Up and Down, Instead of Across

A new way to think about replication, synchronization, and duplication

Brian Cantwell Smith

Replication is hot. Everyone wants to synchronize distributed copies of files—from Lotus Notes aficionado to laptop user to distributed data base manager. Wouldn't it be wonderful to be able to use the "same" mailbox from a portable, at home, at the office?

This memo argues for an odd conclusion (1st cut): that the notions of *replication, synchronization,* and *duplication*, in terms of which this class of problem is normally formulated, are the wrong way to understand replication, synchronization, and duplication.

1. Abstract entities

We meet at a conference; talk for hours about a neat idea. That night, in the hotel, I decide to write. Write what? A *paper*. Or some notes, a memo, an IP—whatever. But definitely *not* a "set of synchronized files."

Some analogies. (i) The 1978 BMW 3.0CSI was one of the most beautiful cars ever designed. "One" of the most beautiful cars. Yet they made lots of them. I don't own one, unfortunately. All I hanker after, though, is an *instance* of the car that was designed. (ii) Shakespeare's *The Tempest* is old; it was written in 1588. There are lots of copies of *The Tempest* around, including a dog-eared copy in my study. It's old, too. It was printed in 1968. (iii) The *New York Times* makes lots of money. Today's *New York Times* has an article about Jerry Garcia. My copy of the *New York Times* is rolled up into a wad. I just used it to squash a bug. (iv) Etc. What these examples show is that people are entirely used to dealing with "same" object(s?) at many different levels of abstraction.

We need to give our systems commensurate skills. In this memo, as a tiny first step, I will argue (2nd cut) that the proper—best, most natural—way to think about replication and distributed data is not, especially in the first instance, as a symmetrical, transitive, etc., "horizontal" relationship among a plurality of entities. Rather, or so at least I claim, systems and their designers should think about the issue in the same way that users (implicitly) think about it: "vertically," as *instances* (copies, editions, performances, whatever) of a single, more abstract, unity.

 Printed at 10:52 PM on August 14, 1995

2. Setup

As the New York Times example illustrates, the division of one into many can be recursive. One thing (the *New York Times*, in the abstract—the thing that is generally pretty neutral, though somewhat biased towards Israel) can give rise to lots of more specific things (each day's "issue"), which can in turn give rise to more different things (the Eastern, Midwestern, and Western "editions"), which in turn give rise to yet more (the one I used to squash the bug, others I picked up this morning for my daughter's paper route).

Here, though, for simplicity, I will consider only single cases of this *one-many fanout*. That simplification will allow me to use the following (relativist) terminology: *abstract* for the thing of which there is one; *concrete* for the things of which there are many.

One other introductory point. Note that the "fan-out" under discussion here, from one to many, is different from another more-familiar kind, of a *whole* into its potentially many *parts*. Using a term from philosophy, I will label the latter (part-whole) relationship *mereological* fan-out. The former, the one I am interested in here, I will refer to as *instantiation* fan-out.

The main example I will use will be the "manuscript" you and I decide to write after we get home from the conference. I would say 'paper', except that the project quickly blossomed into a full book—13 chapters, each with half a dozen associated files (notes, text, outline, figures, references, etc.), all stored in a hierarchical directory structure. I have a Powerbook, an office Mac, and a Mac at home. I would like to work on the manuscript on all three machines. The problems to be dealt with are the obvious ones: how can I edit, reorganize, and generally work on the files on all three machines simultaneously, coordinating disks from time to time, without everything getting all out of synch, all bolloxed up?

3. State of the art

This problem isn't new. Many people are dealing with it—from the designers of Lotus Notes, as mentioned, to programmers at Oracle, to hapless millions who try to do it by hand. Also, far from least, the Bayou folks, at PARC.

On the Mac, there are commercially available utilities whose sole function is to provide this kind of synchronization service. I've tried them all, and they all fail. Two examples will indicate their mode of failure (a third is discussed at the end of the paper):

1. Assume that files A and A′, on the Powerbook and office machine, respectively, have previously been synchronized. At work, I rename A, and move it into a different folder/directory (as part of an effort to tidy up my desktop). On the

Powerbook, I edit its contents. When I next synchronize, I end up (in most programs) with *two files* on each machine: one properly renamed and moved, but with out-of-date contents; the other with up-to-date contents, but in the old place and with the old name.

2. One day, on my office machine, as an efficiency hack, I set up some pointers to allow me to move quickly between various e-mail mailboxes. I implement these as Mac file system aliases (indirect pointers). So file A points to file B, file C points to file D, etc. Then I synchronize with the Powerbook. The process runs to completion without complaint, synchronizing A with A′, B with B′, etc. With Powerbook in hand, I head for the airport. That night, in New York, I open file A′, and try to follow its link. Rather than being led to B′, I am presented with an error message, saying that my office disk is not mounted. *Even though the synchronization routine itself synchronized* B *with* B′*, the new pointer (alias) that it created in file* A′ *points to file* B*, not to* B′. It's as if the program's right hand doesn't know what its left hand is doing!

My claim is that these programs are failing (the designers aren't implementing the "right" behavior) because they don't fully appreciate the following essential fact:

> **Principle of Abstraction:** *As a user, I don't want to think about different versions at all. I want to deal with my manuscript as a* single, coherent unity. ❶

By analogy, consider file caches. File caches keep some data in memory, data that is also stored on disk. There being two places where the data is kept, issues of coordination come up. Sometimes (e.g. in caches that don't implement write-through) the coordination can get pretty hairy. In general, though, whether supported by mechanisms simple or complex, the overall level of coordination achieved is *perfect*—not in the sense that the data in memory and the data on disk are always identical, which they often will not be, but because the person (or program) using the file system can *assume* that they are always identical. As a result, they can assume that they are dealing with a *single, unitary thing* (a file), not with multiple anythings. In sum: even though perfect on-the-run coordination isn't always metaphysically possible, the system is nevertheless able to achieve the requisite level of behavior to allow it to be treated *as if* were perfectly coordinated.[1]

[1]This paper skirts over (and was part motivated by) all sorts of wondrously complex theoretical issues. For example: it is a nice piece of homework to spell out the relationships between *existence*, *concreteness*, and *singularity*. They are definitely not the same thing. For starters, one might think that they should be opposed to non-existence, abstractness, and distribution, respectively. But even that is probably false. In computer science (and for simplicity of discussion, in this memo) we sometimes think that abstraction is opposed to concreteness.

In distributed cases, perfection—even the illusion of perfection—is not in general achievable. I hypothesize (this is pure speculation) that the reason that notions of replication, synchronization, and duplication have entered users' imaginations—i.e., the reason that users have been forced to understand the situation in terms of a *plurality* of files (mailboxes, copies, etc.), instead of a singular *unity*—is because the illusion of perfect singularity can't always be achieved.

That leads to another claim of this paper: Tough! Just because perfection can't be achieved doesn't mean that the ideal should be abandoned. More seriously:

> **Principle of Calvinism:** *Rather than give users (i.e., us) a complex, plural story that is the literal truth, give us a simple (singular) story that remains an unachievable ideal. And then tell us that the system fails. We're grownups; we can handle failure. We've seen it before.* ❷

More specifically, the aim will be to:

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

4. Details

These mandates, I claim, lead to the following model of how to think about the overall situation, depicted in Figure 1. The upper line represents the time course of α, the single, abstract object. The lower lines represent the time courses of the multiple, concrete entities β_i that instantiate the abstract one. In the current example, α would be the book or paper; the β_i would be different file copies.

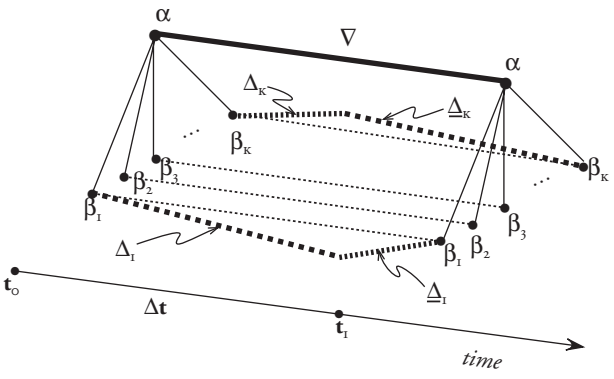

Figure 1 — Abstract and concrete entities

The following assumptions are made: (i) $\mathbf{t}_0$ is the moment in the past when the various concrete entities β_i were last, as they say, "synchronized"; (ii) during the

---

But to think of a hospital as a hospital is to think of it at quite a high-level of abstraction—far above the level of the 2×4s, for example, let alone the arrangement of protein molecules in the wood cells. Yet hospitals are not thereby rendered abstract in the sense of being immaterial, evanescent, or diaphanous.

interval $\Delta\mathbf{t}$ from $\mathbf{t}_0$ until $\mathbf{t}_1$ (now) the concrete β_i were (physically) separate; and (iii) during that time a certain set of operations Δ_i were performed on each the various β_i (indicated in the diagram as Δ_1 and Δ_k).

Given this setup, the problem we face is the following. At $\mathbf{t}_1$, "synchronization" time, a new set of operations $\underline{\Delta}_i$ need to be performed on the various β_i, in order to bring them all into synch. More specifically, given:

1. States β_i at time $\mathbf{t}_1$, and
2. Operations (transaction) Δ_i

the challenge is to figure out the appropriate

3. "Clean-up" operations $\underline{\Delta}_i$.

Two such clean-up operations are shown in the figure: $\underline{\Delta}_1$ and $\underline{\Delta}_k$.

5. Solution

The first step, given the overarching model, is to figure out what *net abstract* operation ∇ has happened to abstract entity α.

As a first approximation, one might think that this should be the sum or union of all the concrete transactions Δ_i: i.e., something like $\mathbf{U}(\Delta_i)$. But that isn't quite right. In general, the Δ_i, each of which is defined over a *specific* concrete entity β_i, 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. (We will see lots of examples below: questions about 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 link a document to).

So instead—this is where the model starts to have a technical impact—I introduce the notion of "lifting" or "abstracting" an operation from the concrete to the abstract level, indicated as '$\uparrow$'. The net abstract transaction can then be represented as

$$\nabla = \mathbf{U}(\uparrow\Delta_i)$$

Given this, each net operation that should have happened to a concrete entity β_k would be approximately the "drop" or, as I will say, *concretization* of ∇, indicated as '$\downarrow\nabla$'. But since dropping may be specific to the particular concrete instance, the concretization operation should be indexed, as '$\downarrow_i$'. So the net operation that should have happened to β_k would in fact be $\downarrow_k\nabla$.

Combining these two leads to what I will call the *Synchronization Equation*

$$\underline{\Delta}_k = \downarrow_k \mathbf{U}(\uparrow \Delta_i) - \Delta_k \qquad \textbf{❸}$$

Equation ❸ is pretty abstract. It is also *parameterized* on just about everything:

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

In a real-world example, these would all need to be defined. But still, even from this much, one can see how the alias problem mentioned above would be solved. When, on one machine, I created an alias pointing to B, the lift operation would have result in a pointer *to the abstract entity* $\hat{\text{B}}$. Then, when this operation was concretized on the other disk, abstract $\hat{\text{B}}$ would be concretized into what I called B′. So the "new" pointer on the synchronized disk would automatically point to B′, as it should.

6. Discussion

What's new in this model is the emphasis placed on *abstraction* and *concretization* operations, and their use, in a kind of "up-and-down" fashion, as a way of moving around among replicated instances. You might think that this sort of vertical thinking is more complicated than going straight across. Strictly speaking that is only true for two instances,[2] but anyway it is not the high-order bit. Much more important is the claim that it is only in terms of an up-and-down model that one can specify what kind of replication/synchronization behavior is *right*.

Making the scheme practicable would require many domain-specific issues to be worked out. Five general things can be said, of which four are relatively straightforward.

6.a. Implementation

First, regarding implementation, in a real-world setting the "abstract" entity α won't quite exist. Or perhaps it *does* exist, "always already," without our help. Perhaps it is not a *compu-*

[2]In general, for **n** instances, the up and down model grows more complicated proportional to **n**, whereas the horizontal model will grow more complicated proportional to $\mathbf{n}^2$. This becomes a serious issue only once the concrete instances start to be *different*. See §s 7 & 8, below.

tational entity. Or maybe it is computational, but is not concrete. Whatever. The point is only this: programmers building real systems will have to deal with concrete β_is, not with αs. That's true; it is also why I formed ❸ so as not to mention α explicitly. It is also okay, as α is doing its work anyway. In two ways: (i) by shaping the way we think about the problem; and (ii) as something in terms of which to define what it is for the various concrete operations to be *correct*. C'est suffis.

6.b. Aspects

Second, in any real setting, an important step in making this scheme manageable would be to divide the operations Δ_i into conceptually separable, and ideally (nearly) orthogonal, different aspects. In the Mac file system, for example, the natural operations break into 5 kinds:

1. **Create**
2. **Modify** zero or more of
   a. Name ⇐ i.e., **Rename**
   b. Location ⇐ i.e., **Move**
   c. Contents ⇐ i.e., **Edit**
3. **Delete**

These operations aren't wholly independent. Create and delete, especially, each have a kind of precedence over the others. The remaining ones, however, can be treated as independent. So, as indicated at the beginning, I ought to be able to edit the contents of a file on one machine, move and rename all the files in a grand reorganization of my desktop on the other machine, synchronize, and have the union of those two operations be the net result on both. That should work *because it makes sense in the abstract case:* reconstructed at the abstract, green, level, I simply edited, moved, and renamed *a single abstract document structure.*

6.c. Accountability

Third, I haven't said anything about what happens when the model (or rather the user?) fails. That is to say, when (for example) I edit the same document on both machines, incompatibly. In such a case, the union operation '**U**' should presumably complain. Any number of strategies are possible at this point: ask the user, privilege the most recent edit, *split* α into two different abstract entities, etc. What is crucial, though, is another deep issue lying just under the surface of this paper, relevant to the Calvinist Principle:

> **Principle of Accountability:** *when a system fails, it should fail intelligibly; when it detects or reports a failure, it should make that failure intelligible.* ❹

Another argument in favor of the two-level model being proposed here (so I claim) is that it is the proper model in terms of which to make synchronization failures intelligible. Tellingly, current designers betray a tacit understanding of this in their (English) error messages. Thus we are given "Conflict: you have edited *this file* on both sides" (emphasis added). Just so; this says exactly the right thing. But note that its *only possible interpretation is at the upper level*, because the only possible (singular) referent for the singular noun phrase "this file" is an *abstract* file. And note, too, that this is a far more intuitive message than the only literally true thing that could be said in terms of a purely horizontal model: "These two files have both been edited since the last time they were synchronized." Au fond, the fact that distinct files have both been edited is only problematic *because they were supposed to be treatable as one*; the fact that they are being synchronized is derivative.

There is a reason why the up-down model is especially well-suited to satisfying principle ❹. In general, accountability always requires *two* models: a primary one, in terms of which to define or characterize success; and a secondary one, in terms of which to make discrepancy from correctness (i.e., discrepancy from the primary model) coherent. The existence of abstract and concrete entities neatly satisfies this requirement.

6.d. Mereology

Fourth, the operations would need to be defined, recursively, over the mereological (part-whole) hierarchy. That, I take it, is the easy part. That's why we went to graduate school. Doing this would probably require protocols or APIs: so that my mail folder would recursively apply the scheme to the individual mailboxes, the mailboxes would apply it to messages, and perhaps the word-processor would be asked to apply it to individual documents (so as to generate the 'union' of a set of edits, for example). Note that, in each step down this part-whole hierarchy, the specific set of applicable concrete operations (the 'Δ_k') would likely change.

Would this be a good feature to add to the OpenDoc specification?

7. Differences

The fifth general issue is more difficult, but at the same time a (potential) source of considerable power. The "up-and-down" model supports the idea of there being *differences* among different concrete instances. Some examples (in a recursive vein, I list four that are at least relatively straightforward, and close with a fifth, that is considerably more challenging):

1. *Partial replication:* On your laptop you may want to store only the headers of messages more than a month old. Otherwise, though, you should be able to treat

these emasculated messages in ordinary ways (file, forward, copy, and delete them; include pointers to them in documents; embed them in outgoing messages). In the model, this could be achieved by defining the concretize operation '$\downarrow_{\text{laptop}}$' to throw-away message bodies.

2. *Variant spellings and formatting:* Suppose you want to store copies of your company's business plan in the New York and London offices. In this case, you define α to be a document consisting of an abstract string of atomic words, and have the corresponding '$\downarrow_{\text{NewYork}}$' and '$\downarrow_{\text{London}}$' instantiate those words with British and American spelling, respectively. The document could even be edited at either end. Changing the heading of a section in New York to read "Analyze the Role of Aluminum in Causing Alzheimer's Disease" would automatically cause the British version to turn into "Analyse the role of aluminium in causing Alzheimer's disease."

   Extra credit: You could even have a single-copy instance of such a scheme, as a model of spelling-correction: '$\uparrow$' could lift attempted spellings into abstract words, the single corresponding '$\downarrow$' could spell them correctly!

3. *Multiple languages:* Alternatively, if you were ambitious, you could use the same scheme for translation—keeping records in both French and Japanese, say. In this case operating versions of the '$\uparrow$'s and '$\downarrow$'s might need human help. (Suggestions like this also lead directly into complex issues of system *localization* or *context-dependence*, discussed in §8, below.)

4. *Versioning:* Can this scheme be applied to versions? Who knows? Papers and software might need to be treated differently. [In general, Paul Dourish thinks not. Rē software, we should ask Jeem.]

8. Context and Intensionality

A fifth example focuses on yet another way in which current schemes fail.

No one, at the moment, is fool enough to ask ordinary synchronization routines to maintain operating system software at disparate sites (e.g., to try to automatically synchronize Mac System Folders). This *should* be supportable, however. It is certainly a crying need. To do it right, though, one would need to be able to handle which I call *indexical* references—so that "the local printer" could point to the Laser Writer down the hall when at work, and to the Inkjet when in my study when at home.

In the proposed up-and-down scheme, such a strategy might be implemented by viewing the *abstract* version of the operating system α as relatively-more *intensional*, and so phrased in

terms of indexical descriptions, and the concrete instances β_i as (at least relatively more) extensional. The concretization operations '↓'s would then be charged with tying these intensional descriptions to their appropriate context-dependent values. On this proposal, that is, *concretization* would start to resemble classical context-dependent *semantic valuation*. Is that a hack, an insight, or what?

9. Conclusion

The moral? Think "vertically" about abstract entities, concrete entities, and the relationships between and among them (such as abstraction and concretization). And let the system programmers convert those thoughts, at the last possible moment, and at the lowest possible level, into invisible "horizontal" synchronization operations on replicated media.

Appendix

> *… This is intended to be filled in by a two-part technical appendix: (i) one presenting (in abstract outline) a general algorithm for recursively implementing ❸, given appropriate abstract and concrete operations, procedures to invoke on error, etc.; and (ii) an instantiation of those operations to deal with the Mac file system case, including aliases (pointers), to show how this model solves the problems identified in current commercial offerings. …*

——end of file——