# 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 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 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/

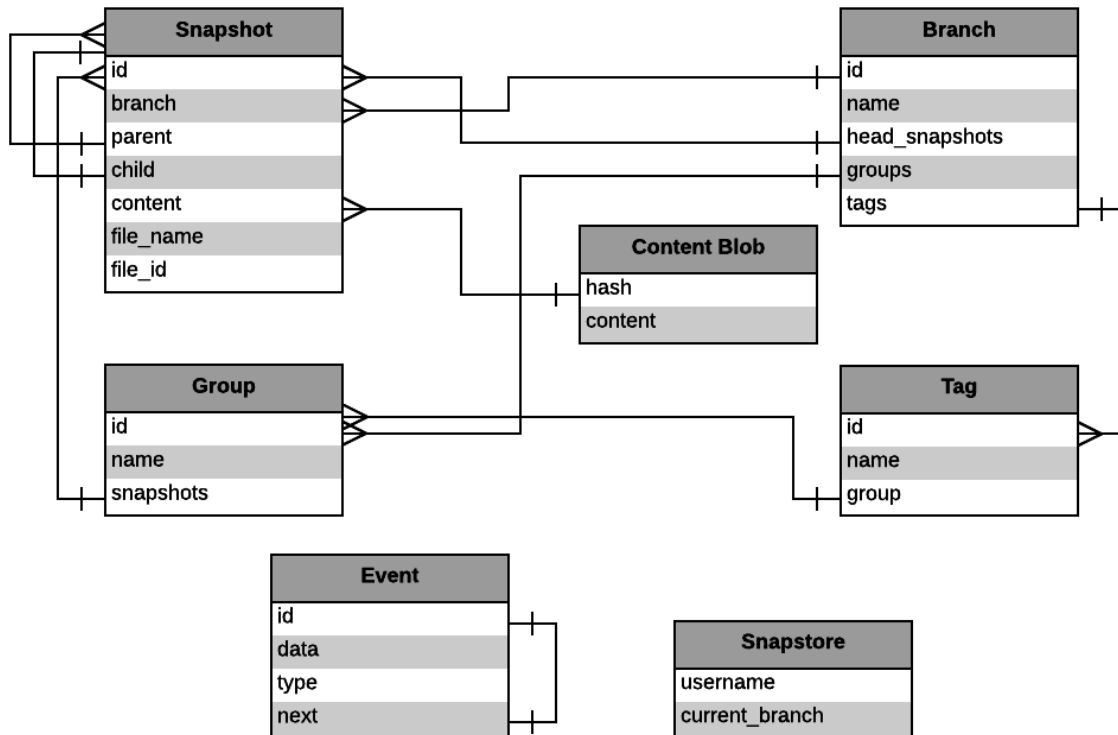[2]www.socket.io

[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.
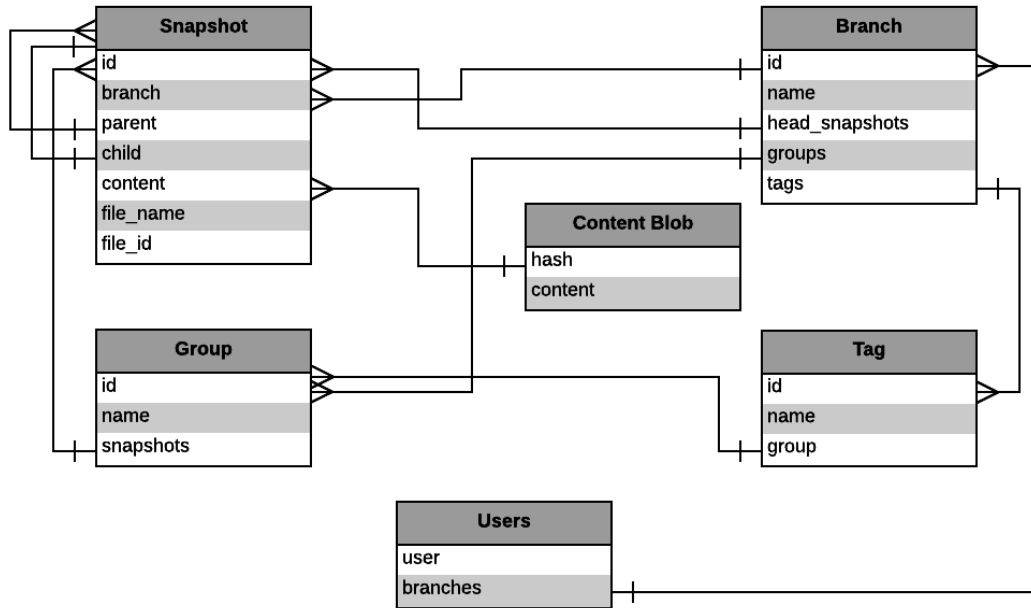
Figure 4-2: Snapstore server data model.

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, 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 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.

## 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, 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 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. DESQ seeks to reach eventual consistency between these queues so that every client on a shared branch has the same data.

**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 11). Once Client-DESQ is called, the event listener is turned off so that the algorithm cannot

be called again until it has finished.

This logic locks the algorithm and the event list. It is important to not add any new events to the event list while Client-DESQ is operating so that they can be added later, after the listener has been turned back on, and sent to the server.

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 14). If there are, the server sends them back as a "New Event", and the client saves them (line 28).

Once Client-DESQ begins, it sends each event in the event list to the server in order (line 33). 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 19-20).

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 22-24). 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.

A non-obvious statement appears on line 26, and it handles the following situation. A client, "Alice", sends a snapshot to the server at the same time as "Bob". Bob's snapshot is confirmed and sent to Alice. Alice's snapshot is then rejected. Bob's snapshot will then be sent to her 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 Alice's rejected snapshot.

**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

45

(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 (line 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). 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 [6]. Any event that requires that another, unseen or incorrectly ordered, event "happened before" 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 inconsistencies are returned to the client (lines 26-27).

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.

1: *events* ← pointer to *Events* collection
2: *socket* ← server socket connection
3: **listener** DATABASE-LISTENER
4:     **on** new database action *event* by user **do**
5:         *events*.append(*event*)
6: **listener** EVENTS-LISTENER
7:     **on** new *event* added to *Events* collection **do**
8:         Events-Listener.off()
9:         Client-DESQ()
10:         Events-Listener.on()
11: **listener** NETWORK-LISTENER
12:     **on** network connection with server established **do**
13:         Events-Listener.off()
14:         *socket*.send("Check For New Events")
15:         Client-DESQ()
16:         Events-Listener.on()
17: **listener** DESQ-RESPONSE
18:     **on** *socket*.response(*response, event, message*) **do**:
19:         **if** *message* == "Confirmed" or "Duplicate" **then**
20:             *events*.remove(*event*)
21:         **else if** *message* == "Reject" **then**
22:             **if** *event*.type == Snapshot **then**     ▷ response is a list of snapshots
23:                 Save *response*               ▷ these snapshot events fix the conflict
24:                 *event*.data.parent = *response*.head
25:         **else**                  ▷ *message* == "New Event"
26:             **if** *response.data.parent* is not a head snapshot **then**
27:                 **return**
28:             Save *response*
29: **procedure** CLIENT-DESQ
30:     **for** each *event* in *events* collection **do**
31:         **if** network connection is lost **then**
32:             **break**
33:         *socket*.send(*event*)

Figure 4-3: Client-DESQ pseudocode

47

```
 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 For New Events" then
 6:             newEvents ← eventMap[socket.user]
 7:             if newEvents.size() ! = 0 then
 8:                 socket.send(newEvents, "New Event")
 9:                 eventMap[socket.user] = []
10:         else if event.data.id in database then
11:             socket.send(event, "Duplicate")
12:         else if event.type  == Group then
13:             if event.data.snapshots in database then
14:                 Confirm-Event(event, socket, eventMap)
15:             else
16:                 socket.send((), event, "Reject")
17:         else if event.type  == Tag then
18:             if event.data.group in database then
19:                 Confirm-Event(event, socket, eventMap)
20:             else
21:                 socket.send((), event, "Reject")
22:         else                                      ▷ event.type  == Snapshot
23:             if event.data.parent is head snapshot then
24:                 Confirm-Event(event, socket, eventMap)
25:             else
26:                 conflictSnapshots ← All snapshots between event.parent and head
27:                 socket.send(conflictSnapshots, event, "Reject")
28: procedure CONFIRM-EVENT(event, socket, eventMap)
29:     socket.send(event, "Confirm")
30:     save event in database
31:     for each user with access to event.branch do
32:         if user is connected then
33:             socket(user).send(event, "New Event")
34:         else
35:             eventMap[user].append(event)
```

Figure 4-4: Server-DESQ pseudocode