Bachelor project proposal: Incremental Distributed Conflict-free Data Synchronization with Write-exclusive Locks and ZFS

Silvan Mosberger

October 23, 2018

1 Description

The goal of this project will be to implement a data synchronization tool based on the ZFS filesystem (or any other base that supports the required primitives). The idea is:

- There is a single node holding the write-exclusive lock to the data, only that node can make modifications
- All other nodes only have read access to a potentially older version of the data.
- A single command is used to transfer the write lock from a node A to node B. This will update B with the newest data from A, while making A a read-only version.

Additionally only the two nodes involved in the lock transfer are required to be online during that operation, which turns out to be non-trivial if efficient incremental updates are wanted. There won't be any modification conflicts between different nodes, because only a single one can ever update the data, which stands in contrast to most conventional data synchronization tools which need to fall back to merging strategies. To implement this, a number of primitive operations are assumed to be available. What makes this work so well is that the ZFS filesystem just so happens to expose the necessary primitives and is very efficient at executing them. Haskell will be used for the implementation due to its big library ecosystem and my familiarity with the language.

1.1 Primitives

The necessary primitives are:

- An efficient snapshot can be taken of the changed data
- The difference between snapshots can be efficiently serialized, transferred between nodes, deserialized and applied to an existing snapshot with a matching version
- A copy of the data on a node can be set read-only or read-write

1.2 How it works

The table below shows a simple walkthrough of how the tool is meant to work. '*' stands for that node having the write lock. A number stands for a snapshot identifier, increasing by one every time the data is modified. Every node may persist a list of snapshots. Operations are `snap` for creating a snapshot, `ro` for setting a node read-only, `rw` for setting a node read-write. `N->M` for updating node `M` with info from `N`. This example shows 3 nodes, A, B and C, starting with A having the write lock, that being transferred to B, then to C, then back to B again, then back to A.

operation	A	B	C	notes
start	*	-	-	We start with a single node A with the write lock
ro A + snap	0	-	-	Making A read-only and taking the first snapshot with id 0
A->B	0	0	-	Initializing node B with the new snapshot
rw B	0	0*	-	Giving the write lock to B, after this point B may modify the data
ro B + snap	0	0,1	-	Keeping snapshot 0 to allow knowing the diff between 0 and the snapshot 1 we might want to send A in the future
B->C	0	0,1	0,1	Send both snapshots 0 and 1 to C, such that C could also know the diff to A without relying on B
rw C	0	0,1	0,1*	
ro C + snap	0	0,1	0,1,2	Keep snapshot 0 for sending to A, snapshot 1 for sending to B
C->B	0	0,2	0,2	Because we sent to B and B was the only node being in state 1, we can discard snapshot 1
rw B	0	0,2*	0,2	
ro B + snap	0	0,2,3	0,2	Keep snapshot 0 for sending to A, snapshot 2 for sending to C
B->A	2,3	2,3	0,2	Because we sent to A and A was the only node being in state 0, we can discard snapshot 0
rw A	2,3*	2,3	0,2	

2 Usecase

The main usecase is data synchronization for a single user having multiple machines. Because the user won't work on two machines at the same time, there are fixed points where they switch from one machine to another. That's when this tool can be used to update all changes they did on the current machine to the other machine. Because all other machines were in read-only mode during the time of modification on the current machine, the user won't be annoyed with having to deal with merge conflicts.

Another usecase is for backing up data. Updating a node with new data while not deleting old snapshots is equivalent to backing up to that node. If dataloss then occurs on a node, it's very easy to recover from a node with a backup by just inverting the data flow. While this functionality would be nice, it's secondary to this project proposal.

3 Goal

The goal of this project is to finish a solid usable implementation of the system described above on the ZFS filesystem. The code should be split into 3 packages: One library package for the implementation of the system parametric over the primitives used. One for the executable using the ZFS primitives. A third package should contain an executable that implements such a system based on primitives in an arbitrary filesystems, which won't be as efficient, but useful for both testing and people that don't have ZFS available.

4 Potential Extensions

While this system can be implemented with just a single command to transfer the write lock between nodes, a number of useful extensions can be added:

- Support transferring the updated data without transferring the write lock. This is needed to implement the backup usecase.
- In case that the node holding the write lock isn't online, it should be possible to force an acquisition of the write lock, temporarily creating multiple locks, at the expense of potential data loss in case the two locks join again. This is also useful for the backup usecase.
- Generating a simple visualization of the data flow between nodes over time.
- Adding other useful basic commands such:
  - Acquiring the write lock without having specified which node currently holds it by asking around, useful when a node wants to write to the data.
  - Pushing the write lock to another unspecified node, useful when the node holding the write lock wants to go offline, what's the best node to push it to?