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]. 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/

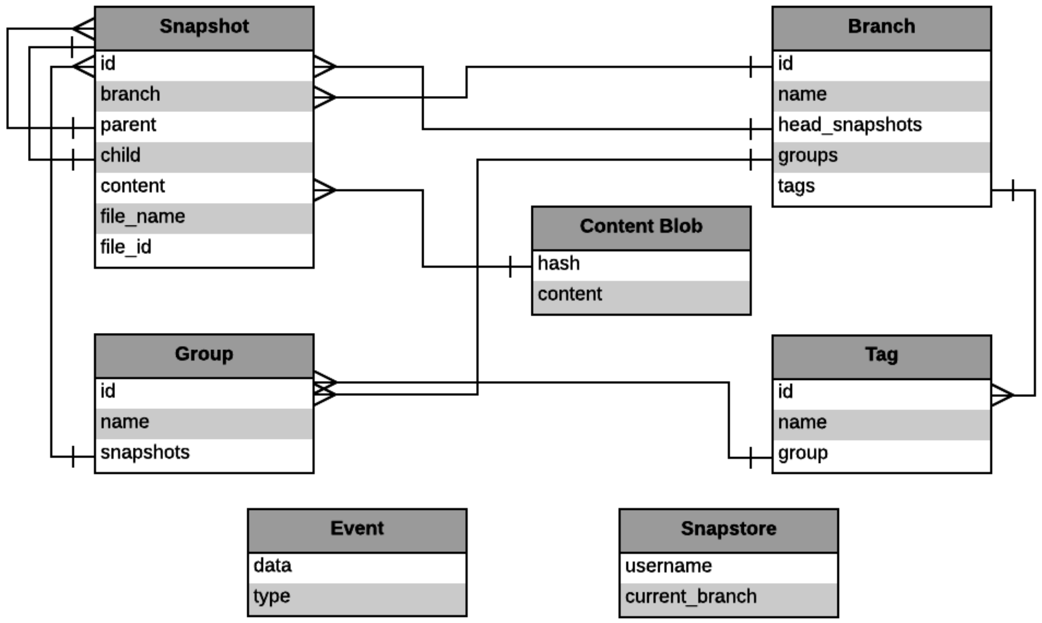

Figure 4-1: Snapstore client data model.

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,

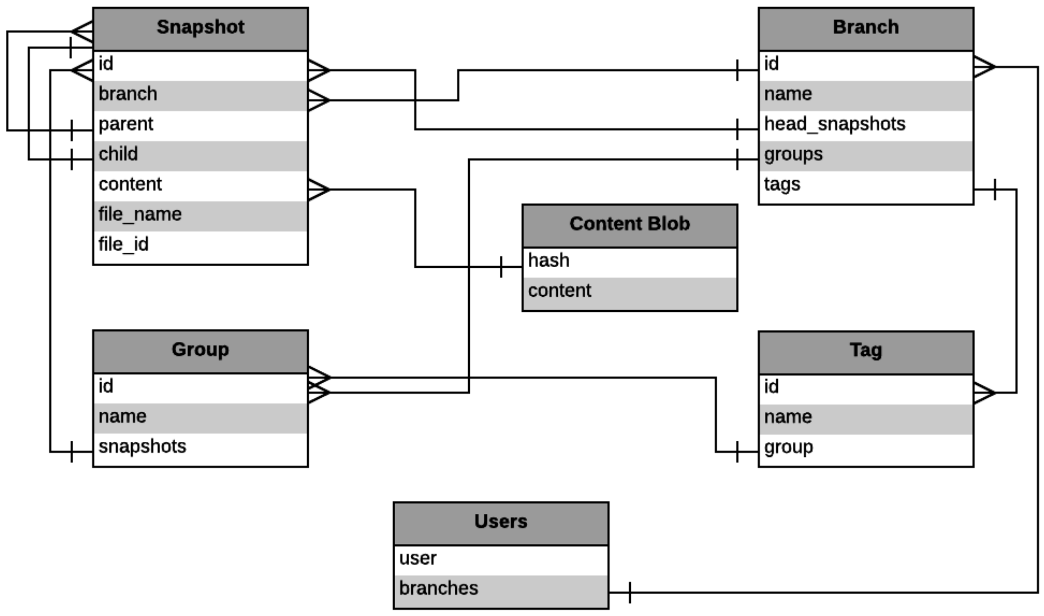

Figure 4-2: Snapstore server data model.

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 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.

Client

Client-DESQ begins when there is an event in the client's event queue. The client will send that event to the server (loop between line 4 and 6).

When the client receives a response from the server, it first saves whatever data the server sends (line 7) because the server only sends confirmed data. If the response is a confirmed or duplicate event, then the client can remove that event from their queue (line 9). If the event was rejected, the client must update the parent of the snapshot that caused the rejection (line 11). This snapshot event stays in the queue to be re-sent.

Server

Server-DESQ begins on the server when it receives an event from a client. If the server has already seen this event, it is flagged as a duplicate (line 6). The event is confirmed if it can be applied to the server without causing any conflicts in the data. This is the case for all group and tag events (line 10) and for snapshot events whose parent is a head snapshot (line 15).

Once the server confirms an event, the event must be propagated to all collaborators. The server finds all clients that have access to that event's branch and sends the event to them (line 10). Because this is a confirmed event coming from the server, the collaborators 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 (line 18). 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.

```
events $\leftarrow$ collection of Events
socket $\leftarrow$ server socket connection
Listener Database-Listener do
    on new database action event do
        Database-Listener.off()
        events.append(event)
        Client-DESQ
        Database-Listener.on()
    end on
end
Listener Network-Listener do
    on network connection with server established do
        Network-Listener.off()
        socket.send("Check Events")
        Client-DESQ
        Network-Listener.on()
    end on
end
procedure Client-DESQ
    if events.size()! = 0 then
        socket.send(events(0))
    on  socket.response(response, message) do:
        if message == "Confirm" or "Duplicate" then
            events.remove(events(0))
            if events.size() != 0 then
                socket.send(events(0))
            close
        if message == "Reject" then
            Save response
            events(0).data.parent = response(-1)          ▷ response(-1) is the head
            socket.send(events(0))
            close
        if message == "New Event" then
            if response.data.parentis not a head snapshot then
                close
            Save response
            close
    end on
```

Figure 4-3: Client-DESQ pseudocode

```
procedure Server-DESQ
    socket ← client socket connection
    eventMap ← persistent storage of unsent events
    on socket.receive(event) do:
        if event = "Check Events" then
            newEvents ← eventMap[socket.user]
            if newEvents.size()! = 0 then
                socket.send(newEvents, "New Event")
                eventMap[socket.user] = []
            close
        if event.data.id in database then
            socket.send(event, "Duplicate")
            close
        if event.type ! = snapshot then
            Confirm-Event(event, socket, eventStore)
            close
        else
            if event.data.parent is head snapshot then
                Confirm-Event(event, socket, eventStore)
                close
            else
                conflictSnapshots ← All snapshots between event.parent and head
                socket.send(conflictSnapshots,"Reject")
                close
    end on
procedure Confirm-Event(event, socket, eventStore)
    socket.send(event, "Confirm")
    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)
    close
```

Figure 4-4: Server-DESQ pseudocode