New New Notes on the ChangeApplier

The previous transcript of notes on the new ChangeApplier implementation is at New Notes on the ChangeApplier, dating from March 2014 - this described the work up to and including the core FLUID-5024 implementation. Since then, as well as already discovering implementation deficiencies in the new implementation, our horizons have been opened somewhat by starting to see apparently unrelated developments as eventually convergent with the ChangeApplier and model relay implementation - especially with regards to asynchrony. This places new current and future requirements on the implementation - all of these imply that the ChangeApplier relay system will require a further rewrite. This rewrite will be quite conservative relative to the wholesale rewrite described in "New Notes" - we expect to preserve all existing semantic, configuration and test cases and conformance with the APIs which we've advertised as "public" so far. However, the handling of transactions in the system will be substantially different and we will support new kinds of change events with new members.

Problem of rotting in the implementation

Within months of the implementation of the "new ChangeApplier" it became fairly clear that the implementation had become irregular and unmanageable. This discovery was written up in FLUID-5373. A succession of new constraints were discovered on the ChangeApplier/ModelRelay implementation, most notably FLUID-5361 which required that as well as atomically notifying across the entire model skeleton, individual priority levels for model listeners should be respected globally across the tree. By the time we reached the recent requirement of FLUID-5585 (supporting "shrinkage" or removal of model material at the end of a lens) it was unclear whether we could even successfully meet new requirements - as it turned out we just managed to scrape a fix for FLUID-5585 by a non-ideal duplication of the "DELETE" change trigger separately from the two paths (implicit and explicit relay) where it was required. As FLUID-5373 writes up, the main issue straining the implementation is the inappropriate notion of "transaction" implemented in the system and forced to do double duty both to represent user transactions as well as "interior transactions" by the so-called "half-transactional system". The code paths have become very tortuous by which a frame at one end of a relay has to signal crucial information "out of band" to the other end, in order to guide its interpretation of the relaying ChangeEvents. The "DELETE" insertion mentioned above is just another example of this.

On the table as substantial ModelRelay bugs we have FLUID-5517 prompted by FLOE-230 and FLUID-5592 prompted by FLUID-5552. No substantial investigation has been done yet, but the former looks like it will be hard to resolve given the issue relates to exactly the "half-transactional updates" causing the current poor code organisation.

The still imperfectly characterised notion of a "transaction"

It become progressively clear that the very idea of a "transaction" represented an unhealthy and poorly-characterised inheritance from the old world of "centralised state". "New Notes on the ChangeApplier" describes some of the ways in which the implementation reflected ways in which our metaphors had moved on - for example, replacing "guards" with "lenses to self" eliminates the need for transactions to be "formally cancellable". However, removing the need for transactions to be cancellable by implication had actually removed another important requirement - that the visibility of updates respect the same atomicity that we apply for exterior notifications. "Should it so happen" that an observer exists who could observe a model at an intermediate point in a transaction, it would be unhelpfully surprising for them not to be able to observe the most recent state which the model had achieved. All the thinking in "New Notes" and contemporary JIRAs assumes that we need to maintain this. However, by relaxing this restriction we can both simplify our implementation as well as making it much more efficient.

Permanently live state displaces and reduces the burden for copying

"New Notes" worries about the then loss of our ability to support "thin" models - those for which we're unwilling to support the expense of a full copy per transaction. This includes both i) models which are very large, and ii) contexts which are stringent with respect to efficiency (e.g. audio synthesis). Our "visibly atomic update" model was the main thing prohibiting us from making an implementation which could "abstract over" whether we had copied the model or not - if we required no visible updates during the relay process, this invariant would require ALL ChangeApplier implementations to perform a copy. If we allow the state to be constantly live, we could produce a no-copy implementation, subject to the fidelity of "oldValue" storage. However, our thoughts on copying and the physical mechanics of change application have also moved on (see later section)

The several ways of actually applying a change

The actual mechanical process of applying a change object to a model used to be considered the most straightforward aspect of the whole process. Repeated discovery since 2008 has shown this to be fraught with subtlety - and we are now in possession of at least 3 distinct strategies for applying a change, which may be appropriate in different circumstances. The strategies can also be combined in pairs, increasing the complexity of this design space further:

- Simpleminded assignment - as operated by fluid.set, this simply attaches the value payload of the change object as the value of the penultimate property - with or without cloning it
- Cautious application - as operated by the 2014 ChangeApplier, this cautiously walks up the chain of properties notionally aligned by the change application, and only performs an action when it detects a mismatch in value or type. It then blends cloned elements from the value payload at the mismatch points.
- Immutable application - a strategy that instead of considering "immediate garbage", considers "ultimate aggregated garbage" accumulated by a system which intends to retain every revision of every model. In this case, we treat both the change object and the target model as though they were immutable data structures - with respect to the target model, we perform a "shallow clone of every node on the path to the root" of the point identified by the "cautious walk" or "simpleminded assignment".

There seem to be lots of practical advantages to the "immutable application" approach, and a "back of the envelope analysis" suggests that the garbage it generates may compare favorably to "front cloning" in many standard cases too. However, there is one classic case in which "front cloning" will always win out - the case where the value payload consists of just a single primitive value. In this case there is no cloning and no garbage - and this case is paradigmatic in many applications - for example an audio synthesis chain where the user is manipulating the slider for a single value. "immutable application" also requires a huge amount of discipline throughout the application - since the data structures representing all the model states and payloads become impenetrably interlinked, a single corruption by a stray assignment by any participant will proliferate across all the participating models causing widespread corruption. This technique is probably only workable in conjunction with language-level guards such as "Object freezing". However, if we could achieve this, it would result in the possibility for considerably efficient sharing of model objects across the entire component skeleton.

Simplified and more appropriate transaction semantics

Now we are quite clear that we must adopt (at least) 2 separate, nested levels of transactionality we need to reengineer the system to allow these to be managed clearly and explicitly. Now we can see the full extent of the system's requirements it also seems that we can simplify ownership semantics. Let us insist that each applier can have at most ONE outstanding transaction of each level, then we can store these in a publically addressible TRANSACTION STACK instead of hidden on closure frames belonging to their callers. For example, let us declare that our initial complement of transactional states are

- `user` (representing the externally visible user state managed by user transactions),
- `relay` (representing the unit of COMPLETE relay updates visible within the relay system) and
- `relayElement` (representing an elementary output from a modelRelay rule)

By moving transaction information IN-BAND into the change list, we reduce the need for the special-case signalling code, as well as making it clear how a transaction-respecting stream of change events can be externally encoded. We will introduce two new ChangeEvent types:

- `startTransaction` - indicating a rise in the transaction level - e.g. `{type: "startTransaction", level: "relay"}`
- `endTransaction` - indicating a fall in the transaction level - e.g. `{type: "startTransaction", level: "user"}`

as well as a new field on each ChangeEvent - `transactionLevel`. It would seem that we might have been able to escape the need for this field - and simply rely on the punctuation provided by the new events - unfortunately, contemplating the possibility of asynchronous relay, this is not possible. We imagine that it might be possible, with asynchronous relay, for a ChangeApplier to be in a new condition - with no active stack frame, but with a transaction still incomplete. It's unclear how exactly the implementation should respond to receiving a fresh update at a lower transaction level, but a reasonable behaviour would be for it to buffer it up until the transaction level reduces so that it could be handled. This requires us to start talking about asynchrony in general:

Getting ready for asynchrony

The possibility for asynchronous relay was just a gleam in the eye when the "new ChangeApplier" was written in 2014. Since then, there have been various developments which can be recognised to be convergent - firstly, getting a clear story with respect to promises with our FLUID-5513 "micropromises" makes it clear that we can defer a broad class of these asynchronous algorithms to our "algorithmic skeleton" fluid.promise.sequence. This became partially clear in our "Cloud-based flow manager" work for the GPII written up at Refactoring of core architecture (GPII arch list) as well as the limitations of the existing approach - part of which could be implemented by purely data-driven relay, and also the ongoing work on our DataSource implementation. This remains split, with two separate avenues of development, one in Kettle, and one upcoming in the core framework. The former is at dataSource.js (Jan 2015), the latter described by FLUID-5542. We require to converge these implementations over time - and it seems, more ambitiously, that they require to converge with the model relay system itself. Kettle's dataSource now makes use of fluid.promise.sequence in the "pseudoevent style" - that is, by using namespaced listeners with definite priorities to act as elements of an asynchronous processing chain. We can converge this with Fluid Datasources by the following steps: i) This needs to be regularised so that the "original source" itself just takes the role of a standard sequence element. This requires restructuring the input and output payloads and their availability. In practice, this will be done using the "megapayloadism" that we started to promote with the GPII refactoring. ii) Then we can position the Infusion DataSource filter as just being a sequence element that lies AHEAD of the "original source", optionally invoking it or not by making a resolving return or no (that is, optionally stalling by returning a never-resolving promise - we still need to deal with the GC implications of this and may need to encode this state separately by extending the promise contract as we did for accumulateRejectReason).

Finally, having done this, we are in a position to integrate this with modelRelay using the following elements - i) "megapayloadism" applied to the entire model state (e.g. directModel, model and options) ii) support for asynchronous modelRelay elements, iii) support for modelRelay elements with namespaces and fixed priorities activated in an asynchronous sequence. To round this off, we would treat GET and SET operations as forward and inverse arms of a relay - in this case a "concretization relay" but there would be no need to treat this separately. The wrinkle with respect to the namespace priorities is exactly the one operated by `fluid.fireTransformEvent` - the actually honoured ordering during a SET (inverse) relay would sort the relay elements into the opposite order by priority.

Implementation and meaning of "manifest transactions"

Our new model for transactions mostly does away with the idea of transaction "instances" - this is of a piece with our general thinking in making all state manifest, and not depending on networks of objects hidden irretrievably in closure frames. Under the new system, the transaction state of an applier also takes the form of state - in this case an array holding a stack of transaction records - and can be inspected by any 3rd party. This reduces garbage, and some of the requirement for separate indexing. However, *some* isolation is still required. Once we have implementation of the FLUID-5249 "global instantiator", it will become even more important to recognise that disjoint collections of components can participate in disjoint transactions. There is no need to link the transactions until the point that the two groups connect. This is not a serious risk until we support asynchronous relay - since it is impossible for distinct groups of participants to build up through purely synchronous relay. Our support for multiple transactions at the **same** model was motivated by the desire to retain the possibility for a Galois-like system (see Kulkarni et al.). However, we can still achieve this by considering that the nodes of the Galois-like system represent entire components connected by relay rules rather than elements of one component's model - and this is in fact the most appropriate model in any case. The point at which one advancing relay "touches" another is the point at which to implement the Galois-like "retreat" by the group losing an asynchrony race.

Other links to Computer Science are caused by our new treatment of transactions. With our idea of "permanently live, though transactional state", we increase the extent to which our model represents a "behaviour" in the FRP sense - see "Is the Signal Representation of FRP Correct?". And our ChangeEvent stream more closely approximates what would correspond to the "event" stream that completely summarises all changes necessary to account for the value of the behaviour. It appears there is virtually no literature on "transactional functional reactive programming" so we would seem to be on largely untrodden ground. However, with this new model we can also state that the "transactional state" of the system itself also corresponds to a behaviour, since it is completely externalisable. This was impossible under the old model whereby a "transaction was represented by an object instance".