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 its parent purpose. The purposes are classified into five categories: data management, change management, collaboration, parallel development, and disconnected development.

Data Management deals with the notion of backup. In case of failure, this 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, not just machine failure. The ability to track and untrack files gives user control over what files will be persistently stored.

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 complex 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 group.

Collaboration concerns a system 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 Development**, 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 [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

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 user inspects a file, the contents of that file correspond to the contents of its head snapshot
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	Whenever a user makes a change on a branch that more than user has access to, that change is propagated by the upstream to all other collaborators on that branch
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
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.

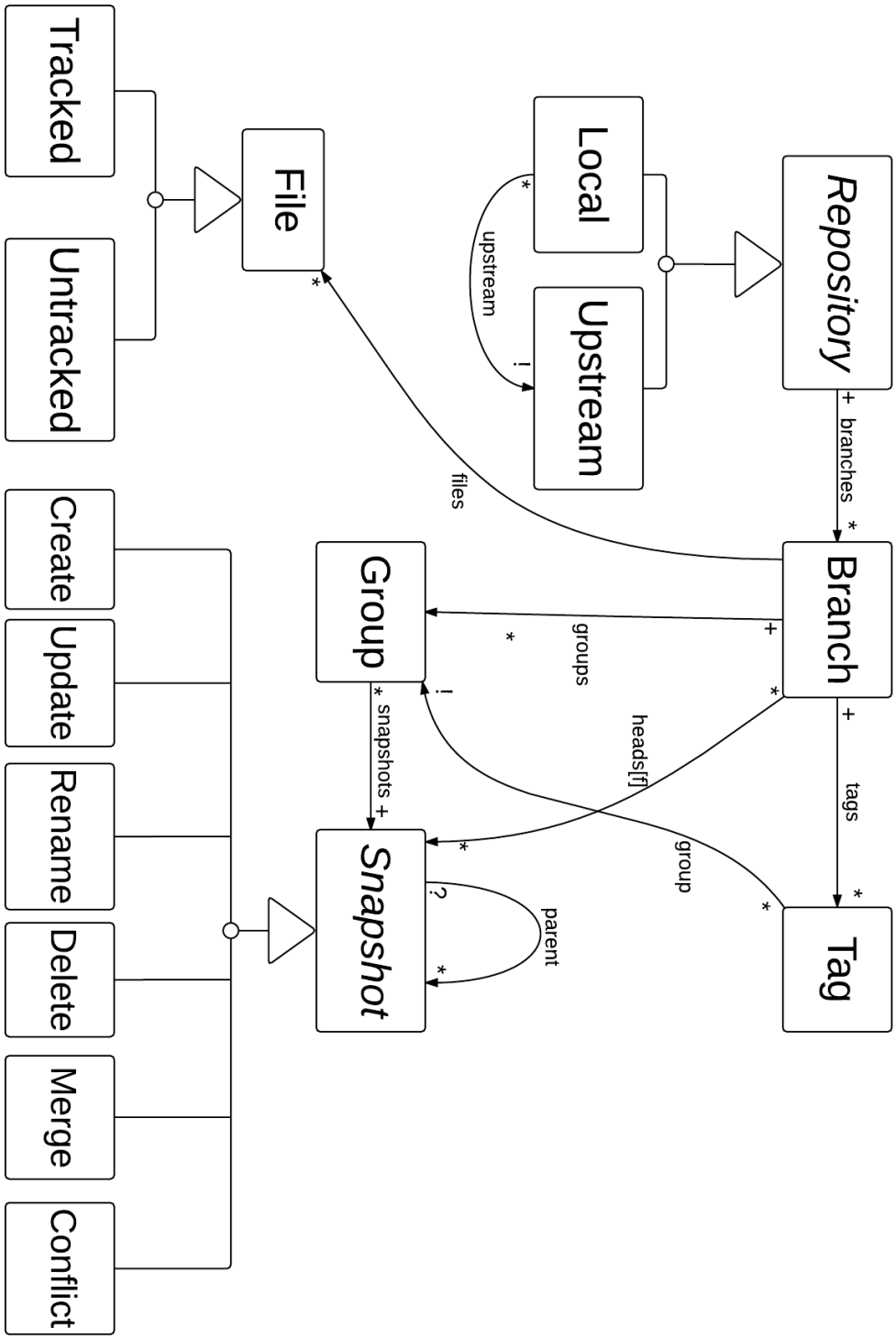

Figure 3-2: Concept Model of Snapstore.

recent snapshot made for that file and reflects the current content of that file on that machine.

The 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.

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. 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 create snapshots.

3.2.2 Grouping Changes — Group

A *group* is a collection of logically related snapshots along with 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 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.

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 may differ from project to project [3].

Tags will always describe groups that are perfectly vertical. A vertical group is one with at most one snapshot from any file. An example of this is tagging a group containing every head snapshot 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 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 consitute 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 recieve those changes.

Branches can also be merged together, synchronizing the parallel development. This 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 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 both local repositories.

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.

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 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.

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 separated the saving of changes with the grouping of those changes, so the snapshot represents the state of a single file.

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 syncing systems have different storage models. 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. Centralized VCSs are easier to learn and use [1], but they require an Internet connection and don't allow users to share directly with each other. Distributed 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 must take place via the upstream repository. Any data a user wants to collaborate on with another user must go through the upstream. 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. 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 branche 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 it's 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 [3].

We wanted to 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.