File Synchronization — Notes (from Old Files[1])

May 6, 2016

I • Background

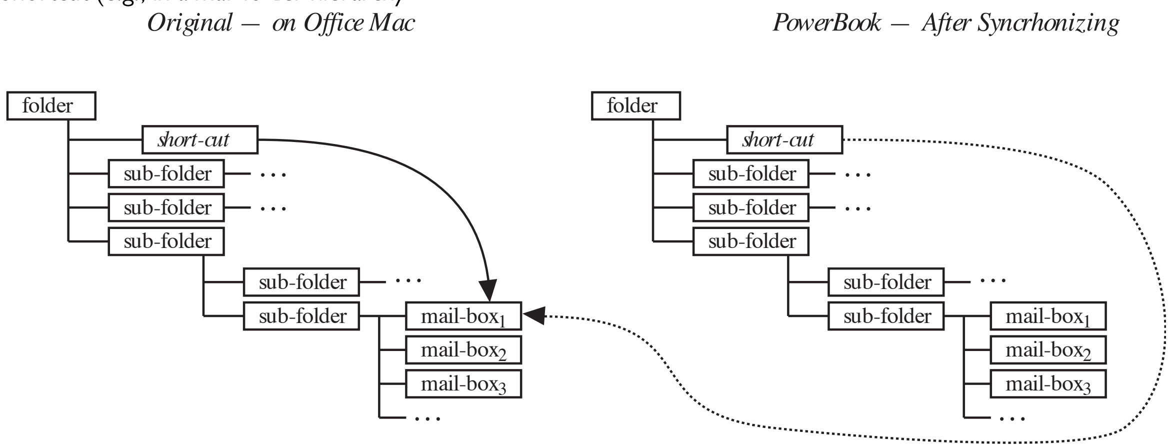
A. Wrong to think of the problem in "horizontal" terms: *replication, synchronization, duplication*, *coordination*, etc.

B. Should instead be thought of "vertically": as layered identities, each layer fanning-out to layers below.

1. Claim: though unfamiliar (to programmers), it is ultimately easier to think of a single entity at various layers of abstraction, instead of multiple (more or less synchronized) copies.

II • Examples

A. Shortcut (e.g., in a mail folder hierarchy


Original — on Office Mac

PowerBook — After Syncrhonizing

B. Moving a file

1. Files f_α and f_β previously synchronized (on machines α and β). f_α moved different folder; f_β's contents edited. When next synchronised, typically end up with two files on both α and β; one properly renamed but with out-of-date contents; the other with up-to-date contents but in old place with old name.

C. Flagging files

1. Want to identify critical files to be backed up every hour. You create a special folder of aliases (hard or soft links) of each file you want backed up. Backup program merely *makes copies of the aliases*, not copies of the files pointed to. *Copying* the files into the special folder won't work, obviously. So solutions require *special techniques* (such as "tagging" or something, which use a different mechanism.

D. Cache coherence strategies: for multiple processors sharing memory, "write through," etc.

III • Principles

A. Initial

1. **Abstraction**: Don't think about different versions of documents. Instead, deal with documents as single, coherent unities.
2. **Disconnection:** Perfect realtime coordination between multiple instances of file (folder, project, data base, etc.) is unattainable. There will always be periods of *disconnection* between two or more concrete instances of a single unitary abstraction.
3. **Failure.** Rather than (in the first instance) providing a complex, plural story that is *true*, provide a simple, singular story that remains *an unachievable ideal*. Then: admit the ideal is unattainable, and explain the failure in its terms.
4. **Adaptiveness:** Adopt (level of) identity that makes sense *for the given operation*, *in the local situation.*
5. **Accountability:** Systems should *fail intelligibly*; detect and report failures in terms of the ideal[2]

[1] from "Up and Down" and "Layers of Identity" (in some files called "(· Synchronization", for some reason).

[2] Tellingly, a tacit understanding of this is evident in error messages. E.g.: "Conflict: you have edited *this file* on both

B. In sum

1. Think in terms of the (single) abstract entity whenever possible;
2. Think in terms of the (plural) concrete entities when necessary—including, paradigmatically, those times when it is necessary to understand how the former, singular, goal cannot be met.

IV • Solution

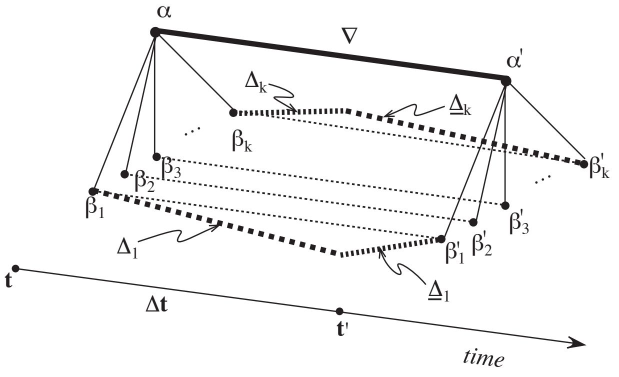


Diagram is screwed up—suggests that Δ_l and Δ_k are somehow inverses!

There is a question about whether one lifts or drops Δs, or only resultant states. (Equivalent in some sense.)

In Folia: α is not abstract; *rather, closest thing to α is what is on the server, which is* authoritative. *Do pass around Δs. Lifts are therefore identity. At the server, no idea of how to interpret or apply Δs; just good for passing them around (Folia never got to sending whole states; note in video streaming that …). With authority, on conflict,*

Google docs have "OT"—operational transformation. All completely identical models (all Δs could apply to arbitrary instances; only issue is temporal sequencing). No lifting or dropping… Standard way to handle it in CS is to lock (with refresh time-out). Call these the easy case*: just replication (distributed, temporal Δs, but accountable only to strict identity).*

So: question (for what follows): (i) what is not easy, *(ii) how common (or useful) are non-easy cases; and (iii) is the lift/drop model useful for them?*

We already know that the $\underline{\Delta}$ may not happen as such (perhaps it is better to lift, make the Δ at the α level, and then drop. But it is possible that even the Δ_l and Δ_k should perhaps not be done at the β level either; rather, a lifted Δ should be "asked" of the abstract level (i.e., in terms of the more abstract model), and then reconcretized. This might enable constraint satisfaction (e.g., no cycles in file directories) to be checked at the level at which they are most simply conceived).

A. Explanation

1. Setup
   a. t: moment in the past when the various concrete entities β_i were last "synchronized"
   b. During the interval Δt from t until t', the concrete β_i were (physically) separate
   c. During that time a certain set of operations Δi were performed on each the various β_i (indicated in the diagram as Δ_1 and Δ_k), transforming them into the corresponding set of β_i'.

B. Problem

1. Determine what new set of operations $\underline{\Delta}_i$ need to be performed on the various β_i at time t' ("synchronization" time) in order to bring them all into synch.

sides"—intelligible only if one takes "this file" as referring *to the abstract file* (better than the only true horizontal thing that could be said: "these two files have both been edited since the last time they were synchronized").

2. I.e., given:
    a. States β_i at time t, and
    b. Operations (transactions) Δ_i
3. Figure out "clean-up" operations $\underline{\Delta}_l$ (two such clean-up operations are shown in the figure: $\underline{\Delta}_1$ and $\underline{\Delta}_k$).

C. Solution: figure out what *net abstract* operation ∇ has happened to abstract entity α.

1. Not: sum or union of all the concrete transactions Δ_l (i.e., something like $\mathbf{\Sigma}(\Delta_i)$).
    a. Each Δ_l, defined over a *specific* concrete entity β_l, may be concerned with issues *specific to that concrete instance*—issues that may or may not be shared by other concrete instances, and may not be defined at all at the abstract level. (E.g., whether to use American or British spelling; whether to print out on the printer at the office or the printer at home; which concrete instance to create a link to, etc.).
2. Instead: "lift" ('↑') operations from concrete to abstract level. Net transaction is:
    a. $\nabla = \mathbf{\Sigma}(\uparrow\Delta_i)$
3. Then: each net operation that should have happened to a concrete entity β_k would be approximately the "concretization" (call it "drop"; indicate as '$\downarrow\nabla$'). But since dropping may be specific to particular concrete instance, concretization must indexed, as '$\downarrow_i$'. So net operation that should have happened to β_k would be $\downarrow_k\nabla$.

D. Synchronization equation

$$\boxed{\underline{\Delta}_k = \downarrow_k\mathbf{\Sigma}(\uparrow\Delta_i) - \Delta_k}$$

Note: the '–' may not be the most perspicuous way of handling "what may meantime have happened on k"

1. 'Δ_k' ≡ concrete operations
2. '↑' ≡ abstract(ify)
3. '$\downarrow_k$' ≡ concretize
4. '$\mathbf{\Sigma}$' ≡ abstract operation sum (perhaps this should be 'integrate' or 'union)
5. '–' ≡ concrete operation difference

V • Notes

A. General

1. Abstract entities don't "exist," computationally (aren't concrete). Point: design the software *in their terms.*

B. Aspects

1. Divide operations Δ_i into conceptually separable—ideally (nearly) orthogonal?—different *aspects*. E.g.:
    i. Create
    ii. Modify zero or more of
        α. Name ⇐ i.e., Rename
        β. Location ⇐ i.e., Move
        χ. Contents ⇐ i.e., Edit
    iii. Delete

On easy model, client would know that they are orthogonal (i.e., $.\alpha.\beta \equiv .\beta.\alpha$). In Folia, the authority doesn't know anything about it. Google docs: authority does everything (client doesn't): basically a full-duplex of the content. So: if no "conflicts" (two renames, e.g.), pretty straightforward either way.

C. Mereology

1. Operations would need to be defined, recursively, over the mereological (part-whole) hierarchy
    a. Folia and Google docs drive down to lowest-level particular objects (e.g., non-overlapping regions, where "non-overlapping" could be wrt a *model* (e.g., sequence of ¶s, each ¶ a sequence of sentences, etc. Folia: character # wrt sections, ¶s, "unit.")
2. Probably require protocols or APIS (mail folder recursively apply scheme to the individual mailboxes; mail-

boxes to messages; word-processor to individual documents (so as to generate the ‘union’ of a set of edits, for example).

3. In each step down this part-whole hierarchy, the specific set of applicable concrete operations (the ‘Δ_K’) would likely change. (Cf. the old “OpenDoc’ and “OLE” frameworks.)

No deep challenges yet, to the easy model, because there are no differences among the instances.

D. Differences

1. Support *differences* among different concrete instances. E.g.:

a. *Partial replication:* On one machine, store only headers of messages more than a month old. But otherwise treat those (emasculated) messages in ordinary ways (file, forward, copy, delete; include pointers to them in documents; embed them in outgoing messages, etc.).

Is this done in imap? Don’t get the bodies at once. Or even if you fetch bodies, don’t fetch images; even if fetch images, don’t “download” attachments. Can still do things like forward and reply…

b. …This could be achieved by defining the concretize or “drop” operation ‘$\downarrow_{\text{laptop}}$’ to throw-away message bodies. That’s all; everything else would fall out!

OK, so far the concrete instance is just “less than” the “full” version. So drop discards some things. No changes to “lift”, except that one notes that certain lift operations won’t be coming from this instance.

c. *Variant spellings and formatting:*

i. E.g.: business plan in New York and London offices. Define α to be a document consisting of an abstract string of atomic words, and have corresponding ‘$\downarrow_{\text{NewYork}}$’ and ‘$\downarrow_{\text{London}}$’ operations instantiate those words with British and American spelling, respectively. Document could be edited at either end. E.g., in New York, changing the heading of a section to:

α. “Analyze the Role of Aluminum in Causing Alzheimer’s Disease”

would automatically cause the British version to turn into

β. “Analyse the role of aluminium in causing Alzheimer’s disease.”

ii. E.g., file names in Mac/Windows/Unix: (‘/’, ‘\’, etc.; ‘ ’ vs. ‘_’; “%20” vs. ‘) URL encodings. cr vs. cr-lf.

iii. Numbering file names…

iv. Capitalization?

v. Ligatures?

Note that to implement *anything remotely relevant to the “lifted” abstract level, you have to represent or describe the abstract level (e.g., the abstract Δ)* concretely. E.g., file directory and member model, *need to have a concrete representation of something being an abstract directory plus members…*

In many of the cases so far, it is probably simpler to use one concrete case as the simplest “description of the abstract case”; call this the “canonical case.” Then one can represent concretizations (drops) as Δs from the canonical model. (Note that being ‘canonical’ is not yet being ‘authoritative.’). However maybe the canonical is not necessarily the easiest for constraints.*).*

d. Spelling correction: one could use a single-instance version of such a scheme as a model of spelling-correction: ‘↑’ to lift attempted spellings into abstract words; the only ‘↓’, to spell them correctly!

e. Translation: Obviously, ‘↑’ to “lift” into mentalese (Hinton’s current 100,000 node graph); various ‘↓’s to drop into different natural languages (probably with human help)

Are constraints easier to state at the upper level. E.g., in file systems: can’t be cyclical. So: determination of whether there is a loop could be easier to determine at the abstract level.

Another example: "lifting" a "description" (wrt a model) of any named hierarchy (e.g., my file system) and import (drop) it into a different instance, like my Mail folder hierarchy.

f. *Versioning:* Would the scheme apply to versions? Do papers and software need to be treated differently?

E. Context dependence

1. What about synchronizing files (applications) in the system folders of different computers? No one even tries it; yet surely it is a crying need. (Cf. "install" and "uninstall" programs.)
2. "Local printer"—is that an "abstract" printer? At least this kind of **localization** should be treatable in this model?

VI • Work to do

A. File system: identify various operations (create, edit, delete, rename, move, etc.)

B. A *general* API? —for the general model. But then what would "specialize" it to particular cases?

C. Meta-data: should each "concrete" file or object, in a system, represent, in meta-data of some sort, what operations it knows how to lift and drop, how it should be treated, modified, moved or whatever?

1. E.g., should aliases/links "say", within them (in meta-data) how they should be treated in different circumstances—e.g., when copied onto a different physical volume?
   a. For example, the aliases mentioned in the early example (of hourly backup) could say, in their metadata, that *they* should not be backed up, but rather that the files they pointed to should be backed up; whereas other aliases might say that they *should* be backed up, but on the backup (or synchronized) volume the corresponding alias should point *to the file on the backup*.
2. Similarly, files could "say" whether, if "moved" (dragged) onto a new disk volume, whether the copy on the old volume should be deleted.

VII • Summary

A. Think "vertically" about abstract entities, concrete entities, and the relationships between and among them (such as abstraction and concretization).

B. Let system programmers convert those thoughts, *at the last possible moment*, and *at the lowest possible level*, into invisible "horizontal" synchronization operations on replicated media.