# Chapter 4

# Implementation

This chapter details the implementation of Snapstore. That includes relevant implementation decisions, the current status of Snapstore, and important algorithms.

Snapstore was built to be cross-platform using Electron[1]. To offer consistency, we used web sockets[2]. Once a client is connected over a socket, Snapstore can be constantly pulling in and pushing 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 functonality 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 database. Each database has a collection of snapshots, branches, groups, and tags. It also has one snapstore document and a binary large object (blob) database.

The snapshot collection holds every snapshot in the local repository. Each snapshot knows the branch it belongs to, its parent and its child, the hash of its content,

[1] http://electron.atom.io/
[2] www.socket.io

and the name and id of the file associated to it. It also contains a boolean that states whether or not it has been confirmed by the upstream server.

When a file is saved, the resulting snapshot must first decide what kind of snapshot it is. Whether it is a create, update, rename, or delete snapshot dictates how it will populate its data fields. A create snapshot, for example, has no parent snapshot. When the snapshot is created, it is added to the collection of snapshots, and the branch updates its list of head snapshots to include it and not its parent.

If two snapshots have the same content, they point to the same blob data in another database table to save space. This blob collection holds hashes of the content along with the actual binary content. Further, duplicate snapshots can be flagged when connecting to the server by identical ids.

The branch data structure has a name that it uses as an identifier for a user, a list of its head snapshots, and a list of groups and of tags in it. When a new branch is created, it is added to the local repository with only a name. If it was cloned from another branch, all head snapshots, groups, and tags are brought in as well. Subsequent actions such as snapshot creation populate the branch. When switching to a branch, the heads snapshots for the target branch are read and applied to the Snapstore folder. The appropriate snapshot trees for each file in the branch are read by starting at each head snapshot and searching backwards.

The group data structure only references a set of one or more snapshots and has a name. The tag data structure, in turn, references a group and has a name.

The snapstore metadata structure holds all necessary metadata for the application. This includes the user's username along with their password for identification purposes with the server, and it holds the value of the current branch. Snapstore uses the current branch to pull up the most recently used branch on startup.

### 4.1.2 Upstream

On the upstream server, the snapshot, branch, group and tag 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.

The user model on the server is a mapping of users to branches to which they have access. It uses this mapping to add users to appropriate socket rooms when they first connect to the Snapstore server. Once users are in those rooms, they can be updated with changes to that branch.

## 4.2 User Interface

**Pictures will go in here when the UI is done.**

## 4.3 DESQ Algorithm

### 4.3.1 Shared Branches

For Snapstore, we wanted users to be able to work on a shared branch. As described earlier, a shared branch is a line of development where a change from one user is propagated to all other users on that line as soon as a network connection is avilable. If there are multiple connected users on a shared branch, a change made by one of them should result in changes to the filesystems of all other users, so as to keep all local working directories consistent.

Furthermore, if there are multiple users on a shared branch, they should each be able to work independently, confident that any changes they make will not be lost. They should instead be intelligently inserted into the snapshot history of the branch. New changes can come in through the network while a user is working, but it should not affect their ability to send their own changes.

We have opted to use a last-write-wins methodology when saving changes between users because it would be an easier paradigm for non-technical users to understand. While this methodology might cause some offline edits to be very far removed from their original parent, we believe that it is appropriate for two reasons. First, in the current highly connected environment of today's computing, making that many offline edits is typically done by choice. Second, if offline edits are indeed an issue, Snapstore allows users to create a separate branch for highly disconnected development.

### 4.3.2 Network Issues

The workflow described above can be difficult to maintain. Multiple users can be making multiple edits at the same time, increasing concerns of concurrency. Furthermore, network concerns and partitions increase the difficulty and uncertainty of this problem.

Imagine a user goes offline and makes multiple edits, all while their shared file is being written to by other, online users. Snapstore should be able to handle their reintroduction to the network without destroying the branch history. Simply trying to push these changes would be pushing an incorrect snapshot tree structure to the server.

The most important invariant Snapstore must provide is a shared snapshot tree for its users. We take the approach that any data that reaches the upstream server and is accepted should be regarded as fact. That is, any snapshots that are confirmed by the server should not be undone. With this invariant, we can more adequately reason about how to propose a protocol algorithm for this process, an algorithm we call Distributed Event Sychronization Queue (DESQ).

### 4.3.3 DESQ

We propose the DESQ algorithm. The algorithm seeks to synchronize all queues - the server queue and all client queues - and to reach eventual consistency in the ordering of their events. For Snapstore, these events are snapshots and their queues are the snapshot trees for a particular file in a particular branch.

In Git terms, this algorithm tries to push these events to the upstream repository. If there is a conflict, the branch rebases and tries to push that event again. This continues until all events are successfully pushed. The rebasing keeps the snapshot tree and the workflow for the branch linear. However, while Git creates new commits during a rebase, Snapstore uses the existing snapshots in the resulting snapshot tree.

These snapshots, at their core, describe a set of database actions and therefore describe an ordering that all parties can agree upon for system-wide accordance.

When a new snapshot action in the database is triggered, that snapshot data is saved to the client queue. This queue, by itself, is a guaranteed in-order sequence of all snapshot database actions by the client. Each snapshot in the queue is related to its parent and its child by a pointer, and these pointers are used to detect inconsistencies. If the client is working by themselves, in their own branch, this queue will simply be mirrored by the server when the network in connected. If, however, the client is working with another client on the same snapshot tree, there may be concurrent events being sent to the server. This can result in issues of ordering across the system.

DESQ is an algorithm that begins when an inconsistency is detected in the system. This can happen in two ways. First, if a client creates an snapshot (that is unconfirmed by default), The algorithm will begin to try to get the snapshot 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 to the server's queue, it will pull in those changes to make the queues consistent. Once the algorithm begins, it will not stop until the inconsistencies are resovled.

Note that this protocol can proceed only when network connections between the server and client are open. If they are closed, the snapshots are queued in the client until the network is available. Then, they are processed in the same way. In the case of Snapstore, we use sockets for all network connections.

### Confirmed Snapshots

In the most basic case, a single client is creating snapshots in their own queue, with no other users having access to that queue. The client will create a snapshot, the snapshot will be sent to the server, and the server will see that no additional snapshots have been added since the parent of this new snapshot. The server will add this snapshot to its version of the queue, set its confirmed flag to true, and send a response back to the client. On the client machine, the snapshot's confirmed flag will also be set to true, and the client can continue with full confidence that this event is consistent for all version of this particular snapshot queue.

### Receiving Snapshots From the Server

When sharing a particular snapshot queue, more than one client will have read and write access to it. If a certain client sends a snapshot to the server and it is confirmed, then that snapshot must be propagated to all other involved clients. When a snapshot is confirmed, the server will find all clients that have access to that queue (all clients with access to the queue's branch). It will then push that snapshot, with the confirmed flag set to true, to those clients. Because this is a confirmed snapshot coming from the server, the other clients can append this snapshot to their own queue, knowing that all queues are still consistent in the system.

When a snapshot comes in from the server to the client, to be inserted into the queue, its parent snapshot should be the end of the client's queue. This is because the rejected snapshot protocol (which can be happening simultaneously) will already be including the snapshot with its response, so there's no need to apply it in this case.

### Rejected Snapshots

To combat the concurrency issues, DESQ takes an approach that results in a last-write-wins methodology. When a snapshot is logged to the clientâĂŹs queue, it is sent to the server to be verified. The server then verifies whether or not it has seen a different snapshot from another client. This is done by checking the parent of the incoming snapshot. If the parent of the incoming snapshot matches the last known confirmed snapshot on the server, it is allowed in. If another client has already sent a snapshot that has been confirmed, then that snapshot will not match the incoming snapshot's parent.

If the server has seen other snapshots, making the server queue inconsistent with the client queue, it rejects the client's snapshot. The rejected snapshot then goes back to the client, along with the snapshot(s) that caused the rejection. These additional snapshots are inserted before the rejected snapshot in the client's event queue, and the rejected snapshot is queued again for confirmation at the front of the queue and sent to the server, starting the algorithm all over again.

Because the rejected snapshot is sent to the front of the queue of snapshots to be sent to the server, this process can continue without disrupting the inherent correct ordering of snapshots for a single client. So, if a client has made multiple offline edits, only the first of those will trigger the rejection. The snapshots causing the rejection will be inserted, and the process will start over with that first snapshot.

Furthermore, this robust way of handling rejections allows for the system to handle consecutive rejections. This can occur in the case where other clients are sending snapshots to the server while another clients' snapshots are beign rejected.

### Duplicate Events

In the case of network outages, it could be the case the client goes down before the server can respond that it has received a valid snapshot. In this case, when the client comes back online, it will simply retry to send that snapshot. Because that snapshot's ID already exists in the server's queue of snapshots, it will simply respond that it has already received the snapshot. This will allow the client to confirm the snapshot as valid while not having to re-push the snapshot to the server.