**Appendix C: consensus**

**Roles and participants**. DLT Technology: Hyperledger fabric. This appendix is a quick recap of Hyperledger transitions validation rules, or consensus.

**The state-transitions model**

With a chaincode, Hyperledger provides a persistent state of the world. State variables are versioned, and the history of previous states may be retrieved.

A “transaction” is a transition from a given state to another. Transactions do not bear any implied semantics about “financial transactions” nor are specifically linked with a notion of “currency” or token, gas, coin or whatsoever. In our documents, and to avoid further misleading terminology, we’ll speak of “transitions”. We shall use the term “transaction” when implying atomicity properties and concurrency control mechanisms, e.g. the semantics of a transaction in classical database systems.

<http://hyperledger-fabric.readthedocs.io/en/latest/txflow.html>

In order to reach a new valid state, a transition must be validated by the different peers maintaining a copy of the state of the chaincode. This process is decomposed in 3 steps: endorsement, ordering and validation.

1. Endorsement: basically, some peers – the endorsing peers - accept the transition proposal transmitted by the SDK client. Accepting a proposal means executing the corresponding chaincode method to simulate its effects.

The exact meaning of “some peers” is defined by the endorsement policy defined when deploying the chaincode (e.g. at least one endorser, all of them, etc…).

Basically, the endorsement step is designed to prevent, as early as possible, the intrusion of fake, spoofed or malformed transitions.

Endorsement may also set up state version dependencies, to manage concurrency control, that is allow the validation process to check for the serializability of transitions. This is called *anchoring* a transaction.

Transitions proposals that do not pass the endorsement step are not recorded in the chaincode’s ledger.

1. The SDK client collects all needed endorsement signatures then sends the proposal to the ordering service
2. Endorsed proposals are checked by the ordering service, according to the endorsement policy (**ESCC**)
3. Ordering: transitions on a given chaincode are pipelined in a guaranteed order, thanks to an ordering service provided by other participants sharing a common ordering service channel (several such channels may be deployed on the network). The ordering service implements a byzantine-tolerant consensus algorithm. The default algorithms proposed by Hyperledger are: NONE (trivial algorithm) and PBFT (Practical Byzantine Fault-Tolerance), which require at least 3 participants.

The ordering service is not designed to implement business rules. The ordering service acts as a zero-knowledge proof mechanism: ordering participants (“orderers”) have no knowledge of the contents of a transaction. The ordering service may be customized. However, designing a tailor-made Byzantine Fault-Tolerant algorithm is no easy tasks, though.

1. Validation: committing peers eventually validate the ordered transitions with a **VSCC** method and proceed to the changes in the state variables. Hyperledger takes care of maintaining the ledger and producing the proper database blocks. At this stage, chaincode specific methods are not more used: only the results from the endorsers are used by committing peers.

Note: **ESCC** (endorsement system), **VSCC** (validation system) and **QSCC** (query system) are part of the “system chaincode” and may not be customized.



Figure 1: endorsement



Figure 2: high-level transaction workflow

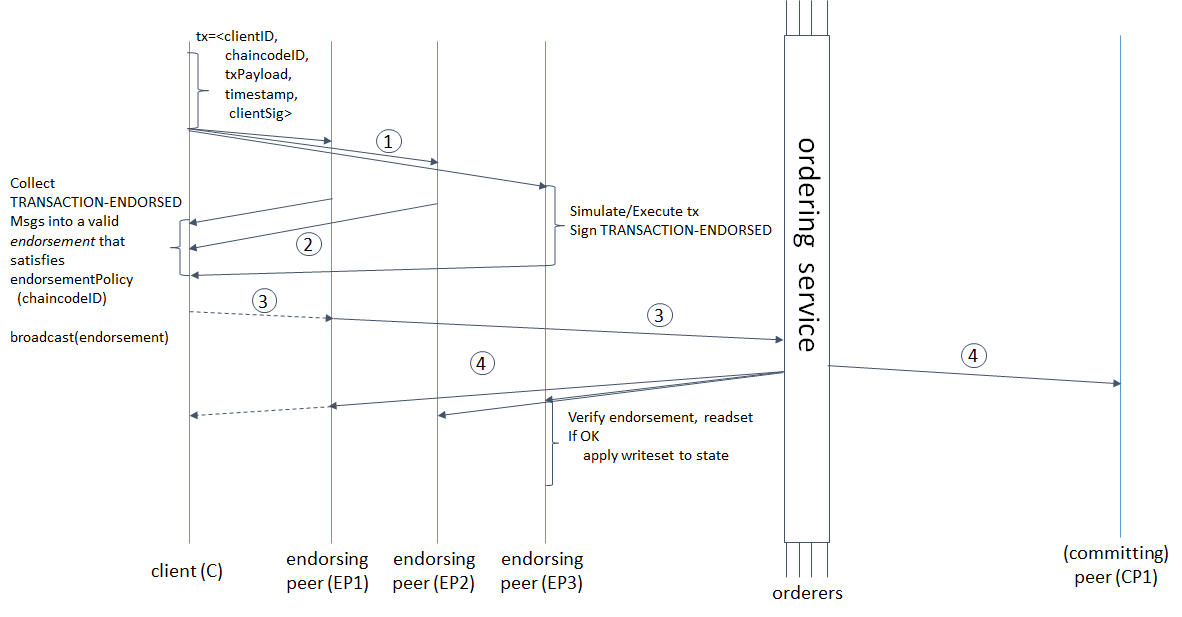


Figure 2: detailed transaction workflow

**Immutability**. Transitions are immutable, in the sense that they are kept in a ledger, cryptographically linked as a “blockchain”. Any change to the chaincode state must pass through these steps (changes initiated directly from within a chaincode at a peer’s node skips the endorsement step).

The immutability of the ledger guarantees that the current state of a chaincode may always be traced back by applying the ordered set of past transitions to the initial state.

Immutability provides the guarantee that “nobody may alter the past”, but this property remains neutral regarding business semantics: the “past” which may not be altered is the database past, not necessarily the business past. It is thus for instance perfectly possible to change a “value date” state variable which has some business implication, while remaining consistent. For example, it could be possible (but not necessarily practical) to modify a NAV at a past value date (and trigger the subsequent adjustments on investor’s holdings).

**Workflow modeling implications**

1. **Business consensus on the result of a smart contract must be achieved by endorsers. Consensus rules may be too complex to be expressed in terms of endorsement policy rules: this must be provided at the application level.**
2. **If any oracle must be provided to a smart contract, the peer maintaining the oracle must be an endorsing peer.**
3. **If some of the endorsers are oracle, and some are not, they must be able to introspect their own role during chaincode execution to execute different checks**

**Database modeling implications**

Conceptually, the state of a chaincode may be arbitrarily complex. It may even be a full-fledged database, with technical structures such as indexes (further testing must be carried on about practical size limits of state variables). In other words, a chaincode is not, even loosely, related to a relational table.

**Chaincode should be thought as an autonomous set of data with a set of change management methods.**

*Pluggable data storage: Hyperledger is designed with a pluggable data storage component. In practice, this is not so easy. Base configuration comes with a LevelDB persistence engine (simple key-value semantics) or, optionally, a CouchDB engine (JSON-aware storage, and simple, NoSQL query language).*

Likewise, transitions may contain arbitrary content. We are not constrained by a ledger recording the same kind of changes: any change is audited then “frozen” on the ledger. Sadly for hard-nosed Bitcoin enthusiasts, the ledger is a much more complex thing than a record of debit/credits on a bank account. The concept of “double spending” is irrelevant in our context, or more precisely, if application designers want to implement such a check, they may do so: such a feature is therefore application dependent and not enforced by the fabric itself.

For other application designers, this model provides a useful framework to design workflows involving several participants (SDK-clients, peers). This corresponds to most business use-cases.

Chaincodes are deployed by peers as sealed, certified, containers (the underlying technology is Docker), and ensure that the same logics are applied by all peers. However, this is the consensual validation mechanisms which really guarantees that no peer has been compromised.

*The question of trust: every participant to the fabric is identified (possibly with a pseudonymous identity). Peers are collectively trusted by SDK-clients. Endorsers are trusted by SDK-clients. Peers trust each other up to a certain point only: they are able to cross-check the others attempt to modify the state of a chaincode.*

**Data redundancy, at the chaincode level**: persisting both state and transitions is a strong redundancy case. Such redundancy is required to guarantee the validity of the state. Consistency is maintained by Hyperledger. Consistency issues which arise under concurrent changes situations are addressed thanks to a serializability mechanism. It is up to the application designer to specify the various version dependencies (or “anchors”) which may be used by the underlying system to check for serializability.

**Hyperledger limitations**

* **Endorsement**: the documentation is quite obscure on this point.

In order to understand, we must understand the different steps in endorsement:

1. **Delivering an endorsement signature:** this is performed by the chaincode method called upon Invoke. Any code may be executed, including checking external resources
2. **Checking that a proposal has gathered the proper signatures:** this is performed when posting the proposal to the ordering service. This check is explicitly expressed with a logical expression from endorsement policies. The richness of such expressions is limited, because the ESCC service must take a local deterministic decision and must not change other chaincodes, even on the local peer.

**Endorsement policy.** The endorsement policy (“required signatures”) is defined at the chaincode level and is therefore the same for all supported transitions for this chaincode. This means that we cannot specify different endorsers for specific transitions.

Example: we cannot specify that changing the NAV should only require the endorsement of a limited set of peers, while posting an order require other peers for endorsement.

A possible work around could be to proceed to different checks in the Invoke method, according to the peer’s functional role and the transition method invoked.

Indeed, this kind of workaround is mandatory to implement an “oracle” in Hyperledger.

Endorsing policy expression: the policy is expressed in terms of MSP organizations and roles. MSP roles (admin, member) are only roles related to the administration of keys and not functional roles.

<http://hyperledger-fabric.readthedocs.io/en/latest/endorsement-policies.html>

Example: AND('Org1.member', 'Org2.member')

Roles are limited to ‘member’ or ‘admin’.

* **Chaincode transaction execution method**
  + A transaction method may query other chaincodes. Queries on the same peer may be quickly performed, bypassing the SDK (direct access to chaincode on the peer). Queries on chaincode maintained by other peers may use the Golang SDK within a chaincode (<https://github.com/hyperledger/fabric-sdk-go/>).
  + Transaction method may interact with external platforms (e.g. legacy systems). The natural way to do this is to use the gRPC protocol.

Other ways seem workable too, such as using Hyperledger custom events. Time out management on such operations remain to be tested, though. Application developers must be especially careful about the deterministic property of such an external call. For instance, let us suppose that we use such a call to check a cash account balance on a legacy bank system. Simply verifying the *current* balance is not enough, since simulating the validation at different times may produce different results with the same input. A correct implementation would be to query the current balance *at a given time*.

* + **Major limitation: changing multiple chaincodes**. A transaction method cannot propagate the context of the current transaction to start another change in a chaincode. This means that atomic changes cannot be carried on with several chaincodes (e.g. cannot commit a transaction spanning several chaincodes). This limitation is acknowledged as temporary by the Hyperledger team (but post-V1 anyway).

It is possible (and demonstrated by available test program), to change another (local) chaincode from within a chaincode. The latter however, is not modified.

It is possible to start a new transaction from within a transaction, thereby implementing nested transactions, but atomicity is not guaranteed.

**Keeping several chaincodes synchronized is thus difficult**. A workaround for this is to put the burden on the SDK client, which would be responsible for restarting any failed phase of the transaction. This is obviously not perfect. A natural consequence of this limitation is to pack data as much as possible within a single chaincode, which in turn may cause some data privacy concerns.

* **Privacy**
  + All peers of a given chaincode may be able to see the full state of this chaincode.
  + Therefore, privacy may be achieved in two ways:
    - Encrypted state variables
    - Separate chaincode, with different sets of peers
  + Confidential transactions: to be investigated

**Note: post-V1 plans regarding privacy:**

* + Chaincode calling chaincode within a confidential domain
  + User-based confidentiality
  + Peer-based confidentiality
* **Notification**
  + The outcome of a transaction is returned to the calling SDK-client. It is possible to use events to carry on more complex notifications.

This is not an easy task, though: the Hyperledger events systems is really designed for peer-to-peer communications on database blocks maintenance and the “custom event” feature is poor and clumsy to us. A provided test example shows how to set up a “chatter channel” between chaincodes.

* + “Waking up” a SDK-client: e.g. the chaincode informs a SDK-client that some action has been performed, for instance requiring some new transition to be initiated.

Example: an issuer has just submitted a registration agreement request. The regulator should now examine it and give a status to this request. Since such examination may not be fully automated, the regulator should initiate a new transition to settle this case.

For go SDK, see <https://github.com/hyperledger/fabric-sdk-go/blob/master/api/peer.go> and the associated unit test.

For nodeJS SDK, see: <https://github.com/hyperledger/fabric-sdk-node/blob/master/test/integration/events.js>

* **Functional roles**: Hyperledger defines roles according to technical requirements (i.e. endorsing peer, committer peer, orderer). This typology may not be mapped in a straightforward way to functional roles (e.g. accountant, issuer, registrar…). Applications must therefore provide their own, custom, role management.

**Consequences on TheFundsChain modeling assumptions**

*State model for a fund*

Our pivotal “fund” chaincode must be an autonomous object with most required data managed in its state, i.e. as much as possible within data privacy requirements. Its state is thus described with:

* Fund master data (legal, operational and fiscal undertakings)
* Detailed anonymous investor holdings (or registry), with possible earmarking for distribution
* Possibility to extend with asset inventories

Privacy requirements require that:

* Workflows in which a regulator and some other authorities participate (registration, jurisdiction distribution agreement) are handled by a separate chaincode (“regulated funds directory”)
* Workflows in which distributors participate (investor onboarding, distribution agreement, investor portfolio) are handled by a separated chaincode (“investor”, “portfolio”)

*Validation model*

**ESCC usage**: ESCC default behavior (i.e. checking the signature of the SDK proposer) is enriched with the following checks:

* Valid functional role for the proposed transition (e.g. only accountants may propose new NAV’s)
* Well-formed JSON message (schema validation)
* Other surface-checks, extending JSON-Schema 4 validation semantics (e.g. transaction type and sign are consistent)

Since ESCC itself is not customizable, such extensions should come as a common utility method called from within chaincode transaction validation method.

**Chaincode transaction method**: there is no default behavior.

* VSCC code depends much upon the type of transaction
* Example: orders
  + Query and verify investor KYC data (managed by a distinct chaincode, managed by other peers)
  + Checks for available units upon redemption (ah: here is this “double spending” feature!)
  + Check for possible investment limits upon subscription
* Example: NAV publication
  + Check published net assets vs units x NAV
* Example: fund registration
  + Query regulator’s directory chaincode and verify the various items signed by the regulator and other authorities (LEI, ISIN…)
* Example: orders settlement
  + External check: cash account verification on legacy system
  + External check: incoming SWIFT messages verifications (SWIFT gateway ~ legacy system)

*Most repetitive queries required by external checks may benefit some cryptographic sugar. Our design proposal is to compute a signed “seal” on data to be checked thereafter. The actual query occurs the first time to retrieve the seal, which is stored on the querying chaincode then locally reused afterwards. Seals may be set to expire (much like a certificate) to force the chaincode to refresh seals from time to time.*

**Ordering service**: default is Kafka

H.F V1 alpha: PBFT has been removed to put forward the Kafka ordering mechanism. Kafka provides a fault-tolerant ordering consensus, but not a Byzantine-fault tolerant one. PBFT status is still under development and announced for V1 “Nest”.

*Need for a trusted global timing service*

Ensuring a deterministic behavior is not always easy, especially whenever workflows are time dependent. This is the case with some of our workflows:

* Orders workflow depends on a time event: the fund’s cut-off (depends on the fund, the time zone)
* Some asynchronous checks require a reliable *and deterministic* timeout management: settlement
* External checks such as verifying cash balance on a legacy system must be based on deterministic timings

Therefore, we see the need for some pivotal, trusted timing service available to all platform participants.

Note that a trusted source for time is also often interesting for cryptographic operations which could be performed at the applicative level (such a delivering “seals” and “earmarking”).

Alternative to chaincode deployment: use a Zookeeper time feature.