Chapter 4

DESQ Algorithm

4.1 Introduction

Any distributed system requires well planned algorithms to coordinate changes across machines given unreliable networks. Snapstore gives the user the ability to work with other users in an immediate way by allowing them to share branches. These branches are independent directory structures. On a shared branch, all involved users are working on a filesystem located in the cloud. Changes made to this filesytem are propagated everywhere when a network connection is open.

However, this workflow can be difficult to maintain. Multiple users can be making multiple edits at the same time, increasing concerns of concurrency. Furthermore, network concerns 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. What should be the behavior of the system when they re-establish network connection? 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. I 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).

4.2 DESQ

The goal of this queue synchronization is to have all queues - the server queue and all client queues - reach eventual consistency in the ordering of their events. These events describe all database actions and therefore describe an ordering that all parties can agree upon for system-wide accordance. 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 on the client until the socket is open. Then, they are processed in the same way.

When a new event in the database is triggered, that event is saved to the client 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. If, however, the client is working with another client on the same event queue, 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. If a client creates an event (that is unconfirmed by default), DESQ will begin to try to get the event confirmed by the server and shared will all appropriate users. If a client connects with 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 DESQ begins, will not stop until the inconsistencies are resovled.

4.2.1 Confirmed Events

In the most basic case, a single client is creating events on their own queue, with no other users having access to that queue. The client will create an event, 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 server will add this event 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 event'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 event queue.

4.2.2 Receiving Events From the Server

When sharing a particular event queue, 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 queue and push that event, with the confirmed flag set to true, to those clients. Because this is a confirmed event coming from the server, the other clients can append this event to their own queue, knowing that all queues are still consistent in the system.

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

4.2.3 Rejected Events

To combat the concurrency issues, this algorithm takes an approach that results in a last-write-wins methodology. When an event 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 event from another client. This is done by checking the parent of the incoming event. If the parent of the incoming event matches the last known confirmed event on the server, it is allowed in. If another client has already sent an event that has been confirmed, then that event will not match the incoming events parent.

If the server has seen other events, making the server queue inconsistent with the client queue, it rejects the client's event. The rejected event then goes back to the

client, along with the events that caused the rejection. These additional events are inserted before the rejected event in the client's event queue, and the rejected event is queued again for confirmation at the front of the queue, sent to the server, and confirmed.

Because the rejected event is sent to the front of the queue of events to be sent to the server, this process can continue without disrupting the inherent correct ordering of events for a single client. Furthermore, this robust way of handling rejections allows for the system to handle consecutive rejections, in the case that other clients are sending events to the server while another clients events are beign rejected.

4.2.4 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 event. In this case, when the client comes back online, it will simply retry to send that event. Since that event already exists in the serverâĂŹs queue of events, it will simply respond that it has already received the event. This will allow the client to confirm the event as valid while not having to re-push the event on the server.

4.3 DESQ and Snapstore

In the case of Snapstore, events in the queue are snapshots. The queue itself is the snapshot tree, a unique collection of events that pertains to a unique (branch, file) tuple. For a given shared branch, various users can queue up snapshots for files to be confirmed by the server. For multiple snapshots coming in at the same time, any of them that apply to the same file will be in contention to be the first confirmed snapshot. All subsequent snapshots will be rejected and returned to the client that sent them along with the confirmed snapshot. This will allow the users to update their individual snapshot trees with the "blessed" events from the server.

This synchronization system is last-write-wins. For a highly disconnected environment, one in which users make dozens or hundreds of edits at a time offline, a rejected

event can be highly separated from its initial parent after DESQ is finished. Firstly, our current computing landscape and working environment is so readily connected that users find themselved working with network connection more often than not. Secondly, if the disconnected environment is indeed an issue, the Snapstore user can simply create another branch from the main branch. This will allow users to work completely independently and in a disconnected way, knowing that they can merge their work at a later time.