Lab 4.1: Step-by-Step SimpleContract Lifecycle Management

Assuming `test-network` is up and running, let's deploy and test `SimpleContract`.

First, open two terminal windows. We will use them to act on behalf of two different organizations—`Org1MSP` and `Org2MSP`.

Navigate to `fabric-samples/test-network` and define all necessary environment variables for `Org1MSP`. Note that TLS is enabled in `test-network`.

```
# cd $HOME/go/src/github.com/hyperledger/fabric-samples/test-network
# export CORE_PEER_TLS_ENABLED=true
# export CORE_PEER_LOCALMSPID="Org1MSP"
# export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org1.exa
mple.com/peers/peer0.org1.example.com/tls/ca.crt
# export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org1.example
.com/users/Admin@org1.example.com/msp
# export CORE_PEER_ADDRESS=localhost:7051
```

During the development environment setup phase, we have downloaded Hyperledger Fabric binaries including `peer`. They are located in the `fabric-samples/bin` folder and utilize configurations stored in `fabric-samples/config`. Therefore, we can update the `PATH` variable and set `FABRIC_CFG_PATH` to simplify `peer` binary usage.

```
# export PATH=${PWD}/../bin:$PATH
```

```
# export FABRIC_CFG_PATH=$PWD/../config/
```

Similarly, configure environment variables for `Org2MSP`.

```
# cd $HOME/go/src/github.com/hyperledger/fabric-samples/test-network
# export CORE_PEER_TLS_ENABLED=true
# export CORE_PEER_LOCALMSPID="Org2MSP"
# export
CORE_PEER_TLS_ROOTCERT_FILE=${PWD}/organizations/peerOrganizations/org2.exa
mple.com/peers/peer0.org2.example.com/tls/ca.crt
# export
CORE_PEER_MSPCONFIGPATH=${PWD}/organizations/peerOrganizations/org2.example
.com/users/Admin@org2.example.com/msp
# export CORE_PEER_ADDRESS=localhost:9051
# export PATH=${PWD}/../bin:$PATH
# export FABRIC_CFG_PATH=$PWD/../config/
```

Now, in the `Org1MSP` terminal window we can run a `peer channel list` command to confirm that we are able to use the `peer` binary without further adjustments.

```
# peer channel list

Channels peers has joined:
mychannel
```

Once a proper environment is established, let’s start the chaincode deployment flow. First, create a chaincode package right in the `test-network` directory.

```
# peer lifecycle chaincode package simple_chaincode.tar.gz --path
../lfd272/chaincodes/simple_chaincode --lang node --label
simple_chaincode_1.0
```

Note that packaging is performed once on behalf of any organization. The created package can be further used by both organizations.

Next, install the package on peers of both organizations. The `install` command should be issued twice—once on behalf of each organization.

```
# peer lifecycle chaincode install simple_chaincode.tar.gz

Chaincode code package identifier:
simple_chaincode_1.0:5e11633859d9976c872a38b13521279adf32d14ba8bffa6e29da4b
43adcde680
```

Save the package identifier and approve the chaincode definition. The chaincode package identifier can also be queried by the `peer lifecycle chaincode queryinstalled` command. Approval should be submitted by both `Org1MSP` and `Org2MSP`.

```
# export
PACKAGE_ID=simple_chaincode_1.0:5e11633859d9976c872a38b13521279adf32d14ba8b
ffa6e29da4b43adcde680
# peer lifecycle chaincode approveformyorg -o localhost:7050
--ordererTLSHostnameOverride orderer.example.com --channelID mychannel
--name simple_chaincode --version 1.0 --package-id $PACKAGE_ID --sequence 1
--tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem

txid [89f186b8344aa5b3d88b64614d4e0bb4959d9f33f2fa7899ba2d582fe2c4bffd]
committed with status (VALID) at localhost:7051
```

As the interaction with peers happens outside of the Docker network, we should use `localhost:7050` as the orderer address, because `orderer.example.com` can not be resolved. However, TLS certificates use domain names inside, so we should notify the `peer` binary that `localhost:7050` in the command is actually the same as `orderer.example.com:7050` in the certificate. The `--ordererTLSHostnameOverride` parameter comes in handy in our case.

Before committing the chaincode definition to the channel, we should ensure that a number of submitted approvals is sufficient. For this purpose, the `peer lifecycle chaincode checkcommitreadiness` command can be used.

```
# peer lifecycle chaincode checkcommitreadiness --channelID mychannel
--name simple_chaincode --version 1.0 --sequence 1 --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem --output json

{
    "approvals": {
```



```
        "Org1MSP": true,
        "Org2MSP": true
    }
}
```

The output signalizes that both organizations have submitted their approvals. Therefore, the chaincode definition can be committed.

```
# peer lifecycle chaincode commit -o localhost:7050
--ordererTLSHostnameOverride orderer.example.com --channelID mychannel
--name simple_chaincode --version 1.0 --sequence 1 --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem --peerAddresses
localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.ex
ample.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.ex
ample.com/tls/ca.crt

txid [ca2a121a9b3e05d1da7386ea235216761eb7e302697a5fff2b8f5c2c68dcb332]
committed with status (VALID) at localhost:9051
txid [ca2a121a9b3e05d1da7386ea235216761eb7e302697a5fff2b8f5c2c68dcb332]
committed with status (VALID) at localhost:7051
```

The `commit` transaction is submitted to peers of both `Org1MSP` and `Org2MSP`. If all targeted peers return successful responses, the chaincode definition is committed to the channel. To confirm this, use the `peer lifecycle chaincode querycommitted` command.

```
# peer lifecycle chaincode querycommitted --channelID mychannel --name
simple_chaincode --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem

Committed chaincode definition for chaincode 'simple_chaincode' on channel
'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin:
vscc, Approvals: [Org1MSP: true, Org2MSP: true]
```

Finally, we should make sure that the business logic of `SimpleContract` is correct. Let's run the `put→get→del→get` sequence of invocations for the `(k, v)` key-value pair.

```
# peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
simple_chaincode --peerAddresses localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.ex
ample.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.ex
ample.com/tls/ca.crt -c '{"function":"put","Args":["k", "v"]}'

Chaincode invoke successful. result: status:200

# peer chaincode query -C mychannel -n simple_chaincode -c
'{"function":"get","Args":["k"]}'

v

# peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n
simple_chaincode --peerAddresses localhost:7051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.ex
ample.com/tls/ca.crt --peerAddresses localhost:9051 --tlsRootCertFiles
${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.ex
ample.com/tls/ca.crt -c '{"function":"del","Args":["k"]}'

Chaincode invoke successful. result: status:200

# peer chaincode query -C mychannel -n simple_chaincode -c
'{"function":"get","Args":["k"]}'

Error: endorsement failure during query. response: status:500
message:"error in simulation: transaction returned with failure: Error: The
asset k does not exist"
```

We should face an error during the execution of the last command, because the `k` key is deleted already. We can check chaincode logs to trace the error. To access chaincode logs, run the `docker logs` command passing the chaincode container name as an argument.


```
# docker logs
dev-peer0.org1.example.com-simple_chaincode_1.0-5e11633859d9976c872a38b1352
1279adf32d14ba8bffa6e29da4b43adcde680
```

As you can see, the chaincode deployment workflow is pretty straightforward. However, there are multiple spots for errors and typos. Therefore, the `test-network/network.sh` script provides a high-level helper function encapsulating the CLI commands execution—`deployCC`. To perform the whole workflow described in this lab, we can simply run the `deployCC` command with appropriate arguments.

```
# ./network.sh deployCC -ccn simple_cc -ccp
../lfd272/chaincodes/simple_chaincode -ccl javascript -ccv 1.0
```

Note that `deployCC` accepts `javascript` as a programming language parameter, while the bare `package` command utilizes the `node` value for the same parameter.

The whole list of possible helper commands and their arguments included in the `network.sh` script can be displayed by running `./network.sh -h`.

