**TheFundsChain - The Object – Chaincode Directory pattern**

**Objectives**

The Chaincode directory pattern is essential to publish lists of available chaincodes to a given usage.

Chaincode directories generally only store a list of references to other chaincodes and publish proxy methods to access their data.

This is the simplest use case of chaincode directories: whenever the directory needs more information about the referenced CC, a query is sent. On many occasions though, it may be useful to store a part of the state, for instance to allow searches or simple filtering with the directory. In this case, a subpart of the referenced CC state is duplicated in the Directory. Duplicated information must be kept in sync.

Moreover, some directories may be enriched and manage their own state: for instance, the Regulated Fund Directory derives from the generic chaincode directory and manages its own workflows and internal state. The fund distribution agreement is also based on a directory. The bottom line is that Directory chaincode is a behavior *pattern*, not a predefined feature.

In essence, chaincode directories define pointers to other chaincode. The protocol described in this document is designed to avoid dangling pointers situations. In other words, we implement a pattern of *intelligent pointers*.

Like pointer indirection, we shall support indirect references, i.e. Directories of Directories.

Our utilities library shall provide with a reusable API to easily scaffold this pattern into a chaincode.

**Concepts**

*Subscription*: whenever a Directory chaincode adds a reference to another chaincode, it explicitly sends a subscription request to this chaincode's state. This request may be denied if ACL are not properly set.

*Subscribers*: a chaincode supporting external references by directories maintain a list of subscribers. Each subscriber may be entitled to different queries / forwarded invokes. The list of subscribers is thus part of the state of the referenced chaincode.

*State change forwarding*: whenever some action is performed on the referenced chaincode (such as: Open(), Close(),Transform(), state changes are pipelined to all Directories which *subscribe* to this chaincode. Updating Directories must be an atomic operation.

Whenever a new Directory chaincode is deployed, it should post subscribe requests to all chaincodes it is supposed to follow up. This list may change over time.

*Forward subscription*: to be able to start a Directory from scratch, knowing only other directories, a protocol must be established to transmit subscriptions from Directory to Directory: this is the forward subscription.

Example: a distributor wants to distribute funds from a issuer. The issuer provides access to its own FundDirectory chaincode. The distributor then sends a “forward subscription” request to the issuer’s directory, which returns all its registered chaincodes.

*Atomicity of updates*: synchronization stressed the important part played by the “cross-chaincode” feature is to forbid transient inconsistent states.

**Abbreviations**

CC: a chaincode object

CCDir: a chaincodes directory chaincode

Subscribed state: the part of cc state shared and synchronized with ccDir

Subscribers list: the list of (CCDir, SDK) which have subscribed.

Additional info: subscribedStateDescription mask + changeNotificationStatus

Clearing: coordinating cross-chaincodes transactions.

This protocol is assumed to be deployed and is described here as an abstraction (begin tran – commit)

**Protocol**

*Subscription protocol: establishing a list of CC for a CCDir*

* CC methods:
  + Invoke: Subscribe(subscriber,forwardingOption,subscribedStateDescription)
  + Invoke: Unsubscribe(subscriber,forwardingOption)
* CCDir methods:
  + Same as CC methods (to support forwarding)
  + addSubscribedMember()
  + removeSubscribedMember()

Note on forwarding: forwarding just passes on the incoming subscription message to the destination chaincode, without altering credentials. Forwarding is not to be mistaken with "subscribing on behalf of…", which is not supported (for the moment).

Messages content outline:

* subscriber:
  + credentials to authenticate the sender
  + chaincode ID of CCDir
  + URL of the companion SDK application
  + Human readable description of the subscriber
* subscribedStateDescription
  + a JSON structure, compatible with the CC state schema, used as a mask: all required data are filled with default values, other are empty, blank or filled with dummies for mandatory structures
  + schema should always define a mask structure with same structure but simplified types and no mandatory attributes (such a mask schema should be generated)
  + mask schemas are also used for entity-based ACL

State synchronization protocol:

* CC methods:
  + Query: querySubscribedState()
  + Event emission: NotifySubscribedStateChange(event) => to subscriber companion SDK
  + Invoke: subscribedStateChangeGotIt(state ID)
* CCDir methods:
  + updateStateFromMember() => triggers a state query on CC

*Subscription: registering a new CC member in CCDir:*

begin tran

CCDir owner SDK calls CC:Subscribe()

CC registers CCDir as a subscriber; returns current subscribedState and initialize its subscriber's status to 'synched'

CCDir owner SDK calls CCDir:addSubscribedMember()

Commit /\* clearing boundaries \*/

*Removing a CC member from CCDir:*

begin tran

CCDir owner SDK calls CC:Unsubscribe()

CC removes CCDir from its registered subscribers

CCDir owner SDK calls CCDir:removeSubscribedMember()

Commit /\* clearing boundaries \*/

*Synchronizing subscribedState:*

* on change of the subscribedState:
  + CC post event to all its subscribers' SDKs (NotifySubscribedStateChange to CCDir's SDK)
  + Mark subscribers as "notified"
  + CCDir SDK consumes event
  + CCDir SDK invokes CCDIR:updateStateFromMember()
    - CCDir query subscribed state from member CC (query: getSubscriberState())
    - CCDir updates its member state from queried state
  + CCDir SDK invokes CC:subscribedStateChangeGotIt(CC:state ID/timestamp)
    - CC mark subscriber as 'synched', with the latest state ID (or timestamp) synchronized
    - If the "GotIt" timestamp refer to a state that has already been changed, a new event is posted.

**Notes:**

State synchronization is based on events which are not persistent and may be lost: subscribers in a 'notified' state get periodically reminded that they should acknowledge sync. Events are posted to the SDK engine (a nodeJS listener).

After some timeout (expressed in days or weeks…) periodical reminders cease to be emitted and the subscriber moves to status 'stalled'.

A ccDir may force a refresh by subscribing again with the same state description: subscriptions that are already registered simply result in a subscribed state refresh.

A ccDir may change the description of its subscribed state by subscribing again: subscribed state description is simply updated by the referenced CC.

Upon boot, ccDir should always force a full refresh, to move out from possible 'stalled' state if they haven't been active for a long time.

getSubscriberState() returns a snapshot of the CC at query time. Changes may have occurred between notification and this query: latest state is returned anyway. New changes may occur and new notifications posted BEFORE the ccDir had a chance to update: if changes are frequent to the point of halting the ccDir (does nothing else than trying to keep up with changes), state update queries may be delayed or better, subscription does not specify to be notified with state changes but operates by polling instead, using getSubscriberState() on its own initiative.

Examples:

* Promoter list of issued funds
* List of regulated funds
* Universe of distributed funds
* List of funds subject to a distribution agreement
* Aggregate list of PTF belonging to an investor
* Aggregate list of PTF belonging to funds for a custodian
* …

Mark the possibility to break consistency, since this is a client-initiated transaction. Restart procedures should always involve a full resync.

SubscribedStateDescription should usually refer to slowly changing properties (suited to master data changes). Subscriptions should occur relatively infrequently. This way, a single CC could manage many subscribers (say, dozens)

For frequently changed data (for instance, subscribing to real-time total AUM of a fund or daily cumulated inflows), here are some suggestion:

* Subscription then polling state, without notification and callback
* Polling using a dedicated query

**API for Chaincode Directory pattern**

API requirements:

* Import struct for the JSON mask schema of subscribed CC(s)
* Import base ccDir state structure
* Import optional payload structure for ccDir

API capabilities:

* Declare a registered CC (there may be several)
* Declare support for ccDir registration
* CC side: registration methods, subscribed state sync methods, subscription clearing support
* Additional state variables for subscription
* ACL support for all methods: extends local CC ACL structures

**Security**

* All methods published for this protocol are subject to ACL (see design doc Authorization)
* ACL may include a CC schema mask to override subscribed state descriptions (with a AND mask operation)
* Events reception is also secured
* All communications are under SSL (including events)

**Byzantine faults**

Byzantine faults with this protocol refer to attempts by either CC, ccDir or companion SDK to "cheat", intentionally or because of malicious behavior of a compromised code.

Here are a number of such "byzantine" scenarios.

* Subscription spam (DOS-type)
* Notification event spam
* Do not comply with clearing protocol
* ccDir does not update after notification or subsequent reminders
* CC sends falsified/wrong state