(full title: thinking about desirable properties of general-purpose abstractions for dealing with versioned decentrally replicated semi-structured user-defined state)

i own multiple computers. i have some state on those computers, and sometimes apparently identical or similar state is accessed from or stored on multiple computers, for various reasons. i want something to manage this for me. i know of some existing paradigms for some of this, which i will list below (i am definitely skipping some things); after, i will explore the grand union of their featuresets that i want:

- git (dvcs; also … others)
- zfs (snapshots for revision control and send+recv for synchronisation)
- google drive (cloud storage; also icloud, onedrive, etc. nfs and cifs/smb is kind of a 'lite' version of this)

what do i like and dislike about these? i will note: i personally use git, zfs, and various ad-hoc mechanisms. i am unhappy with all of these. i think zfs is the best at doing what it does (but there is a lot it doesn't do); and git can do the most things (but there is still a lot of interesting stuff it can't do, and it's mostly bad at doing things outside of its niche); and i use ad-hoc mechanisms where others won't do

with git

:

- we have eventual consistency and strong availability; this is good and nothing else does this
- changes can originate in distinct trust domains (gdrive does this but not very well)
- branches are a neat idea but i don't need them a lot of the time. in particular: a named branch is actually just another versioned datum, just like a file, except it always refers to a commit. oops! named branches as a shadow domain of replicated state? :s (probably this is why we end up with ad-hoc namespaces for branch names to ensure ~unique ownership :)
- there is an unfortunate and arbitrary separation between different repositories
- there is a massive 2x space bloat for large objects (a large file is stored once in a checkout and once as a blob in .git). (in theory, i think this could be fixed without any protocol-level changes to git. but it would be a pain)
- synchronisation must be performed manually with pull and push. the typical model is hub-and-spoke with a central server. in fact you can have an arbitrary connection graph, but no one does this because you have to manage the graph manually and it's a total pain to figure out who has what state
- (i seem to remember hearing some murmurings that some of the microsoft git-lfs tooling (and facebook analogue) might have some like automatic fetch or something? and possibly a custom filesystem too—not sure if this was for spacebloat or staleness detection, or maybe both. but this is not mainstream, it's probably a bit of a pain to set up, and the fetch thing doesn't give you more than centralised availability and trust)

with zfs

:

- there is no possibility of distributing writes as far as i know; i think snapshots are strictly linear, not a dag (in any event there is no nice merge so it can't be more than a tree)
- but there are a couple of interesting things
- there is no space overhead, unlike git
- snapshots can be deleted if they are taking up too much space. it is common to have exponentially spaced snapshots —arbitrary snapshot deletion policy
- there is the same arbitrary delineation as with git; it is the 'filesystem' rather than the 'repository' (but pools are maybe a good abstraction!)

with google drive

:

- clients do not commit to storing data; they can stream. this is transparent
- there is some availability (not sure eventual consistency? should experiment; i think there is a bit but i don't know if you can do manual granular conflict resolution like git)
- there are no arbitrary delineations; there are just all of the google drive files

. there are access controls; a file

has an owner

and some number of collaborators

. (but making a copy of a datum to change it if you don't have permission to change the original is cheesy and stupid)

granularity of versions:

- git: commit (arbitrarily delineated)
- zfs: snapshot (always atomic and 'global' (at the 'fs' level); situated in (real) time)
- google drive: 'change'. typically every time you save to a file
- specifically in the case of a google doc: a change is something intuitive but a bit odd and complicated when you think about it; it tries to capture every insertion and deletion, so no state change is lost (also yadda yadda crdt yay—though this is a slightly different concern and i think the shared-text-editing case can be layered on top of plain versioned state). (of course, we can go even more granular—keep every keystroke)
- specifically in the case of a google doc: a change is something intuitive but a bit odd and complicated when you think about it; it tries to capture every insertion and deletion, so no state change is lost (also yadda yadda crdt yay—though this is a slightly different concern and i think the shared-text-editing case can be layered on top of plain versioned state). (of course, we can go even more granular—keep every keystroke)
- commit is maximally general; primitive to implement other things. domain/application-specific notion of a 'change'; gdocs example above is for text editing, but we can also have editors for various structured formats (video image audio 3d-model…) which can provide their own notion of a 'change' at whatever granularity we decide to care about

why would i want to replicate state?

1. for durability

: if one disc fails, it's ok, because the data are still on the other disc

1. for availability

: if i am at computer x, and want to access some datum that was on computer y, but i can't talk to computer y, it's ok, because the datum is also on computer x

google drive distinguishes these: durability is handled by having your data inside of google's cloud (where it is georeplicated in triplicate, one hopes), and availability is handled by also having a copy on whatever local computer. (data do not 'become durable' until they are synch'd to google's cloud.) but that is because google wants to spy and to centralise control, so it monopolises durability (and to some degree even availability—if x and y want to sync, they can't do it directly, but must take a hop through google, even if they can talk to each other directly; talking to google slows them, and might even be impossible), and really there is nothing to be lost by conflating durability with availability. when some datum is on computer y, i can ask for it to be replicated to computer x without caring whether the 'reason' is durability or availability (it could both at once).

(to play devil's advocate: the reason google does what it does is also that google wants to and is able to provide very high-quality and highly available durable centralised storage as a service. and then it sees no point in building a more complicated decentralised mechanism. but we want the more general mechanism because we care, and then google drive as it exists should be a perfectly good point in the space of expressible policies.)

being able to stream is good. when we have not explicitly requested that an object be replicated locally, if a connection to somebody who has it can be established, then we can stream it. caching policies etc. all nominally transparent to applications just accessing the state, but if they want to be able to care they should be able to try to care, and then i should be able to decide contextually that i don't want to let them care

there is overlap with replication and retention policies for content-addressed data, but it is not the same. still, something for content-addressed data is useful, and it can be used as a primitive for building something for versioned data. a replication policy for versioned data can be automatically translated into a replication policy for content-addressed data (e.g. i want the big beefy server to store every 'change' ever, but my laptop only has to store the 'commits', and even then i only really need

the ones from the last few years or so, or only the ones that changed lines of code that still exist so they'd show up in a git blame…)

versioned means authenticated. by whom? 'the entity who made the change/produced the new version'. synchronisation policies can use this. some transitivity nonsense (x always accepts changes from y, and y has accepted z's change to some thing). tg(?)'s (and arcan's) notion of 'group of devices that belong to one person' is useful here i wonder if there are not some essential tradeoffs between never having to worry that some change was missed and not accidentally picking up unwanted changes. we can of course still advance the pareto front a lot. we also have merge policies; this overlaps with patch composability soundness (todo link to my musings on that)

is there overlap with consistent referentially transparent data (see:notes about the namespace buzzword)? maybe. in general any

consistent replicated state (let alone reftrans) is a special case of eventually consistent state. that framing might sometimes be useful (and we can certainly define and support the interface intersection for whomever wants it); i would guess frequently not, but otherwise they can still compose in at least one other way: versioned data can be embedded in a keyspace (key ‘datum x at version y’—version also having some structure)