Hyperledger Fabric SDK Design Specification v1.0

2016-11-21

Baohua Yang (IBM)
Jim Zhang (IBM)
Pardha Vishnumolakala (DTCC)
Muhammad Altaf (Fujitsu Australia Software Technology)
Patrick Mullaney (LSEG)
Binh Nguyen (IBM)
Kai Chen (IBM)
Tim Snyder (CLS)

Table Of Contents

Revisions

Date	Revision	Description	Author
8/28/2016	0.1	Uploaded draft with initial APIs	Baohua Yang
8/29/2016	0.2	Added introduction	Jim Zhang
8/29/2016	0.3	Added goals and formatted the content	Pardha Vishnumolakala
8/30/2016	0.4	Revise the API	Baohua Yang
8/30/2016	0.5	Added Design section	Muhammad Altaf
9/6/2016	0.6	Added fabric v1.0 content	Jim Zhang, Baohua Yang
9/8/2016	0.7	Add set_crypto_functions to member_services	Muhammad Altaf
9/21/2016	0.8	Add scenario section; Update according to comments.	Baohua Yang
9/21/2016	0.9	Add decryption and signature algorithm details ; Add eventhub content; Add CryptoSuite interface	Muhammad Altaf Patrick Mullaney Kai Chen
11/8/2016 - 11/15/2015	0.9.1	Added diagrams Revised scenario sections Revised APIs Moved COP client design out	Tim Snyder Jim Zhang Muhammad Altaf
11/21/2016	1.0	Ready for TSC to review	SDK work group

Dependencies: this document depends on various components including the Peer, CA, BCCSP (Blockchain Crypto Service Provider) and Ordering Service. This is a living document that will change over time.

1. Overview

Hyperledger Fabric v1.0 provides basic API using Protocol Buffers over gRPC for applications to interact with the blockchain network. The API covers transaction processing, security membership services, blockchain traversal, and event handling. There are many languages supporting Protocol Buffers, including Go, C#, Java, Javascript, Python, and C++. So it is desirable to provide a native software development kit (SDK) for application developers to use.

This document specifies the minimum set of API that an SDK implementation should provide. The goal of an SDK is to provide both the primitives to access the client-facing features in the blockchain network and reasonable level of abstraction on top of that to make developers' life easier, in the native language that the application is written in to ease the development effort. However, using an SDK doesn't prevent the application from calling gRPC directly.

Note that the current REST API is being deprecated, and SDK should not be built using the REST API. The reasons for going with gRPC over REST are 1) ability to control the flow with full-duplex streaming (most of API calls are asynchronous), and 2) better performance. Furthermore, the Peer doesn't have to open up more ports for HTTP/HTTPS since internally we also use gRPC for inter-component communication.

An exception is that the new member service implementation (CA) exposes a REST API.

For reference, the Fabric v1.0 architecture and interaction model can be found in the attached documents in this work item: https://jira.hyperledger.org/browse/FAB-37

2. Goals

1. **Application development**

   Fabric SDK should enable developers to write applications that can interact with the network in various ways. Applications may deploy/invoke chaincodes, listen to the events generated by the network, retrieve information about the blocks and transactions stored in the ledger etc.

2. **Chaincode development (not the focus of v1.0 of this spec)**

   Fabric SDK should enable developers to write and unit test chaincode. Developers should be able to quickly test chaincodes without needing to deploy them to the network.

3. Principles

1. **Well documented api(s), data models, and sample code**

SDK should provide clearly written documentation regarding the available apis, data models, and examples illustrating how the apis can be used.

2. **Ease of use**

   Chaincode developers and application developers are concerned with writing business logic. Even though it is advantageous for developers to be familiar with internals of the Fabric project, it should not be a prerequisite. As such, SDK should not have any compile time dependencies on the Fabric project (other than the proto files that define various contracts). SDK packages/jars/libraries should be made available through well known repositories so that developers can easily install them and start writing chaincode developing applications right away.

3. **Performance**

   The SDK MUST be carefully designed to facilitate high throughput, horizontal scalability and low latency. It should be ideally implemented as a stateless component, or otherwise allows state to be shared across application instances via a database.

4. **Versioning**

   SDK implemented by different languages is suggested to follow the same versioning, and various implementations of the same versioning should keep compatibility in functionality with each other.

   For example, fabric-sdk-node 1.0 and fabric-sdk-py 1.0 should support Fabric 1.0 release, and have the same exposed functionalities like membership operations and transaction processing.

5. **Serviceability**

   The SDK should make it easy to plug in serviceability support, namely logging. It should allow the consuming application to set an instance of a logger. This is useful because an application would likely want to use a common logger for all parts of the code (inside the SDK and outside). And typically an IT organization would have log scraping setup for monitoring and analytics purposes, such that a “standard” log format is desirable. The SDK should have a built-in logger so that developers get logging by default. But it MUST allow an external logger to be set with a standard set of logging APIs.

4. Scenarios

There are many possible scenarios, but let’s focus on a few to illustrate the overall capabilities of an SDK.

Membership Registration and Enrollment

In order to interact with the Fabric, the application must have a proper identity, which is encoded as part of a membership certificate that we call ECert (Enrollment Certificate). This identity may be provided by an external CA (Certificate Authority) or Fabric Membership Services (Fabric-CA), as long as the ECert can be validated by the Fabric components during processing to establish a chain of trust.

If a standard CA is used to manage the distribution of ECert, then transactions will be signed with ECert. This makes each transaction identifiable to a client identity.

On the other hand, the member service provider (MSP) designed for the fabric makes use of a crypto algorithm that derives a key pair from the key pair of the ECert, thus producing a pseudonymous certificate for each transaction, aka TCert. A TCert doesn't represent identity but can be traced back to the originating ECert via the member service. This technique is useful to hide transaction identities in many business scenarios.

Fabric provides an implementation of the MSP interface, which is called "COP" (name is inspired by it's police-like functionality, not an acronym). It is in "fabric-cop" repository under the hyperledger organization in github.com.

If a Fabric-based network is configured to use an external CA, then the user registry would also be maintained externally. The external user registry will be responsible for authenticating the user. An authenticated user can then request for enrollment with the CA in order to obtain the ECert.

As far as the SDK is concerned, there should be two types of APIs: *common APIs* for any CA (built-in or external), and *APIs specific* to the Fabric's built-in member services. The common APIs are responsible for allowing the consuming application to perform the most essential operations as related to user identities, namely "enroll()", which is the process to obtain ECert for the authenticated user, and "getTCerts()", which obtains transaction certificates to allow the user to submit transactions.

The design for the client of the optional member service implementation (aka COP) is described in a separate document.

Chain Construction

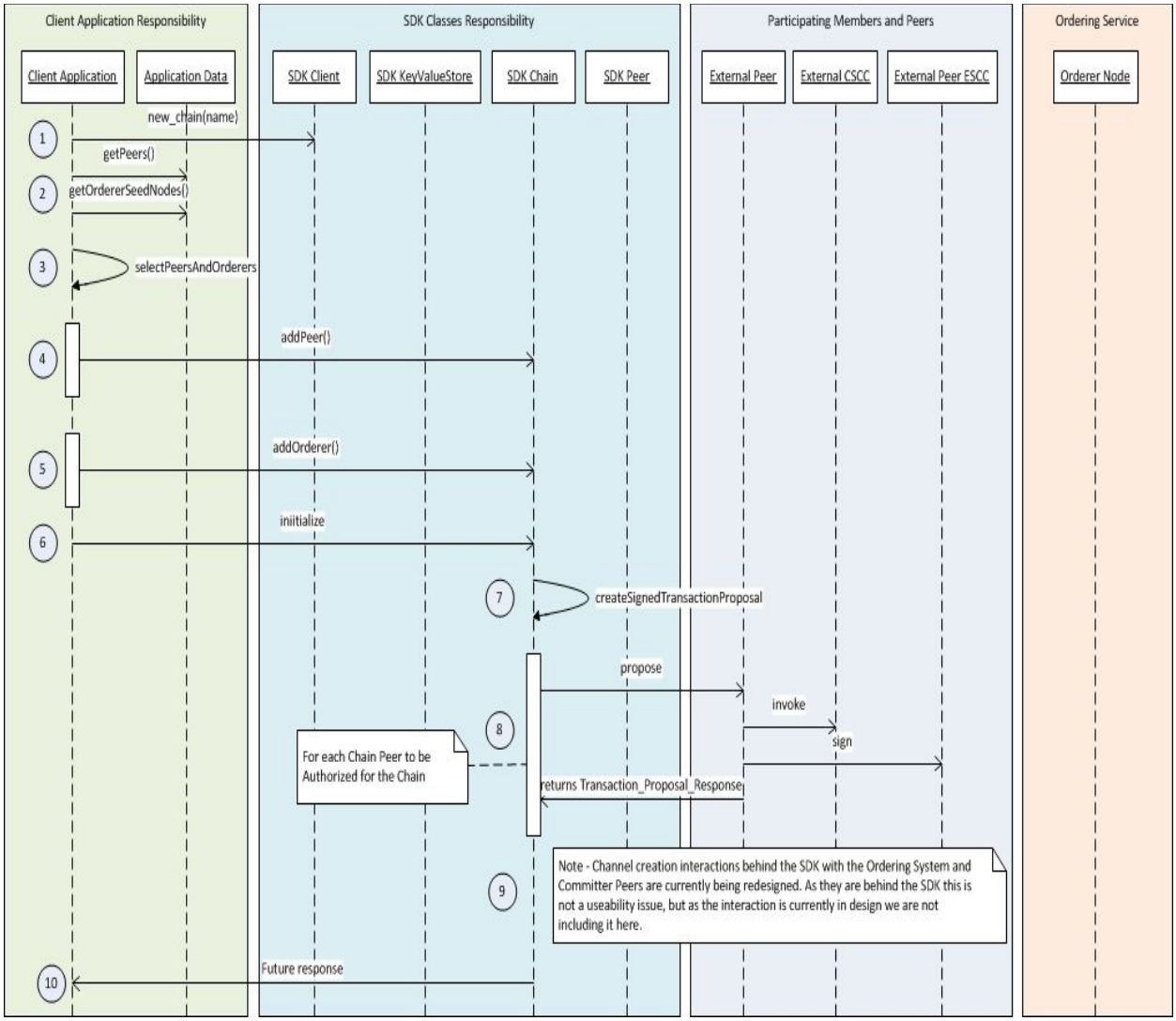
The Hyperledger Fabric supports privacy and confidentiality via the channel and ledger design, which collectively is called a "Chain". Communications (data on the wire) are confined within the channel between the participants, while the blocks (data at rest) are saved on a private ledger

distributed only among those participants. Peers on the network that are not part of the chain are not aware of anything inside the channels and private ledgers, nor do they have access to them either.

Once a chain is constructed, the application can send transactions to the Peers on the chain in a private manner and have the validated transactions committed to the private ledger.

The responsibility to create chains falls on the application. Through the SDK the application initiates the formation of a chain with the group of organizations (representing network membership) and the orderer service.

In the SDK, the combination of a channel and its associated ledger are represented by the Chain class. The application works with the designated orderer(s) to first prepare a new channel and obtains a genesis block that contains key information about the new chain, including the participant information (URL and certificates), orderer information (URL and certificates). The application then coordinates the invitation of the participating Peers to the new channel via a Configuration Transaction that targets the Configuration System Chaincode.



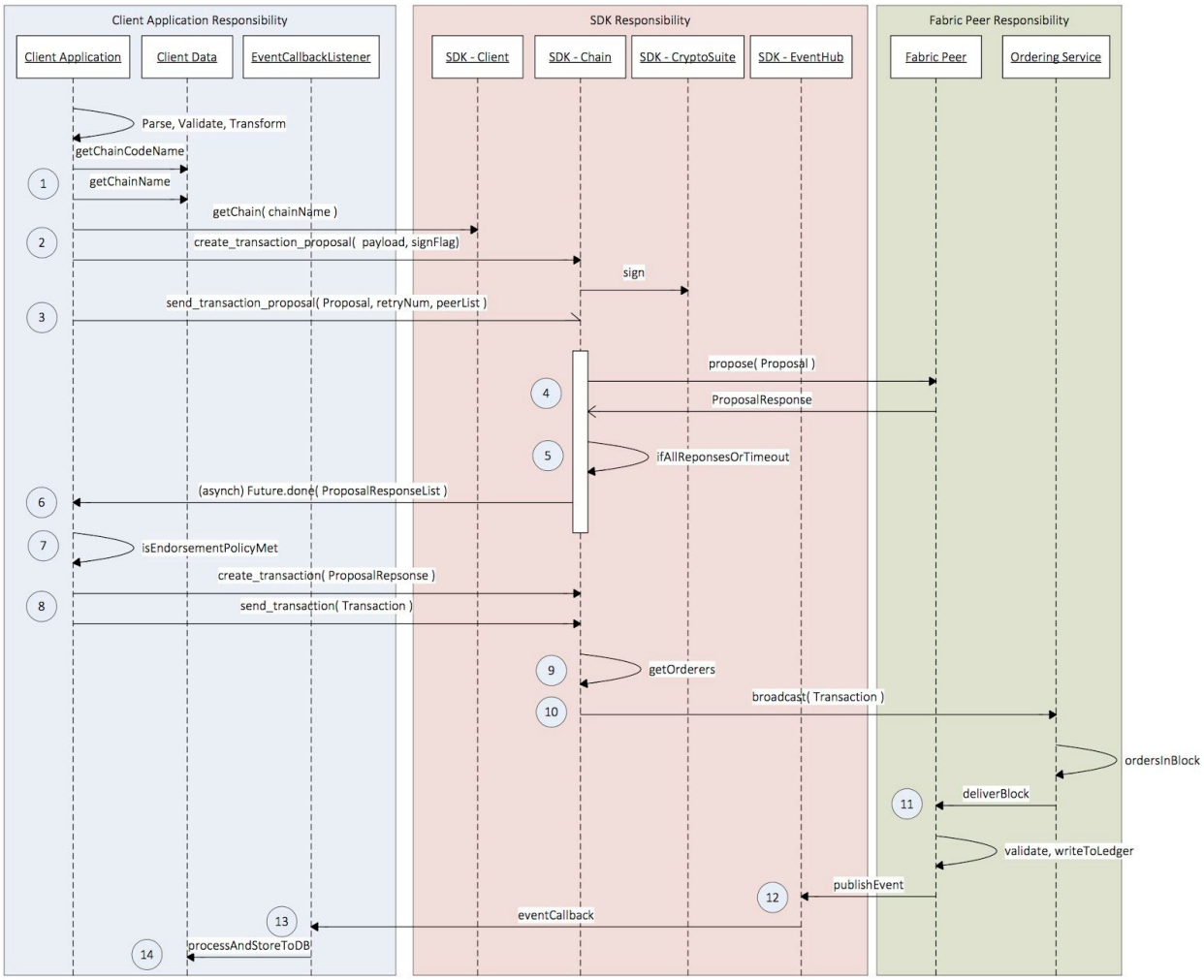
Transaction Support

A transaction involves 2 explicit steps: Endorsing and Committing. Endorsing is to request Peers to run and endorse (sign) the result of the transaction; Committing is to request consensus on the validity of the transaction and its endorsements.

The following diagram illustrates the interaction between the client SDK and the Peers on a transaction. See the architecture document for more details.

First the SDK, working with the application, signs the message with the application's (or the authenticated user's) private key (all external messages require signature authentication). Then it sends the message to 1 or more Peers according to the endorsement policy (implemented through Validation System Chain Code or VSCC). It gets back the responses asynchronously and decides whether or not to proceed to create and submit the transaction to the consensus

service. The decision to proceed with the transaction submission is based on the endorsement policy predicates (like 2 out of 3) based on out-of-band knowledge. Once submitted, the processing of the transaction is asynchronous, so the SDK listens to the commit event to inform the application of the completion or rejection of the transaction processing.



This is very high level description of the transaction flow. The detail involves network and environment configuration on the SDK, security membership management to obtain signing key, handling transaction and event flows, and (depending on application) multiple consensus channels.

5. Client-Server API Reference

The following links point to message and service definitions for the grpc communications with the Fabric (peer, orderer and member service):

Proposal (proposal for deploy or transaction invocation)
ProposalResponse (universal proposal response)

Chaincode (chaincode spec, invocation spec, deployment spec)
ChaincodeProposal (chaincode-specific proposal header and payload)

Transaction (transaction invocation)
ChaincodeTransaction (chaincode-specific transaction action payload)

(New COP API is json based, needs to be added with new member services design)

The message definitions should be a source of inspirations for designing the SDK APIs. The APIs obviously don't have to faithfully reflect the message definitions, because the SDK can use smart defaults and state information to minimize the number of required parameters.

6. Specifications

Here we discuss the design principle and architecture consideration.

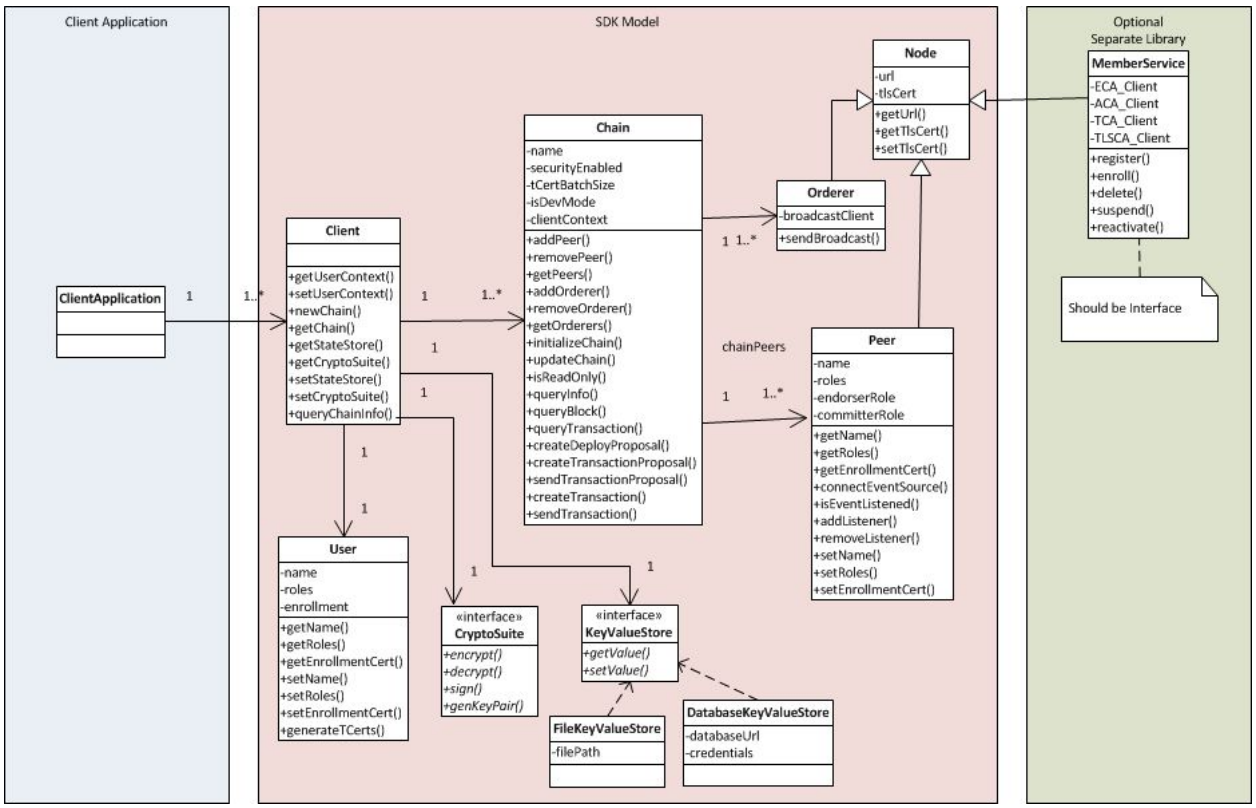
In total, we have several modules of different levels (smaller numbers means higher level):

Package: Hyperledger Fabric Client		
Module	Level	Functionality
Client	0	Main entrance module. It MUST allow users to construct any required objects to perform all supported operations against the network directly, e.g., chaincode deploy, transaction invoke, various queries. In addition, each language implementation can also decide, based on coding styles and prevalent community practices, whether to add convenient methods like sendTransaction(chain, tx);
Chain	1	A chain represents a network of peers formed ad hoc to start a channel of consensus where transactions can be independently processed. A network may have 1 or more chains. The peers within a chain maintain a separate ledger containing transactions delivered on the chain, including any configuration on membership. All transactions are sent on a

		chain, and an application may operate on multiple chains.
Peer	2	Represents a computing node on the network. Peer has roles of *endorser* and/or *committer*, which maintains the ledger. An application may connect to a number of peers it has access to.
Orderer	2	Similar to “Peer”, but represents the endpoint for the ordering service, which could be a single node (local setup during development) or a proxy node to a network of orderers. A Fabric-based blockchain network would have a single ordering service made up of a set of orderer nodes. An application can choose to trust a particular one of the orderers, or a set of orderers, or set up a proxy to propagate transaction broadcasts to the orderers.
User	2	Represents a user that will transact on the network. An instance of this class can be instantiated based on an enrollment certificate. The certificate may be obtained from the member service or from an external CA. In theory, this class could also represent a Peer node membership in the network. However, that is not a concern of an application (it’s more of a network administrative concern), so that is not exposed in this design.
Proposal	3	An enrolled user can issue an transaction proposal to a list of peers to endorse the transaction. Once the endorsement responses are received, the application can determine if enough endorsement signatures have been obtained and whether it should proceed to submit the transaction to the consensus service. This is a wrapper class around the raw GRPC message for proposal, providing convenience methods to construct it.
ProposalResponse	3	Response from the proposal call to the endorser, encapsulating endorsement results (yay or nay), signatures, etc. This is a wrapper class around the raw GRPC message for ProposalResponse, providing convenience methods to access its content (endorsement, signature, etc).
Transaction	3	An enrolled user can submit a transaction after having collected endorsements. A transaction request contains endorsement signatures and the MVCC + post-image, and targets the ordering service API. Transactions may be of two types: *Deploy* and *Invoke*.

		This is a wrapper class around the raw GRPC message for Transaction, providing convenience methods to construct it.
CryptoSuite	3	A crypto suite encapsulates algorithms for digital signatures and encryption with asymmetric key pairs, message encryption with a symmetric key, and secure hashing and MAC.
Package: Member Service		
Module	Level	Functionality
MemberService	0	This is the client for the optional component of the fabric, the member service. The main purpose of this class is to obtain user enrollment certificates from the member service. In addition, this class itself or an extension of it SHOULD also provide additional capabilities available in the Fabric's default member service implementation, namely user registration.

To help illustrate the relationship among the model classes above, the following UML diagram is created:



1. Client

Main interaction handler with end user. A client instance provides a handler to interact with a network of peers, orderers and optionally member services. An application using the SDK may need to interact with multiple networks, each through a separate instance of the Client.

Each client when initially created should be initialized with configuration data from the consensus service, which includes a list of trusted roots, orderer certificates and IP addresses, and a list of peer certificates and IP addresses that it can access. This must be done out of band as part of bootstrapping the application environment. It is also the responsibility of the application to maintain the configuration of a client as the SDK does not persist this object.

Each Client instance can maintain several chains representing channels and the associated sub-ledgers.

- new_chain

  Initializes a chain instance with the given name. This is really representing the “Channel” (as explained above), and this call returns an empty object. To initialize the channel, a list of participating endorsers and orderer peers must be configured first on the returned object.

  Params
  - name (str): The name of the chain, recommend using namespaces to avoid collision

Returns
- (Chain instance): The uninitialized chain instance.

- get_chain

  Get a chain instance from the state storage. This allows existing chain instances to be saved for retrieval later and to be shared among instances of the application. Note that it's the application/SDK's responsibility to record the chain information. If an application is not able to look up the chain information from storage, it may call another API that queries one or more Peers for that information.

  Params
  - name (str): The name of the chain

  Returns
  - (Chain instance or None): the chain instance for the name.

  Error:
  - The state store has not been set
  - A chain does not exist under that name

- query_chain_info

  This is a network call to the designated Peer(s) to discover the chain information. The target Peer(s) must be part of the chain in question to be able to return the requested information.

  Params
  - name (str): The name of the chain
  - peers (array of Peer instances): target Peers to query

  Returns
  - (Chain instance or None): the chain instance for the name.

  Error:
  - The target Peer(s) does not know anything about the chain

- set_state_store

  The SDK should have a built-in key value store implementation (suggest a file-based implementation to allow easy setup during development). But production systems would want a store backed by database for more robust storage and clustering, so that multiple app instances can share app state via the database (note that this doesn't necessarily make the app stateful). This API makes this pluggable so that different store implementations can be selected by the application.

  Params
  - store (KeyValueStore): instance of an alternative KeyValueStore implementation provided by the consuming app.

  Returns
  - None

- get_state_store

  A convenience method for obtaining the state store object in use for this client.

  Params
  - None

  Returns
  - (KeyValueStore instance): The KeyValueStore implementation object set within this Client, or null if it does not exist

- set_crypto_suite

  Sets an instance of the CryptoSuite interface implementation. A crypto suite encapsulates algorithms for digital signatures and encryption with asymmetric key pairs, message encryption with a symmetric key, and secure hashing and MAC.

  Params
  - Suite (object): an instance of a crypto suite implementation

- get_crypto_suite

  A convenience method for obtaining the CryptoSuite object in use for this client.

  Params
  - None

  Returns
  - (CryptoSuite instance): The CryptoSuite implementation object set within this Client, or null if it does not exist

- set_user_context

  Sets an instance of the User class as the security context of this client instance. This user's credentials (ECert) will be used to conduct transactions and queries with the blockchain network. Upon setting the user context, the SDK saves the object in a persistence cache if the "state store" has been set on the Client instance. If no state store has been set, this cache will not be established and the application is responsible for setting the user context again when the application crashed and is recovered.

  Params
  - user (User): an instance of the User class encapsulating the authenticated user's signing materials (private key and enrollment certificate)

- get_user_context

  As explained above, the client instance can have an optional state store. The SDK saves enrolled users in the storage which can be accessed by authorized users of the application (authentication is done by the application outside of the SDK). This function attempts to load the user by name from the local storage (via the KeyValueStore interface). The loaded user object must represent an enrolled user with a valid enrollment certificate signed by a trusted CA (such as the COP server).

  Params
  - name (str): The name of the user

  Returns
  - (User instance): The user object corresponding to the name, or null if the user does not exist or if the state store has not been set

2. Chain

The “Chain” object captures settings for a channel, which is created by the orderers to isolate transactions delivery to peers participating on channel. A chain must be initialized after it has been configured with the list of peers and orderers. The initialization sends a CONFIGURATION transaction to the orderers to create the specified channel and asks the peers to join that channel.

- add_peer

  Add peer endpoint to a chain object, this is a local-only operation

  Params
  - peer (Peer): an instance of the Peer class that has been initialized with URL, TLC certificate, and enrollment certificate

- remove_peer

  Remove peer endpoint from a chain object, this is a local-only operation

  Params
  - peer (Peer): an instance of the Peer class

- get_peers

  Get peers of a chain from local information.

  Params
  - None

  Returns
  - (Peer list): The peer list on the chain

- add_orderer

  Add orderer endpoint to a chain object, this is a local-only operation. A chain instance may choose to use a single orderer node, which will broadcast requests to the rest of the orderer network. Or if the application does not trust the orderer nodes, it can choose to use more than one by adding them to the chain instance. And all APIs concerning the orderer will broadcast to all orderers simultaneously.

  Params
  - orderer (Orderer): an instance of the Orderer class

- remove_orderer

  Remove orderer endpoint from a chain object, this is a local-only operation.

  Params
  - orderer (Orderer): an instance of the Orderer class

- get_orderers

  Get orderers of a chain. This is a local-only operation.

  Params
  - None

  Returns
  - (Orderer list): The orderer list on the chain

- initialize_chain

  Calls the orderer(s) to start building the new chain, which is a combination of opening new message stream and connecting the list of participating peers. This is a long-running

process. Only one of the application instances needs to call this method. Once the chain is successfully created, other application instances only need to call get_chain() to obtain the information about this chain.

Params
- None

Returns
- (bool): whether the chain initialization process was successful

- update_chain

  Calls the orderer(s) to update an existing chain. This allows the addition and deletion of Peer nodes to an existing chain, as well as the update of Peer certificate information upon certificate renewals.

  Params
  - None

  Returns
  - (bool): whether the chain update process was successful

- is_readonly

  Get chain status to see if the underlying channel has been terminated, making it a read-only chain, where information (transactions and states) can be queried but no new transactions can be submitted.

  Params
  - None

  Returns
  - (bool): is ready-only (true) or not

- query_info

  Queries for various useful information on the state of the Chain (height, known peers)

  Params
  - none

  Returns
  - (ChainInfo) with height, currently the only useful info

- query_block

  Queries the ledger for Block by block number

Params
- blockNumber (number)

Returns
- Object containing the block

- query_transaction

  Queries the ledger for Transaction by number

  Params
  - transactionID

  Returns
  - TransactionInfo containing the transaction

- create_deploy_proposal

  Create a proposal for transaction. This involves assembling the proposal with the data (chaincodeID, chaincode invocation spec, etc.) and signing it using the private key corresponding to the ECert to sign.

  Params
  - chaincode_path (string): path to the chaincode to deploy
  - chaincode_name (string): a custom name to identify the chaincode on the chain
  - fcn (string): name of the chaincode function to call after deploy to initiate the state
  - args (string[]): arguments for calling the init function designated by “fcn”
  - sign (Bool): Whether to sign the transaction, default to True

  Returns
  - (Proposal): The created Proposal instance or None.

- create_transaction_proposal

  Create a proposal for transaction. This involves assembling the proposal with the data (chaincode name, function to call, arguments, etc.) and signing it using the private key corresponding to the ECert to sign.

  Params
  - chaincode_name (string): The name given to the target chaincode to invoke
  - args (string[]): arguments for calling the “invoke” method on the chaincode
  - Sign (Bool): Whether to sign the transaction, default to True

  Returns
  - (Transaction_Proposal instance): The created Transaction_Proposal instance or None.

- send_transaction_proposal

  Send the created proposal to peer for endorsement.

  Params
  - transaction_proposal (Transaction_Proposal): The transaction proposal data
  - chain: The target chain whose peers the proposal will be sent to
  - retry (Number): Times to retry when failure, by default to 0 (no retry)

  Returns
  - (Transaction_Proposal_Response response): The response to send proposal request.

- create_transaction

  Create a transaction with proposal response, following the endorsement policy.

  Params
  - proposal_responses ([Transaction_Proposal_Response]): The array of proposal responses received in the proposal call.

  Returns
  - (Transaction instance): The created transaction object instance.

- send_transaction

  Send a transaction to the chain's orderer service (one or more orderer endpoints) for consensus and committing to the ledger.

  This call is asynchronous and the successful transaction commit is notified via a BLOCK or CHAINCODE event. This method must provide a mechanism for applications to attach event listeners to handle "transaction submitted", "transaction complete" and "error" events.

  Note that under the cover there are two different kinds of communications with the fabric backend that trigger different events to be emitted back to the application's handlers:
  - the grpc client with the orderer service uses a "regular" stateless HTTP connection in a request/response fashion with the "broadcast" call. The method implementation should emit "*transaction submitted*" when a successful acknowledgement is received in the response, or "*error*" when an error is received
  - The method implementation should also maintain a persistent connection with the Chain's event source Peer as part of the internal event hub mechanism in order to support the fabric events "BLOCK", "CHAINCODE" and "TRANSACTION". These events should cause the method to emit "*complete*" or "*error*" events to the application.

  Params
  - transaction (Transaction): The transaction object constructed above

Returns
- result (EventEmitter): an handle to allow the application to attach event handlers on “submitted”, “complete”, and “error”.

3. User

The User class represents users that have been enrolled and represented by an enrollment certificate (ECert) and a signing key. The ECert must have been signed by one of the CAs the blockchain network has been configured to trust. An enrolled user (having a signing key and ECert) can conduct chaincode deployments, transactions and queries with the Chain.

User ECerts can be obtained from a CA beforehand as part of deploying the application, or it can be obtained from the optional Fabric COP service via its enrollment process.

Sometimes User identities are confused with Peer identities. User identities represent signing capability because it has access to the private key, while Peer identities in the context of the application/SDK only has the certificate for verifying signatures. An application cannot use the Peer identity to sign things because the application doesn’t have access to the Peer identity’s private key.

- get_name

  Get member name. Required property for the instance objects.

  Returns (str):
  - The name of the user

- get_roles

  Get the user’s roles. It’s an array of possible values in “client”, and “auditor”. The member service defines two more roles reserved for peer membership: “peer” and “validator”, which are not exposed to the applications.

  Returns (str[]):
  - The roles for this user

- get_enrollment_certificate

  Returns the underlying ECert representing this user's identity.

  Params: none
  Returns:
  - Certificate in PEM format signed by the trusted CA

- set_name

  Set the user's name/id.

  Params:
  - name (string[]): The user name / id.

- set_roles

  Set the user's roles. See above for legitimate values.

  Params:
  - Roles (string[]): The list of roles for the user

- set_enrollment_certificate

  Set the user's Enrollment Certificate.

  Params:
  - Certificate : The certificate in PEM format signed by the trusted CA

- generate_tcerts

  Gets a batch of TCerts to use for transaction. there is a 1-to-1 relationship between TCert and Transaction. The TCert can be generated locally by the SDK using the user's crypto materials.

  Params
  - count (number): how many in the batch to obtain?
  - Attributes (string[]): list of attributes to include in the TCert

  Returns (TCert[]):

- An array of TCerts

4. Peer

The Peer class represents the remote Peer node and its network membership materials, aka the ECert used to verify signatures. Peer membership represents organizations, unlike User membership which represents individuals.

When constructed, a Peer instance can be designated as an event source, in which case a “eventSourceUrl” attribute should be configured. This allows the SDK to automatically attach transaction event listeners to the event stream.

It should be noted that Peer event streams function at the Peer level and not at the chain and chaincode levels.

- connectEventSource

Since practically all Peers are event producers, when constructing a Peer instance, an application can designate it as the event source for the application. Typically only one of the Peers on a Chain needs to be the event source, because all Peers on the Chain produce the same events. This method tells the SDK which Peer(s) to use as the event source for the client application. It is the responsibility of the SDK to manage the connection lifecycle to the Peer’s EventHub. It is the responsibility of the Client Application to understand and inform the selected Peer as to which event types it wants to receive and the call back functions to use.

Params:
- None

Result:
- Promise/Future: this gives the app a handle to attach “success” and “error” listeners

- is_event_listened

A network call that discovers if at least one listener has been connected to the target Peer for a given event. This helps application instance to decide whether it needs to connect to the event source in a crash recovery or multiple instance deployment.

NOTE: this requires enhancement on the Peer event producer.

Params:
- eventName (string): required
- chain (Chain): optional

Result:
- (boolean): whether the said event has been listened on by some application instance on that chain

- addListener

  For a Peer that is connected to eventSource, the addListener registers an EventCallBack for a set of event types. addListener can be invoked multiple times to support differing EventCallBack functions receiving different types of events.

  Note that the parameters below are optional in certain languages, like Java, that constructs an instance of a listener interface, and pass in that instance as the parameter.

  Params:
  - eventType : ie. Block, Chaincode, Transaction
  - eventTypeData : Object Specific for event type as necessary, currently needed for “Chaincode” event type, specifying a matching pattern to the event name set in the chaincode(s) being executed on the target Peer, and for “Transaction” event type, specifying the transaction ID
  - eventCallback : Client Application class registering for the callback.

  Returns:
  - [event-listener-ref] a reference to the event listener, some language uses an ID (javascript), others uses object reference (Java)

- removeListener

  Unregisters a listener.

  Params:
  - [event-listener-ref] : reference returned by SDK for event listener

  Returns:
  - statusFlag: Success / Failure

- get_name

  Get the Peer name. Required property for the instance objects.

  Returns (str):
  - The name of the Peer

- set_name

  Set the Peer name / id.

  Params:
  - Name (string): The unique name / id of this Peer.

- get_roles

  Get the user's roles the Peer participates in. It's an array of possible values in "client", and "auditor". The member service defines two more roles reserved for peer membership: "peer" and "validator", which are not exposed to the applications.

  Returns (str[]):
  - The roles for this user

- set_roles

  Set the user's roles the Peer participates in. See above for legitimate values.

  Params:
  - Roles (string[]): The list of roles for the user

- get_enrollment_certificate

  Returns the underlying ECert representing this user's identity.

  Params: none
  Returns:
  - Certificate in PEM format signed by the trusted CA

- set_enrollment_certificate

  Set the Peer's enrollment certificate.

  Params:
  - Certificate: Certificate in PEM format signed by the trusted CA

5. KeyValueStore (interface)

A blockchain application needs to save state, including user enrollment materials (private keys, CA-signed certificates). These state need to be persisted. The "KeyValueStore" interface provide a simple mechanism for the SDK to automatically save state on behalf of the application.

If the application uses a software-based key generation implementation of CryptoSuite, it requires a key value store. If one has not been set by the application, the SDK should default to a built-in implementation, such as one based on local file system.

The SDK can also save user enrollment materials in the key value store as an optional cache. But if the application does not configure one, the SDK should interpret that as the application has chosen to always set the user context for a session, and does not attempt to use a default key value store.

- get_value

  Retrieves a value given a key
  Params
  - key (str): The name of the key

  Returns
  - Result (Object): The value

- set_value

  Sets the value
  Params
  - Key
  - value

  Returns
  - Acknowledgement of successful storage of the value

6. CryptoSuite(Interface)

A crypto suite encapsulates algorithms for digital signatures and encryption with asymmetric key pairs, message encryption with a symmetric key, and secure hashing and MAC. This is a mirror-design of the BCCSP (Blockchain Crypto Service Provider) interface published by the fabric crypto team.

Default implementation currently planned for the peer and COP, and must be implemented by the SDK's default implementation also:

- ECDSA: curves "secp256r1" and "secp384r1"
- AES: AES128, AES256 with CBC/CTR/GCM mode key length = 128 bits
- SHA: SHA256, SHA384, SHA3_256, SHA3_384

- generate_key

  Generate a key based on the options. The output can be a private key or a public key in an asymmetric algorithm, or a secret key of a symmetric algorithm.

  Params
  - opts (Object): an object that encapsulates two properties, "algorithm" and "ephemeral".

  Returns
  - Result (Key): The key object

- deriveKey

  Derives a key from k using opts.

  Params
  - k (Key)
  - opts (Object)

  Returns
  - (Key) derived key

- importKey

  Imports a key from its raw representation using opts.

  Params
  - k (Key)
  - opts (Object)

  Returns
  - (Key) An instance of the Key class wrapping the raw key bytes

- getKey

  Returns the key this CSP associates to the Subject Key Identifier *ski*.

  Params
  - ski (byte[])

  Returns
  - (Key) An instance of the Key class corresponding to the *ski*

- hash

  Hashes messages *msg* using options *opts*.

  Params
  - msg (byte[])
  - opts (Object) an object that encapsulates property “algorithm” with values for hashing algorithms such as “SHA2” or “SHA3”

  Returns
  - (Key) An instance of the Key class corresponding to the *ski*

- encrypt

  Encrypt plain text.

  Params
  - key (Key) public encryption key
  - plainText (byte[])
  - opts (Object)

  Returns
  - (byte[]) Cipher text

- decrypt

  Decrypt cipher text.

  Params
  - key (Key) private decryption key
  - cipherText (byte[])
  - opts (Object)

  Returns
  - (byte[]) Plain text

- sign

  Sign the data.
  Params
  - Key (Key) private signing key
  - digest (byte[]) fixed-length digest of the target message to be signed
  - opts (function) hashing function to use

  Returns
  - Result(Object):Signature object

- verify

  Verify the signature.

  Params
  - key (Key) public verification key
  - signature (byte[]) signature
  - digest (byte[]) original digest that was signed

  Returns
  - (bool): verification successful or not

7. Handling Network Errors

The client SDK communicates with the fabric in two different fashions: stateless HTTP connections and persistent HTTP connections.

“send_proposal” and “send_transaction” calls are stateless calls in a request/response fashion. When a network error occurs, the call will timeout. The SDK should have a configurable timeout value for the application to control the behavior of these calls based on the understanding of the network characteristics between the client application and the fabric nodes.

In addition, these methods may have a “retry” value to allow the SDK to automatically attempt to re-try the HTTP call upon timeout error. After the retries are all attempted and still timeout occurs, the method should then return with an error.

On the other hand, event stream connections between the SDK and the event source Peers are persistent connections. Specifically, in the case of the event streaming interface defined by the Fabric, the connection is bi-directional to allow messages to be sent in both directions. In the

case of network failures, this connection gets broken, resulting in lost events and failure to trigger listeners registered by the client application.

The SDK should make attempts to re-establish the connection on behalf of the application. But if after “re-try” number of attempts the connection can not be restored, it should notify the application of this condition with a reasonably high severity error.

8. Reference

1) Next-Consensus-Architecture_Proposal: https://github.com/hyperledger/fabric/blob/master/proposals/r1/Next-Consensus-Architecture-Proposal.md

2) Consensus endorsing, consenting, and committing model: https://jira.hyperledger.org/browse/FAB-37

3) Node.js SDK https://github.com/hyperledger/fabric-sdk-node

4) Fabric-Cop Design https://docs.google.com/document/d/1TRYHcaT8yMn8MZlDtreqzkDcXx0WI50AV2JpAcvAM5w
5) Next Hyperledger-Fabric Architecture Protocol messages https://docs.google.com/document/d/1qD2zOTxmPoss3Xop_pDdkSXHozIR4N27NEsymosW4Gk