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, tags, and events. It also has one snapstore document and a binary large object (blob) collection.

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

(which is stored in the blob database), and the name and id of the file associated to it.

When a file is saved, the resulting snapshot must first decide what kind of snapshot 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 the snapshot is created, it is added to the collection of snapshots, and the branch updates its list of head snapshots to include this new snapshot, clobbering its parent.

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 actual binary content. Further, duplicate snapshots can be flagged when connecting to the server by looking at their ids.

The branch data structure has a name, 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, the cloned branch will point to all head snapshots, groups, and tags from the original branch. 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. To show the history of a file in the UI, the head snapshot of the file is first found; the rest of the history is found by searching backwards through the snapshot graph.

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 event collection stores all of the unconfirmed snapshots, groups, and tags on the client.

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 available. 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.

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. These changes, whether they deal with snapshots, groups, or tags, should be persistent. 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 collaborating between users on the same branch because it is an easier paradigm for non-technical users

to understand. Plus, with the potential amount of conflicts between snapshots on a shared branch, the number of merges would be too high. So, merging is only done between branches on the local repository.

While this methodology might cause some snapshots created 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 snapshots and groups, 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 graph structure to the server.

We take the approach that any data that reaches the upstream server and is accepted should be regarded as fact. Any events 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

The DESQ algorithm seeks to synchronize all event queues - the server queue and all client queues - and to reach eventual consistency in the ordering of their events. The

Algorithm 1 DESQ

```
1: procedure CLIENT-DESQ
2:     events ← collection of Events
3:     socket ← server socket connection
4: while events not empty:
5:     socket.send(events(0)).
6: on socket.reponse(response, message):
7:     Save response;
```

```
1: procedure SERVER-DESQ
2:     events ← collection of Events
3:     socket ← client socket connection
4: on socket.receive(event):
5:     if event in events then
6:         return "Duplicate Event".
7:     if event.type ! = snapshot then
8:         event.confirmed == True.
9:         events.append(event).
10:        socket.send(event, "Confirmed").
11:        socket.room.send(event, "New Event").
12:    else
13:        if event.parent is head snapshot then
14:            event.confirmed == True.
15:            events.append(event).
16:            socket.send(event, "Confirmed").
17:            socket.room.send(event, "New Event").
18:        else
19:            conflictSnapshots = All snapshots between event and event.parent
20:            socket.send(conflictSnapshots,"Reject Event").
```

events in Snapstore are the creation, update, or deletion of any data structure. The queue of events is the collection of events on the client.

For Snapstore, it is important to maintain a consistent snapshot graph between users, but 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 upstream.

This algorithm tries to push these events to the upstream repository. In Git terms, 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 graph and the workflow for the branch linear. While Git creates new commits during a rebase, Snapstore uses existing snapshots in the resulting snapshot graph.

These events describe a set of database actions and therefore describe an ordering that all parties can agree upon for system-wide accordance. When a new action in the database is triggered, that event data is saved to the client event queue. This queue, by itself, is a guaranteed in-order sequence of all database actions by the client. Each event 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 branch, 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 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 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 events are queued in the client until

the network is available. Then, they are processed in the same way.

Confirmed Events

In the most basic case, a single client is creating events in their own queue, with no other users having access to that queue. For example, if the client creates a snapshot, the snapshot will be wrapped in an event and added to the event queue. The event will be sent to the server, and the server will see that no additional events have been added since the parent of this new event (the parent of the snapshot). The server will add this snapshot to its version of the queue, and send a response back to the client. On the client machine, the event is registered as confirmed and removed from the event collection and queue.

Receiving Events From the Server

When sharing a particular branch, more than one client will have read and write access to it. If a certain client sends an event to the server and it is confirmed, then that event must be propagated to all other involved clients. When an event is confirmed, the server will find all clients that have access to that event's branch. It will then push that event to those clients. Because this is a confirmed event coming from the server, the other clients can apply this event to their local repository, knowing that all queues are still consistent in the system.

When an event comes in from the server to the client, if it is a snapshot event, 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. This only applies to snapshots because they are the only type of event that can cause a conflict. When a snapshot event is logged to the client's queue, it is sent to the server to be verified. The server then verifies whether or not

it has seen a different snapshot event 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 event 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's history inconsistent with the client's history, it rejects the client's snapshot event. The rejected snapshot then goes back to the client, along with the snapshot(s) that caused the rejection. The snapshot(s) that caused the rejection are found by traversing child pointers. These additional snapshots are inserted at the end of the client's snapshot graph (before the rejected snapshot). The rejected snapshot is sent to the server again for confirmation, restarting the algorithm.

Because the rejected snapshot is kept at the front of the event queue to be sent to the server, this process can continue without disrupting the inherent correct ordering of events for a single client. So, if a client has made multiple offline events, only the first of those could trigger the rejection.

Furthermore, this algorithm allows the system to handle consecutive rejections. This can occur in the case where other clients are sending snapshot events to the server while another clients' events 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 an event. In this case, when the client comes back online, it will simply retry to send that event. Because the ID of that event's data structure already exists on the server, it will simply respond that it has already received the event. This will allow the client to confirm the event.