Chapter 4

Implementation

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

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

4.1 Data Structures

4.1.1 Client

Each local repository on the client has its own Mongo[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 representing each data structure and its attributes is shown in figure 4-1.

When a file is saved, the resulting snapshot must first decide what kind of snapshot

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

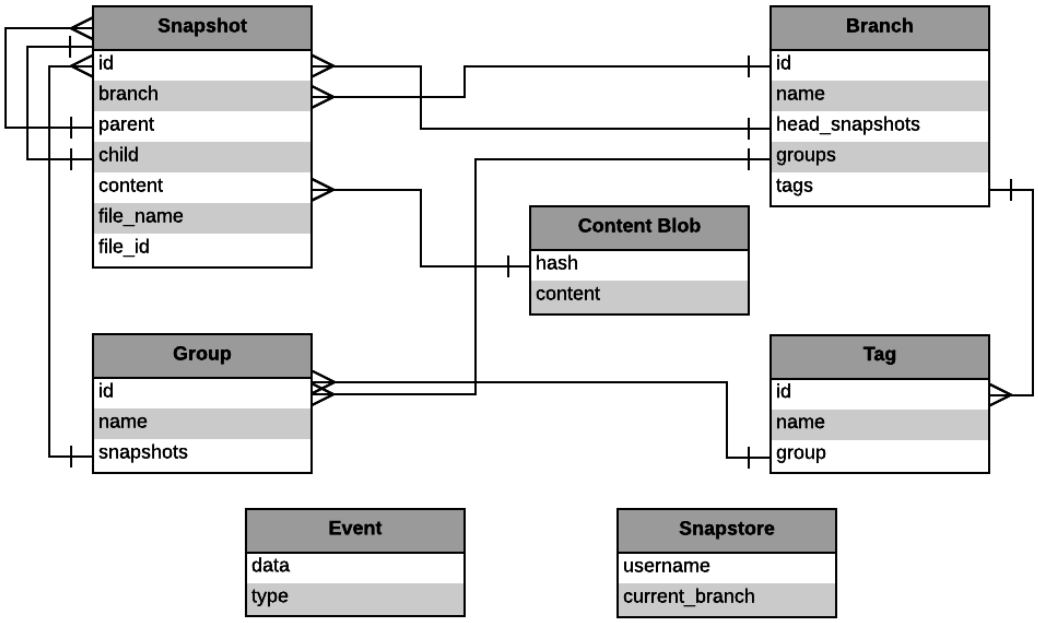

Figure 4-1: Snapstore client data model.

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 snapshots, groups, and tag events on the client that have not been confirmed by the server. As these events are confirmed,

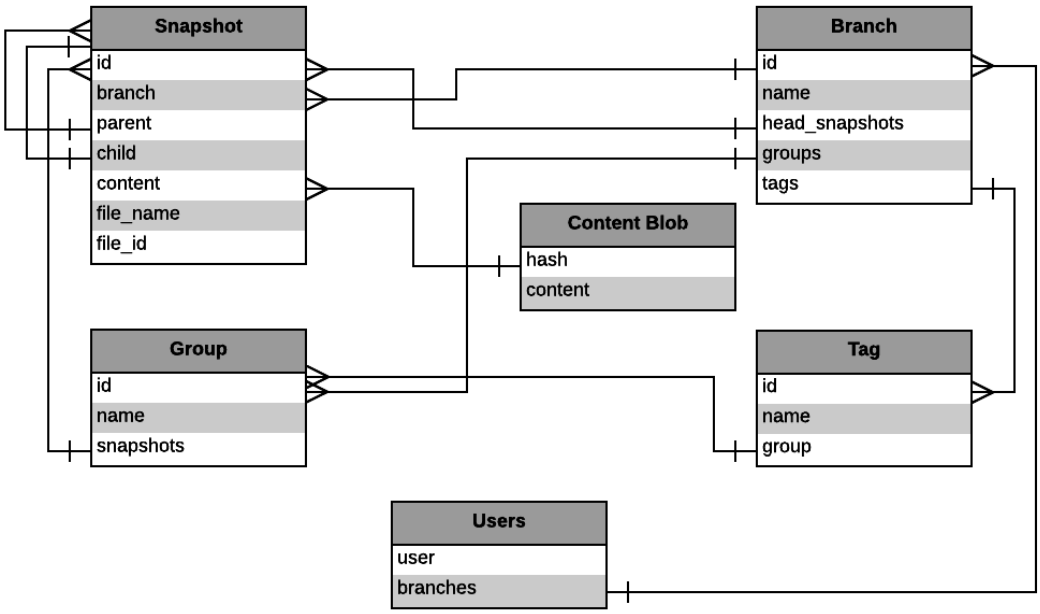

Figure 4-2: Snapstore server data model.

they are erased from the collection.

4.1.2 Upstream

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

4.2 User Interface

Pictures will go in here when the UI is done.

4.3 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 Snapstore folders 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, there needs to be a way to resolve the snapshot graph. Also, a user can go offline and create snapshots, while their shared snapshot graphs 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 in order to the server. Otherwise, the server might, for example, try to create a group containing a snapshot that doesn't exist.

We take the approach that any data that reaches the upstream server and is confirmed should be regarded as fact; it should never be undone. With this invariant, 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. The DESQ algorithm seeks to reach eventual consistency between these queues so that every client on a shared branch has the same data.

DESQ begins when an inconsistency is detected in the system. This can happen in two ways. First, if there is an event in the client's event queue, the algorithm will try to get that event confirmed by the server and shared with all appropriate users. Second, if a client connects to the server and detects that changes have been made, it will pull in those changes. Note that this protocol can proceed only when a network

Algorithm 1 DESQ

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

connection between the server and client is open. If they are closed, the events are queued until the network is available.

This algorithm pushes these events one at a time to the server repository. Events are confirmed if they can be applied to the server without causing any conflicts in the data. On the client, when an event is confirmed, it is removed from the event queue. Events also have a unique ID, so duplicate events can be caught and handled the same way.

When a client sends an event to the server and it is confirmed, the event must

be propagated to all other collaborators. The server will find all clients that have access to that event's branch and send it to them. Because this is a confirmed event coming from the server, the other clients can apply this event to their local repository without adding it to their event queue.

Snapshots are the only type of event that can cause a conflict, due to their inclusion in an ordered snapshot graph. All clients with access to this snapshot graph must agree on its order. The server will reject a client's snapshot event if the snapshot's snapshot graphs on the server and client are inconsistent. The rejected snapshot returns to the client with snapshots that will fix the inconsistencies.

In Git terms, if there is a conflict, the snapshot 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.

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