**Hyperledger Intern**

**How Sawtooth convert**

**to Fabric**

**謝孟剛(Roger Hsieh)**

Table of Contents

[一、Conversion from Smart Contract 3](#_Toc531338490)

[二、Conversion from Client side 21](#_Toc531338491)

[三、Reference 29](#_Toc531338492)

## 一、Conversion from Smart Contract

There are similarities of smart contract between Fabric and Sawtooth. They both use one function to call other functions of smart contract. The function in Fabric called invoke, and the function in Sawtooth called apply. We can convert Sawtooth Transaction Families to Fabric Chaincode through the similarity, or develop conversion tool. But there also are differences between Fabric and Sawtooth. In Fabric, there are composite key, rich query and collection… and so on, but in Sawtooth are not.

The program example tunachain of Sawtooth Processor, written by python:

# The code of tunachain\_state.py. One of the Sawtooth Processor code of the tuna chain example. Define TunachainState class and methods about getting and setting state.

tunachain\_state.py:

import hashlib

import json

import logging

LOGGER = logging.getLogger(\_\_name\_\_)

TUNACHAIN\_NAMESPACE = hashlib.sha512(

'transfer-chain'.encode('utf-8')).hexdigest()[0:6]

def \_get\_address(key):

return hashlib.sha512(key.encode('utf-8')).hexdigest()[:62]

def \_get\_asset\_address(asset\_name):

return TUNACHAIN\_NAMESPACE + '00' + \_get\_address(asset\_name)

def \_get\_transfer\_address(asset\_name):

return TUNACHAIN\_NAMESPACE + '01' + \_get\_address(asset\_name)

def \_deserialize(data):

return json.loads(data.decode('utf-8'))

def \_serialize(data):

return json.dumps(data, sort\_keys=True).encode('utf-8')

class TunachainState(object):

TIMEOUT = 3

def \_\_init\_\_(self, context):

self.\_context = context

def get\_asset(self, name):

return self.\_get\_state(\_get\_asset\_address(name))

def get\_transfer(self, name):

return self.\_get\_state(\_get\_transfer\_address(name))

def set\_asset(self, name, owner):

address = \_get\_asset\_address(name)

state\_data = \_serialize(

{

"name": name,

"owner": owner

})

return self.\_context.set\_state(

{address: state\_data}, timeout=self.TIMEOUT)

def set\_transfer(self, name, owner):

address = \_get\_transfer\_address(name)

state\_data = \_serialize(

{

"asset": name,

"owner": owner

})

return self.\_context.set\_state(

{address: state\_data}, timeout=self.TIMEOUT)

def delete\_transfer(self, name):

return self.\_context.delete\_state(

[\_get\_transfer\_address(name)],

timeout=self.TIMEOUT)

def \_get\_state(self, address):

state\_entries = self.\_context.get\_state(

[address], timeout=self.TIMEOUT)

if state\_entries:

entry = \_deserialize(data=state\_entries[0].data)

else:

entry = None

return entry

# The code of tunachain\_payload.py. One of the Sawtooth Processor code of the tuna chain example. Define TunachainPayload class and methods about getting and the data property.

tunachain\_payload.py:

import json

from sawtooth\_sdk.processor.exceptions import InvalidTransaction

class TunachainPayload(object):

#constructor

def \_\_init\_\_(self, payload):

try:

data = json.loads(payload.decode('utf-8'))

except ValueError:

raise InvalidTransaction("Invalid payload serialization")

action = data.get('action')

asset = data.get('asset')

owner = data.get('owner')

if not action:

raise InvalidTransaction('Action is required')

if action not in ('create', 'transfer', 'accept', 'reject'):

raise InvalidTransaction('Invalid action: {}'.format(action))

if not asset:

raise InvalidTransaction('Asset is required')

if action == 'transfer':

if not owner:

raise InvalidTransaction(

'Owner is required for "transfer" transaction')

self.\_action = action

self.\_asset = asset

self.\_owner = owner

@property

def action(self):

return self.\_action

@property

def asset(self):

return self.\_asset

@property

def owner(self):

return self.\_owner

# The code of handler.py. One of the Sawtooth Processor code of the tuna chain example. Define apply function to call other functions of processor.

handler.py:

import logging

from sawtooth\_sdk.processor.handler import TransactionHandler

from sawtooth\_sdk.processor.exceptions import InvalidTransaction

from tunachain\_processor.tunachain\_payload import TunachainPayload

from tunachain\_processor.tunachain\_state import TunachainState

from tunachain\_processor.tunachain\_state import TUNACHAIN\_NAMESPACE

LOGGER = logging.getLogger(\_\_name\_\_)

class TunachainTransactionHandler(TransactionHandler):

@property

def family\_name(self):

return 'transfer-chain'

@property

def family\_versions(self):

return ['0.0']

@property

def encodings(self):

return ['application/json']

@property

def namespaces(self):

return [TUNACHAIN\_NAMESPACE]

**def apply(self, transaction, context):**

header = transaction.header

signer = header.signer\_public\_key

payload = TunachainPayload(transaction.payload)

state = TunachainState(context)

LOGGER.info('Handling transaction: %s > %s %s:: %s',

payload.action,

payload.asset,

'> ' + payload.owner[:8] + '... ' if payload.owner else '',

signer[:8] + '... ')

if payload.action == 'create':

\_create\_asset(asset=payload.asset,

owner=signer,

state=state)

elif payload.action == 'transfer':

\_transfer\_asset(asset=payload.asset,

owner=payload.owner,

signer=signer,

state=state)

elif payload.action == 'accept':

\_accept\_transfer(asset=payload.asset,

signer=signer,

state=state)

elif payload.action == 'reject':

\_reject\_transfer(asset=payload.asset,

signer=signer,

state=state)

else:

raise InvalidTransaction('Unhandled action: {}'.format(

payload.action))

def \_create\_asset(asset, owner, state):

if state.get\_asset(asset) is not None:

raise InvalidTransaction(

'Invalid action: Asset already exists: {}'.format(asset))

state.set\_asset(asset, owner)

def \_transfer\_asset(asset, owner, signer, state):

asset\_data = state.get\_asset(asset)

if asset\_data is None:

raise InvalidTransaction('Asset does not exist')

if signer != asset\_data.get('owner'):

raise InvalidTransaction('Only an Asset\'s owner may transfer it')

state.set\_transfer(asset, owner)

def \_accept\_transfer(asset, signer, state):

transfer\_data = state.get\_transfer(asset)

if transfer\_data is None:

raise InvalidTransaction('Asset is not being transfered')

if signer != transfer\_data.get('owner'):

raise InvalidTransaction(

'Transfers can only be accepted by the new owner')

state.set\_asset(asset, transfer\_data.get('owner'))

state.delete\_transfer(asset)

def \_reject\_transfer(asset, signer, state):

transfer\_data = state.get\_transfer(asset)

if transfer\_data is None:

raise InvalidTransaction('Asset is not being transfered')

if signer != transfer\_data.get('owner'):

raise InvalidTransaction(

'Transfers can only be rejected by the potential new owner')

state.delete\_transfer(asset)

# The code of main.py. One of the Sawtooth Processor code of the tuna chain example. Define main function to let the processor start.

main.py:

import argparse

import sys

from sawtooth\_sdk.processor.core import TransactionProcessor

from sawtooth\_sdk.processor.log import init\_console\_logging

from tunachain\_processor.handler import TunachainTransactionHandler

def parse\_args(args):

parser = argparse.ArgumentParser(

formatter\_class=argparse.RawTextHelpFormatter)

parser.add\_argument(

'-C', '--connect',

default='tcp://localhost:4004',

help='Endpoint for the validator connection')

parser.add\_argument(

'-v', '--verbose',

action='count',

default=0,

help='Increase output sent to stderr')

return parser.parse\_args(args)

def main(args=None):

if args is None:

args = sys.argv[1:]

opts = parse\_args(args)

processor = None

try:

processor = TransactionProcessor(url=opts.connect)

init\_console\_logging(verbose\_level=opts.verbose)

handler = TunachainTransactionHandler()

processor.add\_handler(handler)

processor.start()

except KeyboardInterrupt:

pass

except Exception as err: # pylint: disable=broad-except

print("Error: {}".format(err))

finally:

if processor is not None:

processor.stop()

We develop the program example material supply chain of Fabric Chaincode with javascript:

// The chaincode of material supply chain which named unity.js.

unity.js:

'use strict';

const shim = require('fabric-shim')

const util = require('util')

//Declare Chaincode class and methods of the Chaincode. Methods including Init, Invoke and other methods about getting and setting blockchain data.

let Chaincode = class {

//Init method: Use Init method to instantiate chaincode.

**async Init(stub){**

console.info('====== Instantiated unity chaincode =======');

let ret = stub.getFunctionAndParameters();

console.info(ret);

let args = ret.params;

// initialise only if 4 parameters passed.

if (args.length != 4) {

return shim.error('Incorrect number of arguments. Expecting 4');

}

//\"args\":[\"a\",\"100\",\"b\",\"200\"

let A = args[0];

let B = args[2];

let Aval = args[1];

let Bval = args[3];

if (typeof parseInt(Aval) !== 'number' || typeof parseInt(Bval) !== 'number') {

return shim.error('Expecting integer value for asset holding');

}

try {

await stub.putState(A, Buffer.from(Aval));

try {

await stub.putState(B, Buffer.from(Bval));

return shim.success();

} catch (err) {

return shim.error(err);

}

} catch (err) {

return shim.error(err);

}

}

// The Invoke method is called as a result of an application request to run the Smart Contract

// function to be called, with arguments.

**async Invoke(stub){**

let ret = stub.getFunctionAndParameters();

console.info(ret);

let method = this[ret.fcn];

if (!method){

console.error('no function of name:' + ret.fcn + ' found');

throw new Error('Received unknown function' + ret.fcn + ' invocation');

}

try {

let payload = await method(stub, ret.params, this);

return shim.success(payload);

} catch (err){

console.log(err);

return shim.error(err);

}

}

//Init characters and data

async initLedger(stub, args, thisClass){

console.info('====== START : initialize Ledger ======');

var time = thisClass.Now();

var timeFullString = time[0]+time[1]+time[2]+time[3]+time[4]+time[5];

var ymd = time[0]+time[1]+time[2];

var hms = time[3]+":"+time[4]+":"+time[5];

let Materials = [];

Materials.push({

Name: 'gochi',

Efficacy: 'good for body',

Color: 'red',

HarvestBatch: 'B1',

Action: '採收、初加工',

Place: '四川',

Weather: '雨後',

Number: '10',

Unit: '公斤',

Temperature: '27',

Fertilizer: '過磷酸鈣',

FirstBatch: 'F1',

Skill: '曝曬',

InspectBatch: '',

Inspecter: '',

Inspect: '',

TotalGrey: '',

SO2: '',

PurchaseBatch: '',

ContractNo: '',

ProduceBatch: '',

ProductNo: '',

ProductName: '',

OwnerID: 'POC0',

TimeStampDate: ymd,

TimeStampTime: hms,

});

…

Materials.push({

…

});

let TMC = [{

Name : 'TMC',

Token : 29850000,

TimeStampDate: ymd,

TimeStampTime: hms,

}];

let PoC = [{

Name : 'farmer1',

Token : 10000,

Role : 'farmer',

TimeStampDate: ymd,

TimeStampTime: hms,

},{

…

},{

Name : 'hospital2',

Token : 10000,

Role : 'hospital',

TimeStampDate: ymd,

TimeStampTime: hms,

}];

let TransactionRecord = [{

Action : 'upload',

Fee : 5,

Participant : 'POC1',

TimeStampDate: ymd,

TimeStampTime: hms,

},{

…

} ,{

Action : 'upload',

Fee : 5,

Participant : 'POC0',

TimeStampDate: ymd,

TimeStampTime: hms,

}];

for (let i = 0; i < Materials.length; i++) {

Materials[i].docType = 'material';

await stub.putState('MATERIAL' + i, Buffer.from(JSON.stringify(Materials[i])));

console.info('Added <--> ', Materials[i]);

let indexName = 'HarvestBatch~MATERIAL';

let HarvestBatchIndexKey = await stub.createCompositeKey(indexName, [Materials[i].HarvestBatch, 'MATERIAL'+i]);

console.info(HarvestBatchIndexKey);

// Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the marble.

// Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null character as value

//await stub.putState(HarvestBatchIndexKey, Buffer.from('\u0000'));

await stub.putState(HarvestBatchIndexKey, Buffer.from(JSON.stringify(Materials[i])));

let indexName2 = 'ProductNo~MATERIAL';

if(Materials[i].ProductNo != ''){

let ProductNoIndexKey = await stub.createCompositeKey(indexName2, [Materials[i].ProductNo, 'MATERIAL'+i]);

console.info(ProductNoIndexKey);

await stub.putState(ProductNoIndexKey, Buffer.from(JSON.stringify(Materials[i])));

}

}

for (let i=0; i< TMC.length; i++){

TMC[i].docType = 'tmc';

await stub.putState('TMC' + i, Buffer.from(JSON.stringify(TMC[i])));

console.info('Added <-->, ', TMC[i]);

}

for (let i=0; i< PoC.length; i++){

PoC[i].docType = 'poc';

await stub.putState('POC' + i, Buffer.from(JSON.stringify(PoC[i])));

console.info('Added <-->, ', PoC[i]);

}

for (let i=0; i< TransactionRecord.length; i++){

TransactionRecord[i].docType = 'transactionRecord';

await stub.putState('TRANSACTIONRECORD' + i, Buffer.from(JSON.stringify(TransactionRecord[i])));

console.info('Added <-->, ', TransactionRecord[i]);

}

console.info('============= END : initialize Ledger ===========');

}

// Upload data(Material)

async uploadData(stub, args, thisClass){

console.info('====== START : upload Data ======');

var time = thisClass.Now();

var timeFullString = time[0]+time[1]+time[2]+time[3]+time[4]+time[5];

var ymd = time[0]+time[1]+time[2];

var hms = time[3]+":"+time[4]+":"+time[5];

if(args.length !=24){

throw new Error('Incorrect number of arguments. Expecting 24');

}

var material = {

docType: 'material',

Name: args[0],

Efficacy: args[1],

Color: args[2],

HarvestBatch: args[3],

Action: args[4],

Place: args[5],

Weather: args[6],

Number: args[7],

Unit: args[8],

Temperature: args[9],

Fertilizer: args[10],

FirstBatch: args[11],

Skill: args[12],

InspectBatch: args[13],

Inspecter: args[14],

Inspect: args[15],

TotalGrey: args[16],

SO2: args[17],

PurchaseBatch: args[18],

ContractNo: args[19],

ProduceBatch: args[20],

ProductNo: args[21],

ProductName: args[22],

OwnerID: args[23],

TimeStampDate: ymd,

TimeStampTime: hms

};

let method = thisClass['getAllResults'];

//let queryResults = await method(stub,'MATERIAL',thisClass);

let querystartKey = 'MATERIAL'+'0';

let queryendKey = 'MATERIAL'+'99999';

let queryIterator = await stub.getStateByRange(querystartKey, queryendKey);

let queryResults = await method(queryIterator, false);

//let queryResultsParse = JSON.parse(queryResults);

await stub.putState("MATERIAL"+queryResults.length, Buffer.from(JSON.stringify(material)));

let indexName = 'HarvestBatch~MATERIAL';

let HarvestBatchIndexKey = await stub.createCompositeKey(indexName, [material.HarvestBatch, 'MATERIAL'+queryResults.length]);

console.info(HarvestBatchIndexKey);

// Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the marble.

// Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null character as value

//await stub.putState(HarvestBatchIndexKey, Buffer.from('\u0000'));

await stub.putState(HarvestBatchIndexKey, Buffer.from(JSON.stringify(material)));

indexName = 'ProductNo~MATERIAL';

let ProductNoIndexKey = await stub.createCompositeKey(indexName, [material.ProductNo, 'MATERIAL'+queryResults.length]);

console.info(ProductNoIndexKey);

await stub.putState(ProductNoIndexKey, Buffer.from(JSON.stringify(material)));

let queryPocResults = await stub.getState(args[23]);

let queryPocResultsParse = JSON.parse(queryPocResults);

let startKey = 'TRANSACTIONRECORD'+'0';

let endKey = 'TRANSACTIONRECORD'+'99999';

let iterator = await stub.getStateByRange(startKey, endKey);

//method = thisClass['getAllResults'];

let results = await method(iterator, false);

let todayRecord = [];

for(var i=0;i<results.length;i++){

if(results[i].Record.Action=="upload" && results[i].Record.TimeStampDate==time[0]+time[1]+time[2] && results[i].Record.Participant==args[23]){

todayRecord.push(results[i]);

}

}

if(todayRecord.length<10){

queryPocResultsParse.Token += 10;

await stub.putState(args[23], Buffer.from(JSON.stringify(queryPocResultsParse)));

let queryTMCResults = await stub.getState("TMC0");

let queryTMCResultsParse = JSON.parse(queryTMCResults);

queryTMCResultsParse.Token -= 10;

await stub.putState("TMC0",Buffer.from(JSON.stringify(queryTMCResultsParse)));

}

//JSON.stringify(response\_payloads[0].toString('utf8'))

let transactionRecord = {

Action: "upload", Fee:0, Participant: args[23], TimeStampDate: time[0]+time[1]+time[2], TimeStampTime:time[3]+time[4]+time[5]

};

await stub.putState("TRANSACTIONRECORD"+results.length,Buffer.from(JSON.stringify(transactionRecord)));

console.info('============= END : upload Data ===========');

}

async queryAllByKey(stub, args, thisClass){

console.info('===== START : query All By Key =====');

let startKey = args[0]+'0';

let endKey = args[0]+'99999';

let iterator = await stub.getStateByRange(startKey, endKey);

let method = thisClass['getAllResults'];

let results = await method(iterator, false);

return Buffer.from(JSON.stringify(results));

console.info('===== END : query All By Key =====');

}

async queryAllByKeyWithoutBuffer(stub, args, thisClass){

console.info('===== START : query All By Key =====');

let startKey = args[0]+'0';

let endKey = args[0]+'99999';

let iterator = await stub.getStateByRange(startKey, endKey);

let method = thisClass['getAllResults'];

let results = await method(iterator, false);

return results;

console.info('===== END : query All By Key =====');

}

async queryData(stub, args, thisClass){

console.info('====== START : query Data ======');

if (args.length != 2) {//Material Key , Poc Key(querier的Key)

throw new Error('Incorrect number of arguments. Expecting 2');

}

let queryPocResults = await stub.getState(args[1]);

let queryPocResultsParse = JSON.parse(queryPocResults);

if(queryPocResultsParse.Token>=5){

queryPocResultsParse.Token -=5;

await stub.putState(args[1],Buffer.from(JSON.stringify(queryPocResultsParse)));

let queryMaterialResults = await stub.getState(args[0]);

let queryMaterialResultsParse = JSON.parse(queryMaterialResults);

let queryPoc2Results = await stub.getState(queryMaterialResultsParse.OwnerID);

let queryPoc2ResultsParse = JSON.parse(queryPoc2Results);

queryPoc2ResultsParse.Token = queryPoc2ResultsParse.Token + 5 \* 0.98;

await stub.putState(queryMaterialResultsParse.OwnerID,Buffer.from(JSON.stringify(queryPoc2ResultsParse)));

let queryTMCResults = await stub.getState("TMC0");

let queryTMCResultsParse = JSON.parse(queryTMCResults);

queryTMCResultsParse.Token = queryTMCResultsParse.Token+ 5\*0.02;

await stub.putState("TMC0", Buffer.from(JSON.stringify(queryTMCResultsParse)));

var time = thisClass.Now();

//let method = thisClass['queryAllByKeyWithoutBuffer'];

let startKey = 'TRANSACTIONRECORD'+'0';

let endKey = 'TRANSACTIONRECORD'+'99999';

let iterator = await stub.getStateByRange(startKey, endKey);

let method = thisClass['getAllResults'];

let results = await method(iterator, false);

//let queryTransactionRecordResults = await method(stub,'TRANSACTIONRECORD', thisClass);

//let queryTransactionRecordResultsParse = JSON.parse(queryTransactionRecordResults);

let transactionRecord = {

Action: "get", Fee:5, Participant: args[1], TimeStampDate: time[0]+time[1]+time[2], TimeStampTime:time[3]+time[4]+time[5]

};

await stub.putState("TRANSACTIONRECORD"+results.length,Buffer.from(JSON.stringify(transactionRecord)));

return queryMaterialResults;

}else{ //not enough for 5 token

//redirect to token store page

console.log("redirect to token store page");

}

console.info('============= END : Upload Data ===========');

}

//query data with partial composite key

async queryHarvestBatch(stub, args, thisClass){

…

}

async queryProductNo(stub, args, thisClass){

…

}

async queryProductNoOnly(stub, args, thisClass){

…

}

async tokenStore(stub, args, thisClass){

…

}

async queryHistoryByKey(stub, args, thisClass){

…

}

async queryByKey(stub, args, thisClass){

…

}

async getAllResults(iterator, isHistory) {

…

}

Now(){

var y,mon,d,h,min,s;

var Now=new Date();

y= String(Now.getFullYear());

mon= String(Now.getMonth()+1);

d= String(Now.getDate());

h= String(Now.getHours());

min= String(Now.getMinutes());

s= String(Now.getSeconds());

//console.log(mon+" type: "+ typeof mon+" length: "+mon.length);

if(mon.length==1) mon = "0"+mon;

if(d.length==1) d = "0"+d;

if(h.length==1) h = "0"+h;

if(min.length==1) min="0"+min;

if(s.length==1) s="0"+s;

var all = [y,mon,d,h,min,s];

return all

}

};

//Let the chaincode start.

shim.start(new Chaincode());

## 二、Conversion from Client side

The difference between Client side program is more than smart contract. We install Chaincode on peer and instantiate Chaincode on Channel in Fabric. Client side communicates with Channel and peer, orderer and Chaincode on it through SDK. But we use address to define different application and resource in Sawtooth. Client side communicates with REST API, then REST API communicates with Validator, and then call Transaction Processor.

The program example of Sawtooth Client side, written by javascript:

state.js

'use strict'

const $ = require('jquery')

const {createHash} = require('crypto')

const protobuf = require('sawtooth-sdk/protobuf')

const {

createContext,

Signer

} = require('sawtooth-sdk/signing')

const secp256k1 = require('sawtooth-sdk/signing/secp256k1')

// Config variables

const KEY\_NAME = 'transfer-chain.keys'

const API\_URL = 'http://localhost:8000/api'

const FAMILY = 'transfer-chain'

const VERSION = '0.0'

const PREFIX = '19d832'

// Fetch key-pairs from localStorage

const getKeys = () => {

const storedKeys = localStorage.getItem(KEY\_NAME)

if (!storedKeys) return []

return storedKeys.split(';').map((pair) => {

const separated = pair.split(',')

return {

public: separated[0],

private: separated[1]

}

})

}

// Create new key-pair

const makeKeyPair = () => {

const context = createContext('secp256k1')

const privateKey = context.newRandomPrivateKey()

return {

public: context.getPublicKey(privateKey).asHex(),

private: privateKey.asHex()

}

}

// Save key-pairs to localStorage

const saveKeys = keys => {

const paired = keys.map(pair => [pair.public, pair.private].join(','))

localStorage.setItem(KEY\_NAME, paired.join(';'))

}

// Fetch current Sawtooth Tuna Chain state from validator

const getState = cb => {

$.get(`${API\_URL}/state?address=${PREFIX}`, ({ data }) => {

cb(data.reduce((processed, datum) => {

if (datum.data !== '') {

const parsed = JSON.parse(atob(datum.data))

if (datum.address[7] === '0') processed.assets.push(parsed)

if (datum.address[7] === '1') processed.transfers.push(parsed)

}

return processed

}, {assets: [], transfers: []}))

})

}

// Submit signed Transaction to validator

const submitUpdate = (payload, privateKeyHex, cb) => {

// Create signer

const context = createContext('secp256k1')

const privateKey = secp256k1.Secp256k1PrivateKey.fromHex(privateKeyHex)

const signer = new Signer(context, privateKey)

// Create the TransactionHeader

const payloadBytes = Buffer.from(JSON.stringify(payload))

const transactionHeaderBytes = protobuf.TransactionHeader.encode({

familyName: FAMILY,

familyVersion: VERSION,

inputs: [PREFIX],

outputs: [PREFIX],

signerPublicKey: signer.getPublicKey().asHex(),

batcherPublicKey: signer.getPublicKey().asHex(),

dependencies: [],

payloadSha512: createHash('sha512').update(payloadBytes).digest('hex')

}).finish()

// Create the Transaction

const transactionHeaderSignature = signer.sign(transactionHeaderBytes)

const transaction = protobuf.Transaction.create({

header: transactionHeaderBytes,

headerSignature: transactionHeaderSignature,

payload: payloadBytes

})

// Create the BatchHeader

const batchHeaderBytes = protobuf.BatchHeader.encode({

signerPublicKey: signer.getPublicKey().asHex(),

transactionIds: [transaction.headerSignature]

}).finish()

// Create the Batch

const batchHeaderSignature = signer.sign(batchHeaderBytes)

const batch = protobuf.Batch.create({

header: batchHeaderBytes,

headerSignature: batchHeaderSignature,

transactions: [transaction]

})

// Encode the Batch in a BatchList

const batchListBytes = protobuf.BatchList.encode({

batches: [batch]

}).finish()

// Submit BatchList to Validator

$.post({

url: `${API\_URL}/batches`,

data: batchListBytes,

headers: {'Content-Type': 'application/octet-stream'},

processData: false,

success: function( resp ) {

var id = resp.link.split('?')[1]

$.get(`${API\_URL}/batch\_statuses?${id}&wait`, ({ data }) => cb(true))

},

error: () => cb(false)

})

}

module.exports = {

getKeys,

makeKeyPair,

saveKeys,

getState,

submitUpdate

}

The program example of Fabric Client side, written by javascript:

invoke.js

'use strict';

/\*

\* Copyright IBM Corp All Rights Reserved

\*

\* SPDX-License-Identifier: Apache-2.0

\*/

/\*

\* Chaincode Invoke

\*/

var Fabric\_Client = require('fabric-client');

var path = require('path');

var util = require('util');

var os = require('os');

//

var fabric\_client = new Fabric\_Client();

// setup the fabric network

var channel = fabric\_client.newChannel('mychannel');

var peer = fabric\_client.newPeer('grpc://localhost:7051');

channel.addPeer(peer);

var order = fabric\_client.newOrderer('grpc://localhost:7050')

channel.addOrderer(order);

//

var member\_user = null;

var store\_path = path.join(\_\_dirname, 'hfc-key-store');

console.log('Store path:'+store\_path);

var tx\_id = null;

// create the key value store as defined in the fabric-client/config/default.json 'key-value-store' setting

Fabric\_Client.newDefaultKeyValueStore({ path: store\_path

}).then((state\_store) => {

// assign the store to the fabric client

fabric\_client.setStateStore(state\_store);

var crypto\_suite = Fabric\_Client.newCryptoSuite();

// use the same location for the state store (where the users' certificate are kept)

// and the crypto store (where the users' keys are kept)

var crypto\_store = Fabric\_Client.newCryptoKeyStore({path: store\_path});

crypto\_suite.setCryptoKeyStore(crypto\_store);

fabric\_client.setCryptoSuite(crypto\_suite);

// get the enrolled user from persistence, this user will sign all requests

return fabric\_client.getUserContext('user1', true);

}).then((user\_from\_store) => {

if (user\_from\_store && user\_from\_store.isEnrolled()) {

console.log('Successfully loaded user1 from persistence');

member\_user = user\_from\_store;

} else {

throw new Error('Failed to get user1.... run registerUser.js');

}

// get a transaction id object based on the current user assigned to fabric client

tx\_id = fabric\_client.newTransactionID();

console.log("Assigning transaction\_id: ", tx\_id.\_transaction\_id);

// createCar chaincode function - requires 5 args, ex: args: ['CAR12', 'Honda', 'Accord', 'Black', 'Tom'],

// changeCarOwner chaincode function - requires 2 args , ex: args: ['CAR10', 'Dave'],

// must send the proposal to endorsing peers

var request = {

//targets: let default to the peer assigned to the client

chaincodeId: 'fabcar',

fcn: '',

args: [''],

chainId: 'mychannel',

txId: tx\_id

};

// send the transaction proposal to the peers

return channel.sendTransactionProposal(request);

}).then((results) => {

var proposalResponses = results[0];

var proposal = results[1];

let isProposalGood = false;

if (proposalResponses && proposalResponses[0].response &&

proposalResponses[0].response.status === 200) {

isProposalGood = true;

console.log('Transaction proposal was good');

} else {

console.error('Transaction proposal was bad');

}

if (isProposalGood) {

console.log(util.format(

'Successfully sent Proposal and received ProposalResponse: Status - %s, message - "%s"',

proposalResponses[0].response.status, proposalResponses[0].response.message));

// build up the request for the orderer to have the transaction committed

var request = {

proposalResponses: proposalResponses,

proposal: proposal

};

// set the transaction listener and set a timeout of 30 sec

// if the transaction did not get committed within the timeout period,

// report a TIMEOUT status

var transaction\_id\_string = tx\_id.getTransactionID(); //Get the transaction ID string to be used by the event processing

var promises = [];

var sendPromise = channel.sendTransaction(request);

promises.push(sendPromise); //we want the send transaction first, so that we know where to check status

// get an eventhub once the fabric client has a user assigned. The user

// is required bacause the event registration must be signed

let event\_hub = channel.newChannelEventHub(peer);

// using resolve the promise so that result status may be processed

// under the then clause rather than having the catch clause process

// the status

let txPromise = new Promise((resolve, reject) => {

let handle = setTimeout(() => {

event\_hub.unregisterTxEvent(transaction\_id\_string);

event\_hub.disconnect();

resolve({event\_status : 'TIMEOUT'}); //we could use reject(new Error('Trnasaction did not complete within 30 seconds'));

}, 3000);

event\_hub.registerTxEvent(transaction\_id\_string, (tx, code) => {

// this is the callback for transaction event status

// first some clean up of event listener

clearTimeout(handle);

// now let the application know what happened

var return\_status = {event\_status : code, tx\_id : transaction\_id\_string};

if (code !== 'VALID') {

console.error('The transaction was invalid, code = ' + code);

resolve(return\_status); // we could use reject(new Error('Problem with the tranaction, event status ::'+code));

} else {

console.log('The transaction has been committed on peer ' + event\_hub.getPeerAddr());

resolve(return\_status);

}

}, (err) => {

//this is the callback if something goes wrong with the event registration or processing

reject(new Error('There was a problem with the eventhub ::'+err));

},

{disconnect: true} //disconnect when complete

);

event\_hub.connect();

});

promises.push(txPromise);

return Promise.all(promises);

} else {

console.error('Failed to send Proposal or receive valid response. Response null or status is not 200. exiting...');

throw new Error('Failed to send Proposal or receive valid response. Response null or status is not 200. exiting...');

}

}).then((results) => {

console.log('Send transaction promise and event listener promise have completed');

// check the results in the order the promises were added to the promise all list

if (results && results[0] && results[0].status === 'SUCCESS') {

console.log('Successfully sent transaction to the orderer.');

} else {

console.error('Failed to order the transaction. Error code: ' + results[0].status);

}

if(results && results[1] && results[1].event\_status === 'VALID') {

console.log('Successfully committed the change to the ledger by the peer');

} else {

console.log('Transaction failed to be committed to the ledger due to ::'+results[1].event\_status);

}

}).catch((err) => {

console.error('Failed to invoke successfully :: ' + err);

});

## 三、Reference

1. <https://github.com/hyperledger/education/tree/master/LFS171x/sawtooth-material/sawtooth-tuna>