**TheFundsChain: data redundancy and synchronization**

**Data synchronization**. DLT Technology: Hyperledger fabric. This note studies the issue of synchronizing multiple chaincodes operating together.

Synchronizing redundant data items, or avoiding to having to do so, has been at the heart of database research for almost 50 years. The whole point of the relational theory, for instance, has been to provide mathematical tools to check for redundant data.

The advent of DLTs don’t make things different. However, there is a change of scope when thinking about redundancy, as we now consider redundant data items at the level of a whole consortium of participants. In particular, considering privacy requirements may lead to redundant data items.

Further, besides actual redundancy (say, the same balance amount stored at several places), we must be aware of relationships between data, i.e. pointers or links that might become stale if no synchronization effort is carried on (e.g. changes on several items must be performed atomically).

**Problem statement with H.F**

At the heart of our issue is the chaincode transaction definition in HL. Chaincodes are assumed to be fully autonomous entities. H.F guarantees that changes carried on with one single chaincode are atomic (among other desirable transaction properties). No such guarantee is given whenever a “semantic change” must propagate across several chaincodes.

In other words, there is no H.L equivalent to the ubiquitous database statement:

Begin transaction T

Update A set x = “abc”

Update B set y = “12345…”

Commit T



No post-V1 announcement has ever been released regarding such a feature, which is understandable, given the complexity of the matter. There is one reference in the document, though, to this functionality (stating that it would be a “post-V1 feature…). JIRA records remain mysteriously silent on the matter.

Every operation that require some sort of synchronization must therefore dealt with at the application level. We provide below a couple strategies to deal with synchronization, for the following (sample) use cases.

1. A fund chaincode requires an agreement by a regulator. The regulator is not a peer for the chaincode. Every participant may consult the agreement status of a fund.
2. A portfolio chaincode holds an investor’s inventory, with holdings in several funds. The portfolio owner (peer) is not a peer of all these funds (if any at all)
3. A Fund chaincode must be able to check an investor’s KYC status but is not a peer of the distributor’s KYC directory
4. A portfolio chaincode holds an investor’s current cash position, which is not accessible to the peers of a fund. However, whenever a transaction on a fund occur, the cash position must reflect the exchange of fund shares against cash.
5. More generally, exchanging asset A against asset B without A and B managed by the same peers requires some sort of synchronization.

**Simple strategies**

The best way to deal with redundant data is probably to avoid redundancy… This may be implemented by proper data modeling and an adequate determination of peers.

**Query strategies**

Strategies based on queries are the conceptual equivalent of joins in the relational world. We discuss here several implementation strategies, namely: queries to a remote chaincode (slowest), queries to a local, managed by the same peer (reasonably fast), query cache mechanisms (faster).

The nice thing with remote queries is that only a subset of the remote state may be published through the query, thus achieving efficient data privacy.

Local queries may be used when two chaincodes are managed by different set of peers, but at least one peer is common. We like this to share public data, such as regulatory filings, with a private chaincode.

Queries may be used aggressively (e.g. query every time a data item is needed), with a cache (e.g. together with an expiry time), or on demand, with a suitable event subscription mechanism.

Valid cache information may be signed for improved trust.

*Examples*

Use case (a): the agreement is modelled as a verifiable stamp, signed by the regulator on a RegulatedFundDirectory chaincode. Only the regulator may create this item, which doesn’t have to live on the Fund chaincode state: every time the fund needs to check the validity of its agreement, it queries the RegFundDirectory to figure out the status of its agreement (Query).

*Optimization 1*: if the RegulatedFundDirectory is deemed public, queries would benefit from a local chaincode (all participants to a given jurisdiction become non-endorsing peers of the RegulatedFundDirectory).

Now if we want, say, to keep private the intermediary status of this chaincode until validation, we might encrypt data to be shared only between a fund issuer and the regulator. Alternatively, one may define two chaincodes for regulation workflows: one handling the application process (private) and one only the final outcome (public).



*Optimization 2*: suppose that for some reason, we don’t want to deploy this Directory to all participants. In this case, the query is remote (sending a query message over the network). Because this method is certainly too slow to address our needs, we may decide to cache it in the state variables of the Fund chaincode.

Best way to cache is to associate cached data with a signed stamp which expires after some time. Whenever the stamp expires, the Fund chaincode must query for a renewed one and cache it again. This works well, but fundamentally, one must trust the fund chaincode to behave properly. This is not really an issue since such chaincodes must be certified before being published on any channel.



Optimization 3: moving further on, on may imagine a subscribe / publish event mechanism to refresh the cached state upon modifications performed on the remote chaincode (e.g. the regulator invalidates the agreement, the event is pushed to the Fund chaincode…).



Again, events are nice, but difficult to trust in a generalized chaincode mechanism: trusted chaincodes and peers may collaborate with (unreliable or made reliable thanks to Kafka, it doesn’t really matter) events, but general business relationships may probably not be based on such a fragile mechanism. It is not very clear in HL documentation how a chaincode receiving an event could change its state. More precisely, the event should be sent to a maintainer of the chaincode (e.g. a peer) which would trigger a transaction… Events are not a strong point in H.L…

Use case (b): we like the idea of cached inventory to dynamically retrieve an investor’s holdings on various Fund chaincodes. This mechanism provides, in effect, a digital “depositary receipt”. Such a receipt may be communicated to other parties, and the signature may be checked by everyone to guarantee its validity. In this case, the investor’s may be left the full responsibility of updating its position cache.



Use case (c): KYC data is stored on a chaincode shared only between a regulator and a distributor. Other participants (e.g. query from a Fund chaincode) may access partial, summarized data and a proof of KYC due diligence, without revealing the identity of the investor. The result of this check may be maintained cached within the chaincode state for a while (KYC status doesn’t change overnight…). The drawback of caching appears clearly here: whenever the KYC status changes, a regulator might wish to see the new status applied immediately. Because of the cache expiry, this may be delayed for a while. Daily cache validity sounds a fair trade-off.



How about use cases for which such a compromise is not possible? (i.e. immediate action is required). Mmmh. Obviously difficult with a cache. Otherwise, events may be sent to signal a cache refresh (again, we stress that a chaincode receiving an event should not be allowed to change its state…).



**Conclusion on query strategies**

Our API framework may benefit from a common implementation of the (remote|local) query mechanism with stamp + expiry for caching.

Setting an expiry implies that the cache is deemed valid until expiry: this is okay for slowly changing data

Covered use cases:

* Local: public data such as AML black list, chartered participants and roles within a jurisdiction, other investment restrictions imposed by regulator (e.g. redemption freezes…)
* Local: all slowly changing data managed by a non-peer: registration, agreement, … (public chaincode)
* Remote: investor’s portfolio holdings (“depositary receipt”), with short expiry (next day, actually)
* Remote: investor’s KYC

As we may see above, this pattern (aka “joining chaincodes”) is very powerful and covers many use cases.

However, there are many limitations:

* All cases studied above require that only 1 set of peers actually owns (write) the shared information item
* Caching / Event-based synchronization may quickly become a bit hairy to develop, unless neatly packaged as a common utility
* Reusing signed data is useful but cumbersome, especially when publishing it to other users (will they be able to cross-check the signature?)
* In particular, use cases (d) and (e) do not fit: a more involved technology is required here.

**Complex strategy: transaction synchronization**

Use-case: let us assume 2 chaincodes of class Portfolio, Pi and Pj, keeping an account of the inventories of investors I and j, plus a cash account in asset Z. Inventories may be composed of lines of fund A and fund B (A and B are again chaincodes). A and B store a registry of their investors’ holdings. Besides, we let A and B keep a store of the available cash on their asset balance sheet.

Note that in our funds use-case, if the retained payment processing scenario only relies on oracles, without managing a cash position, we may mostly avoid this complex strategy.

Let us summarize the state of each of these chaincodes:

Pi {(xi,A) ;(yi,B) ;(ci,Z)} Pj {(xj,A) ;(yj,B) ;(cj,Z)}

A { (xi,i) ; (xj,j) ; (cA,Z)} B { (yi,i) ; (yj,j) ; (cB,Z)}



Notice how xi, xj, yi, yj are stored twice in order to isolate privacy domains: fund administrators shouldn’t be able to look through their investor’s portfolio of assets. Investors shouldn’t be able to see other investors’ holdings in a fund.

As stated before, one might decide that only A maintains xi and xj and that Pi and Pj only get a view by querying remotely A and B (“digital statement of holdings”). However, this doesn’t solve the cash issue. In this peculiar test case, we assumed that cash is managed on the ledger. If, say, i purchase some more A shares at price d, the DVP transaction would logically translate as:

Begin transaction dvp

Pi <= Pi{+zi, A) ; (-di,Z)}

A<=A((+zi,i) ; (+di,Z)}

Commit dvp

As we already stated, unfortunately, this is not (directly) supported by H.L. More generally, cross-asset transactions are not possible when asset registries are split in different chaincodes (it is still possible to create one “multi-asset” chaincode to hold all registries: cf. Fabric Composer). Remember that in this setup, we decided to split chaincodes to maintain data privacy (no single shared ledger for everything). If we decided to do so (including cash), this ledger should be “anonymized”, “scrambled” or whatsoever in order to maintain some kind of privacy (this reverts us back to the Bitcoin age, or a kind of Ethereum). It would also be the end of our flexible product model, with specialized chaincode for a given class of products. In other words, the single ledger paradigm works okay for POCs, but will prove limited and difficult to maintain when deployed on a large range of products.

Let us continue with our base assumption that chaincodes are independent and find a way to synchronize them…

1. If a synchronization of such a “global transaction” is possible at all, it will require a trusted coordinator
2. The trusted coordinator doesn’t necessarily need to know the transaction details
3. Target chaincode must maintain an additional status with their transaction workflow, as “prevalidated” (and also “invalidated” to handle global rollbacks).
4. In order to maintain serializability, unitary transaction within the global transaction must be anchored to each target chaincode’s state. This is a problem, since with an anchored state, transactions cannot be processed in parallel. In the case of several transactions, they must be ordered and settled one by one… Probably not what we wanted initially.
5. The coordinator does not need to be a peer of the target chaincodes
6. The coordinator implements the [2-phase commit protocol](https://en.wikipedia.org/wiki/Two-phase_commit_protocol) or [3-phase commit protocol](https://en.wikipedia.org/wiki/Three-phase_commit_protocol)
7. It could (but sounds scary to implement) cross different channels. Let us assume that all chaincodes involved in a single global transaction live on the same channel. So does the coordinator chaincode.

Such a mechanism could be implemented at the DLT level, or implemented by a special “coordinator” chaincode running on a specific set of nodes (not necessarily the same as the peers, in a way similar to the endorsers/orderers/committes partitioning). Kafka could play a role to secure the delivery of global transactions.

A tactical design would favor a chaincode-based implementation, relying on a ledger of anonymous global transactions.

The coordinator may only ensure that all local transactions within a global transaction have been validated by the peers of the chaincodes involved in the global transaction. It does not (should not) perform checks such as double-spending, etc…

**Musings. Alternatives to setting up a global transaction**

Here is another idea: let’s assume now that there is no cross-chaincode transaction for DVP, but a sequence of transfers on mirror entries.

In this example, chaincodes A and B maintain a cash account for each investor (ci, i). Every purchase is preceded by a credit transfer. Every redemption is followed by a debit.

A and B now supports the following operations: debit, credit, buy, sell.

Example:

If, say, i purchase some more A shares at price d, the DVP transaction would logically translates as:

A<=A((+di,i) } // prepay

A<=A((+zi,i) ; (-di,i) ; (+di,Z)} // purchase and payment

The problem now is how to transfer cash to A (same as how to transfer cash to Pi, in the previous setting), but at least, only one single chaincode is involved.

Now PTF pi may retrieve an aggregate cash + holdings position by querying the various chaincodes. Gosh! This “distributed” cash account doesn’t look much good. It could be pretty cool however, to implement corporate actions such as dividend payment (and yes, some event notification should be posted). Mmmh… Sounds pretty counter-intuitive, doesn’t it?

**Musings reloaded. Alternatives based on callback: chaincode ping-pong**

Scenario: DVP

1. Order submission
   * Tx 1: SDK client representing investor (i) interacts with PTF(i)
   * PTF(i) interacts with FUND A (remote: no common peer)
2. {… FUND reaches settlement state (i.e. NAV price revealed) }
   * Order settlement: FUND A interacts with PTF(i) (no common peer)
     + A invoke Pi : ask for cash di
     + Pi invoke A: send cash
     + A invoke Pi: send security
     + Pi invoke A: confirm settlement
     + << hole>>
     + Pi update its state

Scenario Dividend

1. Investor register its PTF(i) on fund ((i) may own several PTF…)
   * Dividend detachment: Issuer SDK interacts with FUND A

**Fabric JIRA related issues**

FAB-067. <https://jira.hyperledger.org/browse/FAB-67>