Hyperledger Fabric SDK Design Specification v1.0

08.29.2016

Baohua Yang, Jim Zhang, Pardha Vishnumolakala, Muhammad Altaf

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	Muhammad Altaf

Overview

SDK is the primary way for an application to interact with a Hyperledger Fabric based network. Besides the SDK, there are other means to interact with the network, each for a slightly different purpose. The "peer" command is for administrative purposes locally on the peer nodes (start, stop, etc.). The REST API is for peer nodes to publish metadata such as operational status, security settings, etc.

Even though today the REST API that runs directly on the peer nodes allows many operations to be run, such as deploy chaincode, and submit transactions, going forward this will not be the case any longer for a couple of reasons:

- grpc is the primary communication channel between the client and the peer network
- From operation's point of view, peer nodes are responsible for computations involved in transaction processing, and a REST API layer should be run as a front-end layer instead of directly on the peer nodes

Therefore, SDK designs are tightly coupled to the Fabric's grpc communication spec between the client and the peer. They should not be designed after the REST API.

This document explains what a Fabric software development kit (SDK) should accomplish by defining the data models and APIs available to chaincode and application developers.

This spec is targeting fabric v1.0.

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

Goals

1. **Chaincode development**

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

2. **Application development**

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

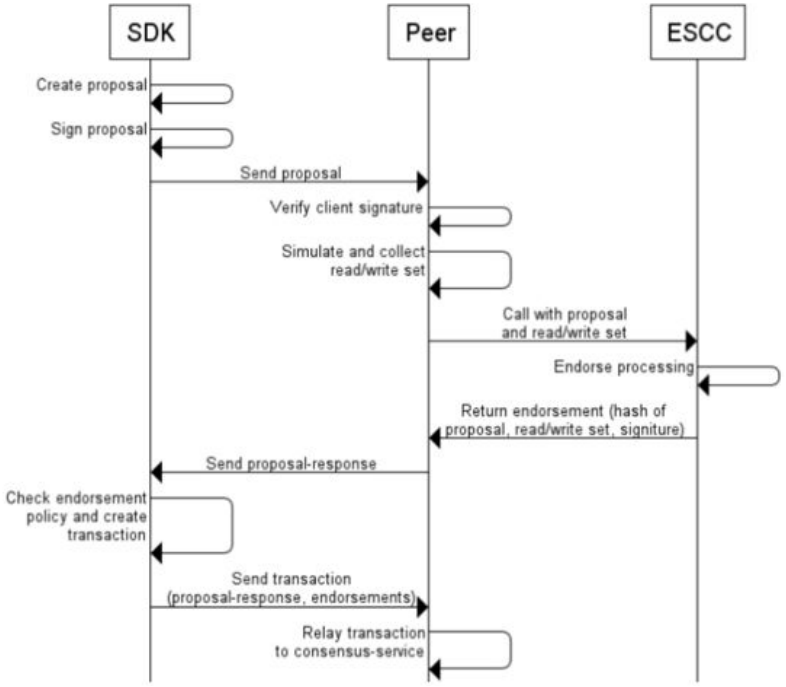
3. **Well documented api(s), data models, and sample code**

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

4. **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.

Scenarios



Membership Enrollment

The SDK will enroll a member to the chain, and should remember the authentication at local store.

Transaction Support

The SDK will support:

1) Create and sign a proposal, then endorse the proposal to peer (endorser). A response will be received from peer;
2) If the response is successful, SDK will check the endorsement policy. If meet the policy, then create a transaction and send it to peer.

What is the difference between Chaincode and Application code?

Both chaincode and application code contain business logic. The main difference is where the code is deployed.

Chaincode is deployed and executed on the network and can read/write world state using api provided by SDK.

Application code is not deployed on the network. Applications will use the SDK to invoke the chaincode deployed on the network, listen to various events generated by the network, query transactions and blocks present in the ledger.

Server-Side API Reference

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

fabric.proto
chaincode.proto

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.

Specifications

Here we discuss the design principle and architecture consideration.

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

Module	Level	Functionality

Client	0	Main entrance module. Users should mostly use this as the interaction handler. E.g., Client.transaction_deploy(chain, tx).
Chain	1	To support Client, presents a network of peers. Captures settings and metadata about the network, such as what peers have been added, member service to use, etc.
MemberService	1	To support Client, provide membership related operations.
Peer	2	To support Chain, represent a single peer node. Peer has roles of *submitter* or *endorser*. An end-client may connect to any peer.
Member	2	To support Chain, represents a membership (Ecert + Tcert) to interact with the Chain. Can be a special “registrar” type that can register other members.
EndorsementProposal	3	An authorized member (aka “user”) can issue an Endorsement proposal to a list of peers. Only proposals that get enough endorsement signatures should proceed to the ordering and validation phase.
Transaction	3	An authorized member (aka “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*.

1. Client

Main interaction handler with end user.
Client can maintain several chains.

- new_chain

  Init a chain instance with given name.

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

  Returns (Chain instance): The init’ed chain instance.

- get_chain

  Get a chain instance

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

Returns (Chain instance or None): Get the chain instance with the name.

- Key_related_process(TBD)

  TODO.
  store or change persistent and private date
  Params
  - name (str): The name of the key

  Returns (): The result

- member_register

  Register a user. Add a new user entry to the user registry in member service.
  Params
  - name (str): The name of the user
  - password (str): The password of the user. Nil value can be passed in which case a password is generated and returned.
  - roles (str[]): any combination of 'client', 'peer', 'validator', 'auditor'. Defaults to “client”.
  - affiliation (str): name of org this user belongs to. Defaults to the same affiliation as the registrar.

  Returns
  - Registered member (Member): The registration result, or None for failure.
  - enrollmentSecret: if “password” was not nil in the call parameter, it’s echo’ed here. Otherwise a high-strength password is generated and returned.

- member_enroll

  Enroll a member. Exchange the one-time user enrollment secret for a user certificate.
  Params
  - name (str): The name of the chain
  - password (str): The password of the user

  Returns
  - certs (Cert data): The created data

- transaction_deploy

  Deploy a chaincode to peer.
  Params
  - peer (Peer): The instance of the peer
  - transaction (Transaction): The instance of the transaction

  Returns
  - result(TransactionResponse): The return result response

- transaction_invoke

  Invoke a chaincode to peer.
  Params
  - peer (Peer): The instance of the peer
  - transaction (Transaction): The instance of the transaction

  Returns
  - result(TransactionResponse): The return result response

- transaction_query

  Query a chaincode from peer.
  Params
  - peer (Peer): The instance of the peer
  - transaction (Transaction): The instance of the transaction

  TODO: maybe use the transaction id is enough
  Returns
  - result(TransactionResponse): The return result response

2. Chain

The “Chain” object captures settings for a network of peers and member service(s), including: a list of peers that the client app has access to (there will be peers that the client app doesn’t have access to); Url(s) to member service(s); algorithms for encryption, digital signature and hashing, etc.

- add_peer

  Add peer endpoint to a chain
  Params
  - name (str): The name of the chain
  - type (str): The type of the peer
  - address (str): The address of the chain

  Returns
  - result(Bool): The response

- get_peers

  Get peers of a chain.
  Params
  - None

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

- get_status

  Get chain statue.
  Params
  - None

  Returns
  - (Chain status): The status of the chain instance.

- create_transaction_proposal

  Create a proposal for transaction.
  Params
  - transaction_proposal (Transaction_Proposal): The transaction proposal data
  - Sign (book): Whether to sign the transaction, by default to True

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

- send_transaction_proposal

  Send the created proposal to peer.

Params
- transaction_proposal (Transaction_Proposal): The transaction proposal data
- retry (Number): how many times to retry when failure, by default to 0

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_response (Transaction_Proposal_Response): The proposal response.

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

- send_transaction

  Send a transaction.
  Params
  - transaction (Transaction): The transaction

  Returns
  - result (Transaction_Response): The returned response from peer.

- get_membersrvc

  Get a MemberService.
  Params
  - name (str): The name of the member service.

  Returns
  - memberservice (MemberService instance): The member service instance.

3. MemberService

- get_name

  TODO:do we need this?
  Get member name.
  Params
  - name (str): The name of the chain

  Returns (str): The name of the MemberService instance.

- register_user

  Implement register user.
  Params
  - name (str): The name of the user
  - password (str): The password of the user

  Returns (True or False): The result

- is_registered

  Determine if this user has been registered.
  Params
  - name (str): The name of the user

  Returns (True or False): The result

- enroll

  Implement enroll
  Params
  - cert (Cert): The cert of the user

  Returns (True or False): The result

- is_enrolled

  Determine if this name has been enrolled
  Params
  - name (str): The name of the member

  Returns (True or False): The result

- delete_enrollment

  delete member
  Params
  - name (str): The name of the member

  Returns (True or False): The result

- get_enrollment

  Get member info
  Params
  - name (str): The name of the enrollment

  Returns (Chain instance): The inited chain instance.

- get_enrollment_ecert

  Get enrollment certificates for enroll
  Params
  - name (str): The name of the chain

  Returns (Cert): The cert

- get_enrollment_tcert

  Use for transaction, there is a 1-to-1 relationship between TCert and Transaction
  Params
  - name (str): The name of the chain

  Returns (Tcert): The Tcert

- get_transaction

  Get a transaction from member.
  Params
  - name (str): The name of the member

  Returns (Transactions): The transactions.

- process_confidentiality

  Process some security and identity info.
  Params
  - name (str): The name of the chain

  Returns (Chain instance): The inited chain instance.

- Set_crypto_functions

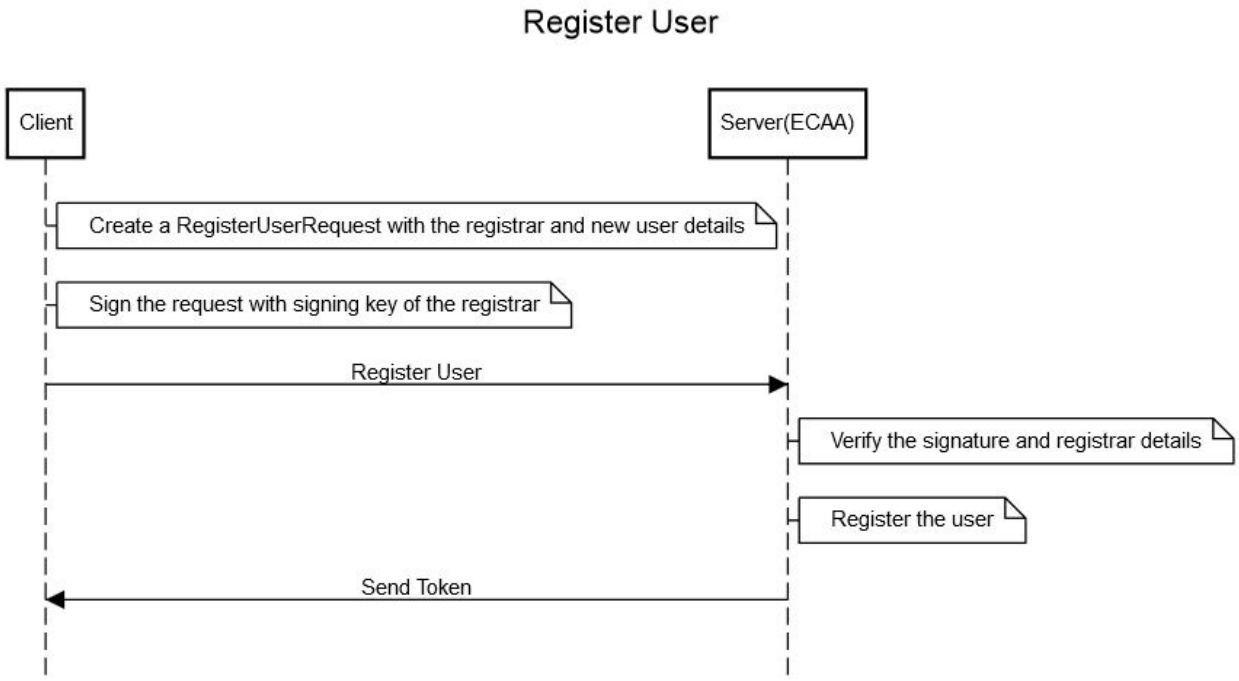
  Use the crypto implementation provided in the API instead of using the default implementation. This function can be used where user plugs in their own implementation of crypto on the server side, so they would need to plug in a compatible key generation and encryption/decryption functions in the SDK as well. This is important for countries like Russia and China where some customized crypto algorithms must be used.
  Params
  - functions (CryptoFunctions): An implementation of the CryptoFunctions interface implementing the key pair generation and decryption functions

Design

This section provides a detailed design of various functions of the SDK for better understanding of the underlying infrastructure and APIs.

Member services design

- **Register User**
  This API allows registering a new user with the membership services. This operation can only be performed by an already enrolled user with registrar role. To enroll a registrar for the first time, see the next section “Enroll User”.
  Registering a user involves following steps:
  (1) Create a RegisterUserRequest with the registrar and new user details
  (2) Sign the request with signing key of the registrar (used while enrolling the user)
  (3) Update the request with the signature
  (4) Send request to ECAA on the server side
  (5) Server registers the user and sends back the token that should be used while enrolling the user



- **Enroll User**
  User enrollment is a two step process.
  (1) Send a create Certificate request to the server with two public keys:
    - A signature key that will be used to verify the signature of the keys
    - An encryption key that will be used to encrypt the data

In response to this request, server creates a challenge and sends it to the client after encrypting it with the encryption public key to ensure that the client is in possession of the encryption private key

Creating Certificate pairs:
The certificate pairs supported by the ECAP endpoint are 256-bit ECDSA certificates with curve secp256r1. They can be easily generated using language specific APIs and no custom manipulation is required. For example, if using java, following could generate an ECDSA key pair using bouncy castle library.

```
ECGenParameterSpec ecGenSpec = new ECGenParameterSpec("secp256r1");
KeyPairGenerator g = KeyPairGenerator.getInstance("ECDSA", "BC");
g.initialize(ecGenSpec, new SecureRandom());
KeyPair pair = g.generateKeyPair();
```

(2) Client decrypts the token returned by the server in step 1, adds it to the request, creates the signature with the signing private key and sends another Create Certificate request to the server. Server verifies that signature and decrypted token is correct, server sends enrollment certificates to the client.

Decrypting the token:
Client needs to decrypt the token with his encryption private key (generated in step 1) in order to proceed with the final step of the enrollment process. The token received from the server is constructed of the following components:

65 byte ephemeral public key	28 byte encrypted message	32 bytes MAC

a) Ephemeral public key: This key can be used to derive the shared secret that can be used to generate the decryption key and to calculate mac of the encrypted token.
b) Encrypted message: This field contains a 16 byte IV followed by the actual token encrypted with AES using an encryption key. The encryption key can be generated using the shared secret derived in the step a above.
c) MAC: This field contains the MAC of the encrypted message. To ensure that the message contents have been extracted correctly, MAC of the encrypted message must be the same as this 32 byte MAC.

The following steps must be performed on the received challenge in order to decrypt the token correctly.

Deriving Shared Secret:
To derive a shared secret, a key agreement must be established between the ephemeral public key and the private key using ECDH key agreement algorithm. This key agreement then can be used to generate the shared secret. Following java code snippet can be used as an example to derive the shared secret:

```
KeyAgreement keyAgreement = KeyAgreement.getInstance("ECDH",
"BC");
keyAgreement.init(keyPair.getPrivate());
keyAgreement.doPhase(ephemeralPublicKey, true);
byte[] sharedSecret = keyAgreement.generateSecret();
```

Generating MAC and encryption keys:
HKDF with SHA3-256 can be used to generate the mac and encryption keys. MAC key generated here can be used to calculate the mac of the encrypted message, and the encryption key can be used to decrypt the token contained in the encrypted message. Following code snippet can be used as an example to generate MAC key and encryption key.

```
HKDFBytesGenerator hkdfBytesGenerator = new
HKDFBytesGenerator(new SHA3Digest());

hkdfBytesGenerator.init(new HKDFParameters(sharedSecret,
null, null));

byte[] encryptionKey = new byte[32];
hkdfBytesGenerator.generateBytes(encryptionKey, 0, 32);

byte[] macKey = new byte[32];
hkdfBytesGenerator.generateBytes(macKey, 0, 32);
```

Calculating MAC:
Once the MAC key has been generated using the shared secret, we can now calculate the mac of the 28 byte encrypted message and check if it matches with the 32 byte MAC provided at the end of the token. If both MACs don't match, we can safely assume that either the message has been corrupted or we did not use the correct algorithm to derive the MAC. The hash digest to calculate MAC is SHA3-256. Following can be used as an example to calculate MAC.

```
HMac hmac = new HMac(new SHA3Digest());
hmac.init(new KeyParameter(macKey));
hmac.update(encryptedMessage, 0, encryptedMessage.length);
byte[] mac = new byte[32];
hmac.doFinal(mac, 0);
```

Decrypting the token:
This is the final step to decrypt the token embedded in the encrypted message. The encrypted message contains 16 byte IV as first 16 bytes of the message, and the last 12 bytes are the actual encrypted token. The encrypted token can be decrypted using the AES algorithm using the IV and encryption key generated above. Following code snippet can be used as an example to decrypt the token:

```
Cipher cipher = Cipher.getInstance("AES/CFB/NoPadding");
cipher.init(Cipher.DECRYPT_MODE, new
SecretKeySpec(encryptionKey, "AES"), new
IvParameterSpec(iv));
Byte[] token = cipher.doFinal(encryptedToken);
```

Generating signature:
To generate the signature, we'll need the private key to sign the message and a binary representation of the request itself. To sign the request, we'll first need to create the hash of the request using SHA3-256 digest. The following sample code snippet creates the SHA3-256 hash of the request.

```
Digest digest = new SHA3Digest();
byte[] hash = new byte[digest.getDigestSize()];
digest.update(requestBytes, 0, requestBytes.length);
digest.doFinal(hash , 0);
```

Once the sha has been calculated, we need to create a deterministic K calculator using SHA-512 digest. This can be created using java bouncy castle library as follows:

```
DSAKCalculator kCalc = new HMacDSAKCalculator(new SHA512Digest())
```

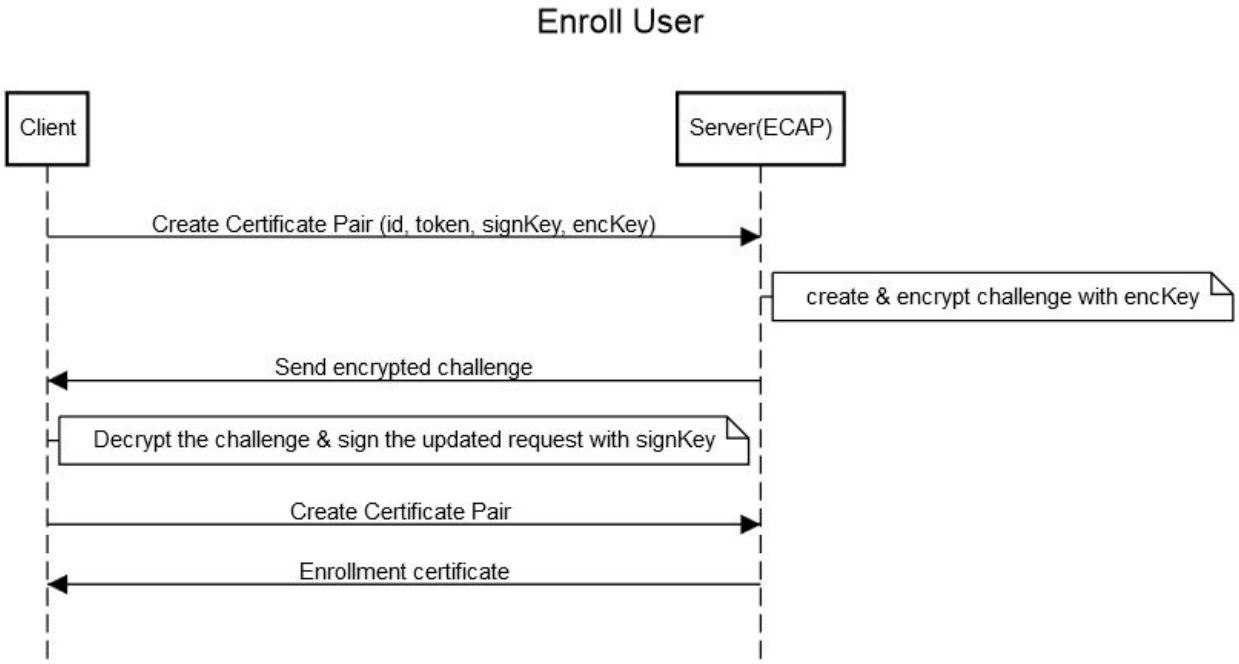
We can use this kCalc to sign the hash using a standard implementation of ECDSA signature algorithm. In java, following code snippet can be used as an example to understand the process.

```
X9ECParameters params = SECNamedCurves.getByName("secp256r1");
ECDomainParameters ecParams = new ECDomainParameters(
params.getCurve(), params.getG(), params.getN(), params.getH());

ECDSASigner signer = new ECDSASigner(kCalc);
ECPrivateKeyParameters privKeyParams = new ECPrivateKeyParameters(
((ECPrivateKey) privateKey).getS(), ecParams);
signer.init(true, privKeyParams);
BigInteger[] sigs = signer.generateSignature(hash);
```

The sig is an array of two BigInteger elements representing R and S fields of the signature respectively.

The process of enrolling the user can be summarized by the following diagram:



There are few important details about this process that must be kept in mind.

- The keys must be generated using ECDSA algorithm (Server accepts Elliptic Curve sec256r1 only) and the challenge must be decrypted using ECIES algorithm.
- A user can only be enrolled once. If a user has been already enrolled, the enrollment must be removed first before attempting another enrollment. Therefore it is important to store the enrollment certificates on client side.

- **Removing user enrollment**

Reference

Node.js SDK

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

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