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/

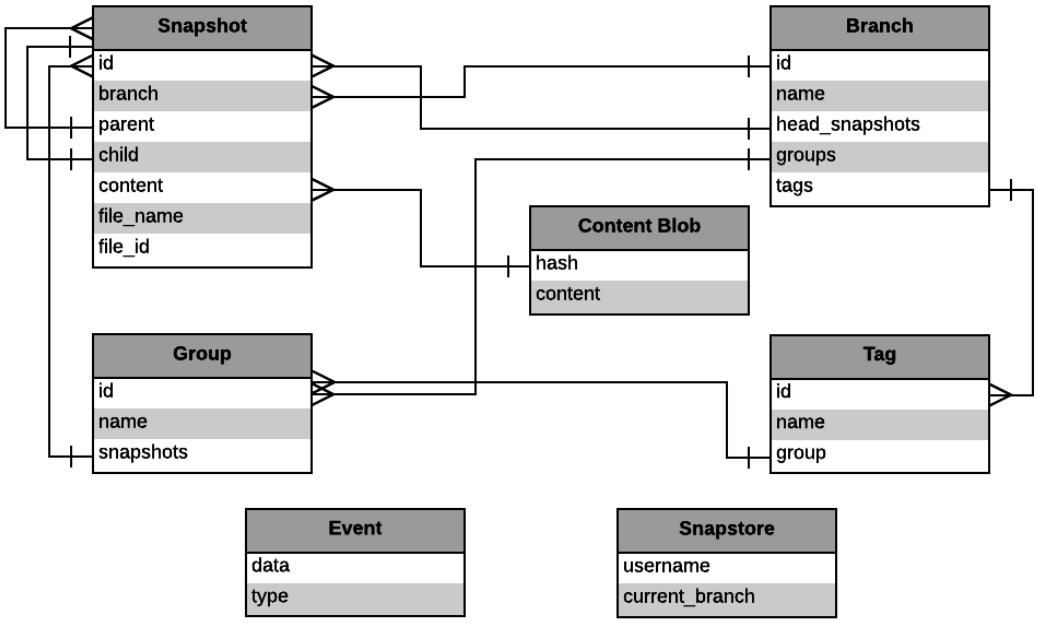

Figure 4-1: Snapstore client data model.

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,

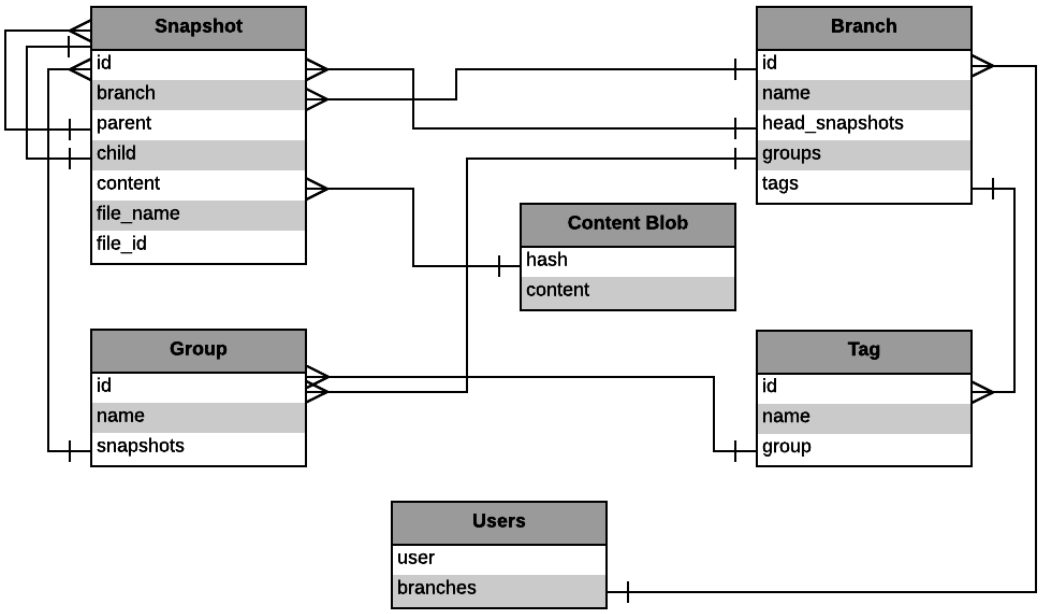

Figure 4-2: Snapstore server data model.

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 can be called 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 13). Once Client-DESQ is called, both listeners are turned off, and the algorithm cannot be called again until it has finished and both listeners are turned back on. This logic places an implicit lock on the algorithm.

If Client-DESQ is started via network connection, the client also queries the server

to see if there are any new events the client needs (line 17). If there are, the server sends them back as "New Events", and the client saves them (line 40).

Once Client-DESQ begins, it sends the first event in the queue to the server (line 25). For each event that it sends, it waits for a response for the server before sending another (lines 30 and 35). This level of blocking ensures that events are sent to the server in the order they were created on the client.

If an event is confirmed by the server, or if it is a duplicate event, the client removes this event from their queue and either stops sending events if it has none or sends the next event (lines 27-30).

If a snapshot event is rejected by the server, the client saves the snapshot events that will fix the conflict and updates the event snapshot's parent to be the head (lines 32-34). It then sends the updated snapshot event again (line 35).

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.

```
1: events ← collection of Events
2: socket ← server socket connection
3: Listener Database-Listener do
4:     on new database action event by user do
5:         events.append(event)
6:     end on
7: end
8: Listener Events-Listener do
9:     on new event added to events do
10:        Network-Listener.off()
11:        Events-Listener.off()
12:        Client-DESQ()
13:        Network-Listener.on()
14:        Events-Listener.on()
15:    end on
16: end
17: Listener Network-Listener do
18:    on network connection with server established do
19:        Network-Listener.off()
20:        Events-Listener.off()
21:        socket.send("Check Events")
22:        Client-DESQ()
23:        Network-Listener.on()
24:        Events-Listener.on()
25:    end on
26: end
27: procedure CLIENT-DESQ
28:    if events.size()! = 0 then
29:        socket.send(events(0))
30:    on  socket.response(response, message) do:
31:        if message == "Confirm" or "Duplicate" then
32:            events.remove(events(0))
33:            if events.size() != 0 then
34:                socket.send(events(0))
35:            close
36:        if message == "Reject" then
37:            Save response                    ▷ These snapshot events fix the conflict
38:            events(0).data.parent = response(-1)        ▷ response(-1) is the head
39:            socket.send(events(0))
40:            close
41:        if message == "New Event" then
42:            if response.data.parent is not a head snapshot then
43:                close
44:            Save response
45:            close
46:    end on
```


Figure 4-3: Client-DESQ pseudocode

```
1: procedure SERVER-DESQ
2:     socket ← client socket connection
3:     eventMap ← persistent storage of unsent events
4:     on socket.receive(event) do:
5:         if event = "Check Events" then
6:             newEvents ← eventMap[socket.user]
7:             if newEvents.size()! = 0 then
8:                 socket.send(newEvents, "New Event")
9:                 eventMap[socket.user] = []
10:            close
11:        if event.data.id in database then
12:            socket.send(event, "Duplicate")
13:            close
14:        if event.type ! = snapshot then
15:            Confirm-Event(event, socket, eventMap)
16:            close
17:        else
18:            if event.data.parent is head snapshot then
19:                Confirm-Event(event, socket, eventMap)
20:                close
21:            else
22:                conflictSnapshots ← All snapshots between event.parent and head
23:                socket.send(conflictSnapshots,"Reject")
24:                close
25:    end on
26: procedure CONFIRM-EVENT(event, socket, eventMap)
27:    socket.send(event, "Confirm")
28:    for each user with access to event.branch do
29:        if user is connected then
30:            socket(user).send(event, "New Event")
31:        else
32:            eventMap[user].append(event)
33:    close
```

Figure 4-4: Server-DESQ pseudocode