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, using a simplified entity relationship notation, representing each data structure and its attributes is shown in Figure 4-1.

[1]http://electron.atom.io/
[2]www.socket.io
[3]https://www.mongodb.com/

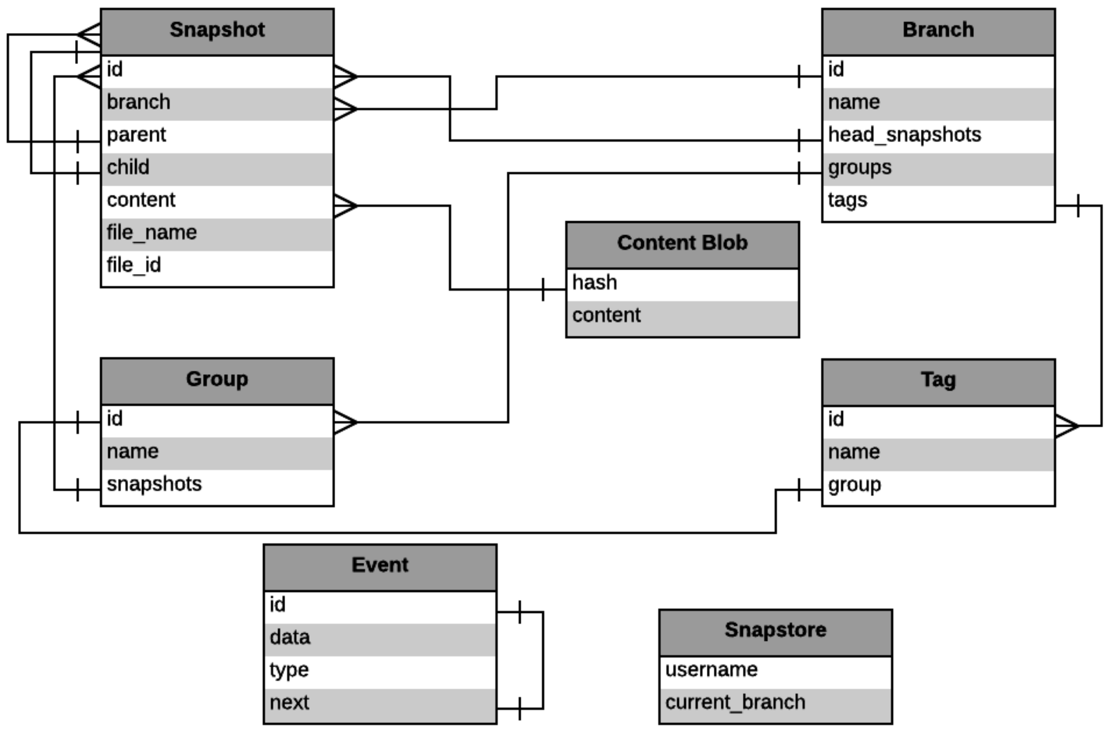


Figure 4-1: Snapstore client data model.

In this diagram (and in the diagram for the server's data structures), attributes in a data structure are connected to another attribute of a data structure if they are references to that attribute. Each relationship is either a one-to-one relationship – notated by a a straight line – or a one-to-many relationship – notated by a line with an end that splits into three lines. For example, a snapshot can have multiple parents, so there is a one-to-many relationship between the "parent" attribute and the "id" attribute. On the other hand, a snapshot can only have one child, so there is a one-to-one relationship between the "child" attribute and the "id" attribute.

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 a snapshot is created, it is added to the snapshot collection, and its branch updates its head snapshots to include the new snapshot, while removing its parent. 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.

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.

The event collection stores all of the snapshot, group, and tag events on the client that have not been confirmed by the server. As these events are confirmed, they are removed from the collection.

4.1.2 Upstream

On the upstream server, the snapshot, branch, group, tag, and blob data structures are the 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 shown in Figure 4-2.

The user model on the server is a mapping of users to branches to which they have access. When a user shares a branch with another user, that branch is added to this mapping for that user. The server uses this mapping to share data with the appropriate users. The "unsent_events" attribute in the user collection is used to persistently store any events from a user's collaborator that can't be pushed to the user because they are offline. This attribute is used in the data synchronization algorithm explained in section 4.2.

4.2 Keeping Data in Sync

In Snapstore, users can 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 who are collaborators on that branch, keeping all of their local

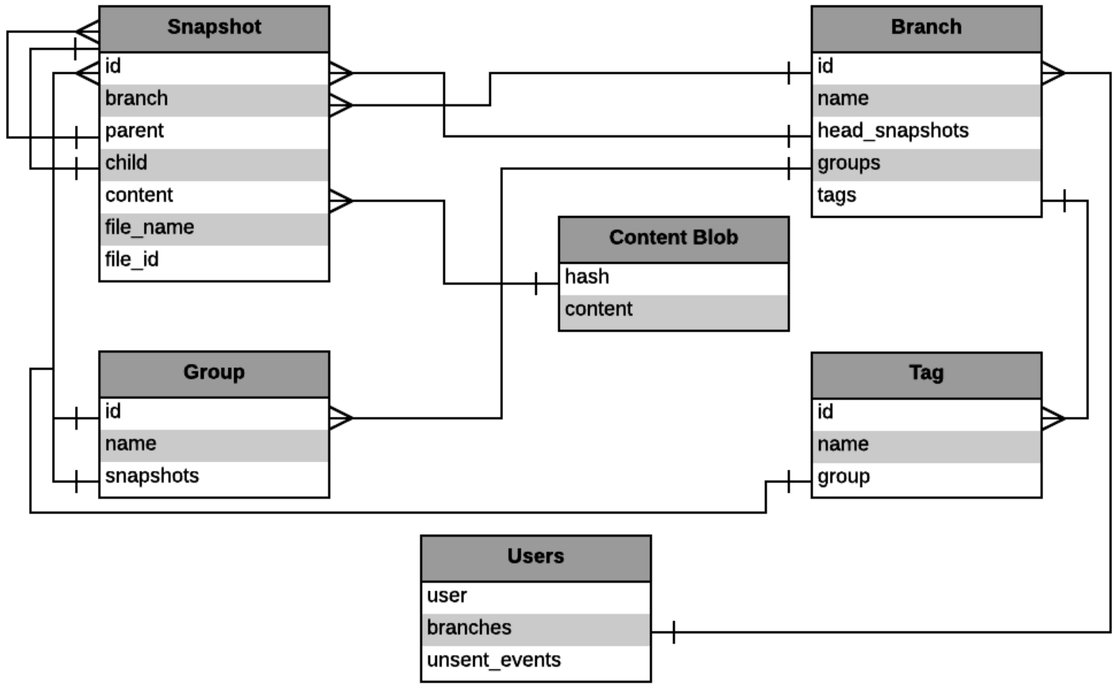


Figure 4-2: Snapstore server data model.

repositories consistent.

However, this workflow can be difficult to maintain. Multiple users can be making edits at the same time, increasing concurrency issues. If two snapshots are made at the same time, for example, the snapshot graph must be resolved and distributed to all clients. Also, a user can go offline and create snapshots, while their shared files are being written to by other, online users. Snapstore should be able to handle their reintroduction to the network without destroying the snapshot graph.

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 to the server in order. Otherwise, the server might, for example, try to create a group containing a snapshot that doesn't exist.

We take the approach that any event that reaches the upstream server and is confirmed should be regarded as correct for the entire system. This means that it should never be undone or modified, even if a conflicting event arrives. This also means that all clients can safely save every event that comes from the server. Using this idea, we designed a protocol algorithm for this process, called Distributed Event Synchronization Queue (DESQ).

Each client has their own ordering of events, or database operations, that are stored in a client queue until they are confirmed by the server. DESQ seeks to reach eventual consistency between these queues so that every client on a shared branch has the same data.

```
events ← pointer to Events collection
socket ← server socket connection
listener DATABASE-LISTENER
    on new database action event by user do
        events.append(event)
listener EVENTS-LISTENER
    on new event added to Events collection do
        Client-DESQ()
listener NETWORK-LISTENER
    on network connection with server established do
        socket.send("Check For New Events")
        Client-DESQ()
listener DESQ-RESPONSE
    on  socket.response(response, event, message) do:
        if message == "Confirmed" or "Duplicate" then
            events.remove(event)
        else if message == "Reject" then
            if event.type == Snapshot then         ▷ response is a list of snapshots
                Save response                      ▷ these snapshot events fix the conflict
                event.data.parent = response.head
        else                                       ▷ message == "New Event"
            if event.type == Snapshot and event.data.parent is not a head then
                return
            Save response
procedure CLIENT-DESQ
    for each event in events collection do
        if network connection is lost then
            break
        socket.send(event)
```

Figure 4-3: Client-DESQ pseudocode

Client

Client-DESQ, whose pseudocode is in Figure 4-3, can be triggered in one of two ways. Either a database action has occurred (listener defined on line 3), or a network connection with the server has been established (listener defined on line 9).

Once Client-DESQ is called, it cannot be called again until it has finished sending each event in the list. This is because Snapstore was implemented in JavaScript, which uses an single-threaded event loop. While Client-DESQ is running, events can be *queued up* to be added to the event list, but they won't actually be added. This effectively locks the event list. It is important to not add any new events to the event list while Client-DESQ is running because these events could be missed and not sent until another added event triggers the Events-Listener.

If Client-DESQ is started via network connection, the client first queries the server to see if there are any new events the client needs (line 11). If there are, the server sends them back as a "New Event", and the client saves them (line 24).

Once Client-DESQ begins, it sends each event in the event list to the server in order (line 29). Because this communication is happening on a socket over TCP, these events are guaranteed to arrive in order, the same order they were created in the client.

If an event is confirmed by the server, or if it is a duplicate event, the client removes this event from their event collection (lines 15-16).

If an event is rejected by the server, the client responds in one of two ways. If the rejected event is a snapshot event, the client saves the snapshots that will fix the conflict and updates the event snapshot's parent to be the head (lines 18-20). If the rejected event is a group or a tag event, nothing needs to be done; the event stays in its current location in the event collection.

In Git terms, if there is a conflict in a snapshot graph, the graph rebases and Snapstore tries to push that snapshot again. 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.

During Client-DESQ, if the network connection is lost, the process stops (lines 27 and 28). No events are discarded or lost when this happens, but the process terminates so that it can begin anew once a connection is restored. The event is left in its same location in the list, maintaining the event list's order.

A non-obvious statement appears on line 22, and it handles the following situation. A client, Alice, sends a snapshot to the server at the same time as Bob. Alice's snapshot is confirmed and sent to Bob. Bob's snapshot is then rejected. Alice's snapshot will then be sent to Bob twice, once by the "New Event" response, and once by the "Reject" response. To mitigate this, we ignore any snapshot "New Event" response whose parent is not a head on the client. In this case, that head is Bob's rejected snapshot. The client discards any events from the server that cause a conflict on the client because the client knows that a resolution to that conflict must be on its way.

This scenario can be understood by looking at Figure 2-5 in section 2.2.3. Once Alice's snapshot 3 is confirmed by the sever, it will be sent to Bob as a "New Event". However, Bob will discard this snapshot because its parent, 1, is not the head snapshot on Bob's client, 2. The eliminates Bob saving snapshot 3 twice; he only saves it once the "Reject" response arrives.

Server

Server-DESQ, whose pseudocode is in Figure 4-4, begins on the server when it receives an event over the socket from a client. If this event is the message "Check For New Events", the server looks in its storage of unsent events for that user and sends them (lines 5-9). If the event is not "Check For New Events", it must be an snapshot, group, or tag event.

If the server has already seen the event, it is flagged as a duplicate (lines 10-11) and returned to the client.

An event is confirmed if it can be applied to the server's database without triggered any causality issues (discussed in the following paragraph) in the data. When an event is confirmed, it is first returned to the client who sent it. After that, the event must be propagated to all collaborators of the event's creator. The server finds all users that have access to that event's branch (line 31). For each of those users, they are either online and can be forwarded the event (line 32-33), or they are offline and the event must be stored for them (line 34-35). The event is stored in the user collection on the server. Any collaborator that receives this event can apply it to their local repository because it is a confirmed event coming from the server.

Events are rejected if they cause a conflict in causality on the server. The logic which detects this causality conflict for the group event, the tag event, and the snapshot event (lines 13, 18, and 23, respectively) make up a strict ordering of events for Snapstore. This allows Snapstore to use the "happened before" relationship between system events [5]. Any event that requires that another, unseen or incorrectly ordered, event "happened before" it is rejected. If this rejected event is a group or a tag, only a rejection message is sent because the client will eventually send the snapshots or group that will fix the conflict (lines 16 and 21). If the rejected event is a snapshot, other snapshots that will fix the conflict are returned to the client (lines 26-27).

```
procedure SERVER-DESQ
    socket ← client socket connection
    eventMap ← persistent storage of unsent events
    on socket.receive(event) do:
        if event == "Check For New Events" then
            newEvents ← eventMap[socket.user]
            if newEvents.size() != 0 then
                socket.send(newEvents, "New Event")
                eventMap[socket.user] = []
        else if event.data.id in database then
            socket.send(event, "Duplicate")
        else if event.type == Group then
            if event.data.snapshots in database then
                Confirm-Event(event, socket, eventMap)
            else
                socket.send((), event, "Reject")
        else if event.type == Tag then
            if event.data.group in database then
                Confirm-Event(event, socket, eventMap)
            else
                socket.send((), event, "Reject")
        else                                                ▷ event.type == Snapshot
            if event.data.parent is head snapshot then
                Confirm-Event(event, socket, eventMap)
            else
                conflictSnapshots ← All snapshots between event.parent and head
                socket.send(conflictSnapshots, event,"Reject")
procedure CONFIRM-EVENT(event, socket, eventMap)
    socket.send(event, "Confirm")
    save event in database
    for each user with access to event.branch do
        if user is connected then
            socket(user).send((), event, "New Event")
        else
            eventMap[user].append(event)
```

Figure 4-4: Server-DESQ pseudocode

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 section 7.2.

5.1 Advantages

Easy to Get Started

Snapstore fulfills its goal of being easy to setup and use right away for a new user. Users log in and immediately have access to its more basic functionality. Snapshots of their files are automatically created, and everything is being saved to a local and an upstream repository. No configuration or learning of Snapstore is required for any of these benefits.

Automatic Saving

Because snapshots are automatically taken, there's no need to manually save your work. Instead of having to stop and save your work from the Snapstore application, simply using a typical hot-key for save in your editor will create a snapshot.

Working with Branches

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.

Branches also mesh well with the user's current knowledge of their filesystem; switching branches also switches the files in their Snapstore folder. There's no extra idea for the user to grasp when changing between independent lines.

Reversion

Reverting to a previous snapshot is another, easy one-click operation. It edits the user's filesystem so that their computer and Snapstore are always in agreement about the contents of files. Furthermore, the creation of a new snapshot for a file reversion is an easy concept to understand as a new user. The operational principle behind a snapshot holds: whenever an edit to a file is saved to disk, a new snapshot is created.

File and Snapshot Graph 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 having all the functionality of Snapstore in one place is convenient.

Furthermore, the intuitive presentation of the snapshot graph makes it easy to navigate for the user. Although this graph may be cluttered (an issue explored in section 5.2), the graph is intuitive and the relationship between snapshots (nodes) in the graph makes sense.

5.2 Issues

Snapshot Graph Search

The frequency with which snapshots are taken results in a cluttered snapshot graph. The snapshot graph loses meaning when there are too many snapshots to navigate it effectively, and it makes searching through the snapshot graph inefficient. This is not ideal for making use of Snapstore's most fundamental concept, the snapshot.

Cloned Branch Snapshot Selection

During the cloning of a branch, selecting which snapshots to copy to the clone is difficult for the same reason: there are too many snapshots. Choosing subsets from a snapshot graph is difficult, even though choosing the entire snapshot graph would not be. Though we don't know if users will, in fact, want to clone subsets of snapshot graphs, Snapstore does not currently allow them to do so efficiently.

Operations for Each File in Interface

For each file in the UI, there is both a file icon and a file name, but the function that clicking the file name performs is not clear. It opens the snapshot graph for that file. There are no affordances for this functionality, and it is quite possible that a new user would never be able to figure it out.

Lack of Native Operating System Support

Snapstore, unlike Dropbox, does not have any support for native applications such as being able to used from the native file manager, for example. This lack of support is annoying as a user trying to access Snapstore's functionality while completing other tasks.

A user could be exploring their filesystem from their task manager and want to observe a file's snapshot graph. However, they would have to open up Snapstore to do so. This inefficiency is not attractive to the prospective Snapstore user.

5.2.1 Conclusions

Snapstore does well at its two goals. First, it is easy to use for new users; with its opt-in complexity strategy, it's easy for users to get started right away with simpler, file syncing tasks. Second, it delivers an internal conceptual model that is easy for the user to understand.

Snapstore does have room for improvement. These improvements deal mostly with the presentation of Snapstore's concepts to the user and with the interface in general. Snapstore needs work refining the interface to make the concepts easier to navigate. This is a crucial step in Snapstore's development because without it, the underlying conceptual model of Snapstore cannot become the mental model of the end user. It is important, though, throughout this redesign, that the conceptual model is never compromised for the sake of the interface. The conceptual design and the interface design must complement each other in presenting the concepts to the user.

Chapter 6

Related Work

6.1 Other Applications of the Theory of Conceptual Design

In [8], 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 [10], 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. The author 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 to design a new version control system from scratch. We did, however, use the purposes of version control enumerated in [7] to guide the design of Snapstore.

6.2 Version Control and File Syncing

We also studied the file syncing and version control system spaces as case studies. 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

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.

Merging is also not well supported by many file syncing tools. Google Drive preserves every edit made on a shared document through a process called operational transformation [9]. When a conflict arises, a newline is inserted. This is fine for some text documents, but it can break code documents. Dropbox does not allow files to be merged at all. With these systems, 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 file name 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. Files in Snapstore are identified with a unique id, not a file name. Because of this, Snapstore can track renames and maintain file history more robustly.

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 concepts is simplified to just the branch in Snapstore.

Similar to file syncing systems, the file name in a version control system is often a unique identifier. In Git, if a user renames a file without using the “git” command, Git sometimes recognizes that as deleting a 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 seamless history by using the file id as an identifier. 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 using the upstream repository.

Chapter 7

Future Work

Snapstore is currently a minimum viable product as a file syncing system. However, there is more work that needs to be done before it is fit to be a quality version control system. These improvements fall into two general categories: implementation and user study.

7.1 Implementation

7.1.1 Functionality

The first area of functionality that needs to be created is the grouping and tagging of snapshots. These data structures must be maintained by their branch, as snapshots are. Any changes must be propagated to all collaborators on their branch.

The next feature that needs to be implemented is the ability to clone and merge branches. One potential interface problem we foresee 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 events collection must also be implemented for Snapstore to follow the algorithm outlined in section 4.2. Currently, because only snapshots are implemented (not groups or tags), the snapshot graphs act as the event list. Instead of leaving the list when they are confirmed, a boolean flag is set in the snapshot to designate it as confirmed by the server. To allow for group and tag events, an event collection must be implemented as the medium by which all data is sent to and received from the server.

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. No principles of good interface design have been consciously applied, and the interface 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 [10]. It is important that future interface work on Snapstore not obscure the conceptual design that it is trying to show the user. It is the goal, after all, that this conceptual design becomes the user's mental model of Snapstore.

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 should include file-specific functionality like reversion and accessing a file's snapshot graph.

Snapstore would also benefit from another Dropbox feature, a menubar icon. 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 test how well participants like the functionality provided by Snapstore and the interface used to provide it. An interesting result of this study 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 could gather more quantitative data. We could measure the time it takes for participants to perform actions they are used to performing with their file syncing or version control system. With those numbers, we could 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? In *Proceedings of the 36th International Conference on Software Engineering*, pages 322–333, New York, NY, USA, 2014. ACM.

[2] F. P. Brooks. *The Mythical Man Month:Essays on Software Engineering.* Addison-Wesley Professional, 1995.

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

[4] Daniel Jackson. Towards a theory of conceptual design for software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2015, pages 282–296, New York, New York, USA, 2015. ACM.

[5] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, jul 1978.

[6] Donald Norman. *The Design of Everyday Things.* Basic Books, 1988.

[7] Santiago Perez De Rosso and Daniel Jackson. What's wrong with git?: A conceptual design analysis. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. ACM.

[8] Santiago Perez De Rosso and Daniel Jackson. Purposes, concepts, misfits, and a redesign of git. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, 2016.

[9] Yi Xu, Chengzheng Sun, and Mo Li. Achieving convergence in operational transformation: Conditions, mechanisms, and systems. In *Proceedings of the 17th ACM conference on Computer supported cooperative work and social computing*, CSCW 2014, pages 505–518, New York, NY, USA, 2014. ACM.

[10] Xiao Zhang. A conceptual design analysis of dropbox. 2014.