LFS272:
HYPERLEDGER FABRIC ADMINISTRATION

- V.5.30.2019 -

Lab 1. Environment Setup and Fabric Installation

System Prerequisites

It is safe to allocate at a very minimum 30GB of disk space towards using Hyperledger Fabric in a development environment (close to 7GB will account for just running Docker containers). To ensure the containers do not slow down and cause the environment to hang, it is best to allocate at least 4GB of RAM towards use with this course. Lastly, your CPU should be at least dual core or higher.

Ubuntu 16.04 is the recommended operating system for this course. Ubuntu environments can be run in an Ubuntu desktop OS natively installed, or in a virtual machine (VM). Popular options for VMs include VirtualBox, AWS EC2, and Vagrant.

Note: *Lab exercises in this course are interdependent and build on one another.*

cURL

First, let's update our package list and install any newers versions using **apt**:

```
sudo apt update
sudo apt -y upgrade
```

Next, we will install curl using **apt**:

```
sudo apt install curl
```

GIT

Similar to how we installed curl, we will now install git using **apt**:

```
sudo apt install git
```

Python

Since we are running Ubuntu 16.04, we should already have the Python version we need automatically installed. We will still perform a double check to ensure Python is installed:

```
python2.7 —version
```

If for some unknown reason you return an error, you can install it using the following command:

```
sudo apt install -y python-minimal
```

Docker

Before moving on, let's update our package list:

```
sudo apt update
```

We will now use **apt** to install a few packages Docker is dependent upon:

```
sudo apt install apt-transport-https ca-certificates gnpg-agent
software-properties-common
```

Now we need to curl down our gpg key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -
```

Next, let's add the Docker repository (stable version):

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Once more, let's update our package list:

```
sudo apt update
```

Now we will use **apt** to install the Docker Community Edition, using **-y** to bypass all yes/no questions:

```
sudo apt -y install docker-ce
```

Next, you need to add your current username to the Docker Group so you can access it in case of "non-root-user" issues. It's important here to replace the brackets placeholder with the name of your user account.

```
sudo usermod -aG docker <YourUserNameGoesHere>
```

Docker-Compose

First, we need to pull **docker-compose** using curl:

```
sudo curl -L
https://github.com/docker/compose/releases/download/1.18.0/docker-comp
ose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose
```

We need to change the permissions for docker-compose so we can execute it later:

```
sudo chmod +x /usr/local/bin/docker-compose
```

Let's perform a reboot to ensure everything we've done is placed into effect.

```
sudo reboot now
```

Project Folder

Please change directories into your desktop:

```
cd ~/Desktop
```

Use the following command to curl down the **fabric-samples** project folder, and Docker images for Hyperledger Fabric v1.4:

```
curl -sSL http://bit.ly/2ysbOFE | bash -s 1.4.0
```

Please verify that a new **fabric-samples** folder is now on your desktop:

```
ls
```

Now, take a quick look into what this folder holds.

```
cd fabric-samples && ls
```

Sanity Checks

Lastly, verify that everything was installed properly:

```
git -v
docker-compose -v
docker -v
```

SUCCESS!

Starter Network

To give a simple overview of what the starter network is, it is an extremely scaled down, barebones Hyperledger Fabric network. It has one peer, one organization, and a single Solo orderer. We begin with a network such as this so that we can show how a sysadmin can scale a network and its operations from a very simple level all the way to a fairly complex one.

```
cd fabric-samples
```

Bootstrap the Basics

Now, in your command line, run:

```
wget https://s3.us-east-2.amazonaws.com/lfx-start1/startup.sh
```

```
chmod u+x ./startup.sh
```

```
./startup.sh
```

Notes

NOTE ON YAML FILES: Throughout this course, we will be working with YAML files. It is vital that you understand how to structure and format YAML files, or else you will run into many issues. If you are not familiar with YAML grammar, we recommend that you familiarize yourself with the basics first before you move on with the course. You can learn more about YAML syntax at https://yaml.org/.

Note on Lab Structure: Sometimes, things may seem to be unnecessary or more difficult than needed, but this is always for a strategic reason. We want everyone to understand the key details that may seem unnecessary but will help us later in the course. It is important to understand the entire process, and when there are shortcuts we will discuss them, but it remains vital to understand the entire process.

Lab 2. Expanding Our Organization Members

Objectives

- Create the peer definition.
- Deploy the peer.
- Add the peer to a channel.

Important Note:

YAML is a key configuration language that we use throughout Hyperledger Fabric. Numerous simple configuration issues/startup issues arise due to improper use of YAML grammar and formatting. Things like spacing are simple "need-to-know" capabilities in order to be successful. To learn more about YAML, visit: *YAML Ain't Markup Language (YAML™) Version 1.2*.

Creating the Peer

To add a network peer, we must bring up another separate container. So, we must first do:

```
cd startFiles
```

This is where our network composition files and artifacts lie.

Edit the docker-compose.yml File

Please open **`docker-compose.yml`** in your favorite code editor.

The first thing we will do inside the **docker-compose.yml** file is to create a new container definition (in the *services* section). Our container name will be **peer1.org1.example.com.** We can specify a lot of the container configuration details to be very similar to **peer1** since the two are from the same org (which have similar details). We can add the following information under the *services* section:

```
  peer1.org1.example.com:
    container_name: peer1.org1.example.com
    image: hyperledger/fabric-peer
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer1.org1.example.com
      - CORE_LOGGING_PEER=info
      - CORE_CHAINCODE_LOGGING=debug
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
      - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
      -
CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_basic

    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start
    ports:
      - 8051:7051
      - 8053:7053
    volumes:
        - /var/run/:/host/var/run/
        -
./crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.ex
ample.com/msp:/etc/hyperledger/msp/peer
        -
./crypto-config/peerOrganizations/org1.example.com/users:/etc/hyperled
ger/msp/users
        - ./config:/etc/hyperledger/configtx
    depends_on:
      - orderer.example.com
    networks:
      - basic
```

One of the most important/detrimental things to pay attention to here are the environment variables, which set values like the MSP identity, logging levels, etc. The other important thing is our volumes. The *volumes* section is how we expose/map our local folders, like the

crypto-config or **network-artifacts** (aka our configuration directory) configuration folders to our Docker container for access.

Edit the **crypto-config** File

Since we are adding a new peer to **Org1**, we must reconfigure what the template for **Org1** is going to be. We need to edit the generation template for **Org1** so that it accommodates the creation of two user identities (in the form of x509 certificates).

Open **crypto-config.yaml** in your preferred code editor.

What we are looking to edit is the value for **Template.Count**, which can be found under the *org1* section. We will be changing that value from 1 to 2. So that section should read:

```
Template:
  Count: 2
  # Start: 5
  # Hostname: {{.Prefix}}{{.Index}} # default
```

Next, we need to re-generate the crypto certificates from our updated **crypto-config** using **cryptogen**.This time, we can use a command to create new certificates without totally removing the old ones.

```
cd ~/Desktop/fabric-samples/startFiles
../bin/cryptogen extend --config=./crypto-config.yaml
```

We can see our newly created certificates here:

```
ls crypto-config/peerOrganizations/org1.example.com/peers
```

Generating the Initial Configuration

First, we need to generate our genesis block.

```
../bin/configtxgen -profile OneOrgOrdererGenesis \
-outputBlock ./config/genesis.block
```

We can look at this newly created genesis block by using the **inspectBlock** command.

```
../bin/configtxgen -inspectBlock ./config/genesis.block
```

Start the Peer Container

Now, let's use our **`docker-compose`** definition to quickly bring up **`peer1.org1`**

```
docker-compose -f docker-compose.yml up \
-d peer0.org1.example.com peer1.org1.example.com cli
```

Let's return our peer containers to confirm they were started with no issues.

```
docker ps --filter name=peer
```

Since we joined **`peer0`** in bootstrap, we must also do the same for **`peer1`**. First, we must enter into our **`peer`** container.

```
docker exec -it peer1.org1.example.com bash
```

We will set the identification Path to the Admin for Org1, so that we are authorized to pull and edit the configuration. (**Note**: The reason the path is **`/etc/hyperledger/msp`** is because we mapped it that way at container startup using volumes.)

```
export
CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.
com/msp
```

Next, we need to pull the genesis block for the current channel we would like to append our peer to.

```
peer channel fetch oldest allarewelcome.block -c allarewelcome \
--orderer orderer.example.com:7050
```

Join the Peer to the Current Channel

```
peer channel join -b allarewelcome.block
```

Now, we can check and see if our peer is actually joined.

```
peer channel list
```

Now please exit out of the **`peer1`** container.

SUCCESS!

Lab 3. State Database

Objective

- Set up and configure CouchDB as the state database for all peers in the network.

To avoid confusion, and make configuration easier, we will use our **`docker-compose.yml`** file to override any state database configuration default that should arise during startup. LevelDB is the default state database. We would like to show you how to configure CouchDB for all peer operations as the state database.

Every peer of the network must possess one ledger copy and one personal state database. Let's start by creating the definition for a **`couchDB`** container so that all our peers can access it. Let's check to make sure we have a **`couchDB`** image.

```
docker images hyperledger/fabric-couchdb
```

If we don't see it, we can pull down the **`couchDB`** Docker image that we can use it.

```
docker pull hyperledger/fabric-couchdb
```

Please open **`docker-compose.yml`** in your favorite code editor.

```
nano docker-compose.yml
```

Defining CouchDB in YAML Startup

Because each peer in the network must have their own **couchDB** instance, it is important to create as many definitions as there are peers in the network. We will walk through the first definition of **couchDB0**, which will be a database for **peer0.org1.example.com**. Then, you can follow this example as a template to create the databases for the rest of the peers in the network.

Under the *services* section of the file, create a new container definition for **couchdb**. We will uniquely identify it by concatenating **couchDB** with the **peerName** it was meant for, and reference the image we just pulled. Create a new container definition in the *services* section:

```
couchdbOrg1Peer0:
    container_name: couchdbOrg1Peer0
    image: hyperledger/fabric-couchdb
```

For our environment, we will populate it with a username and password which correlates to each peer we set it for.

```
environment:
  - COUCHDB_USER=peer0.Org1
  - COUCHDB_PASSWORD=password
```

Lastly, the default port we want to expose for **couchdb** is 5984 (which is how we can access Fauxton as well).

```
    ports:
      - "5984:5984"
    networks:
      - basic
```

Trying It Yourself

Here is the total result of what we just did:

```
couchdbOrg1Peer0:
  container_name: couchdbOrg1Peer0
```

```
image: hyperledger/fabric-couchdb
environment:
  - COUCHDB_USER=peer0.Org1
  - COUCHDB_PASSWORD=password
ports:
  - "5984:5984"
networks:
  - basic
```

Using this as a template, create the **couchDB** container definition for **peer1.Org1.example.com**. Please remember to make sure you give each container a:

1. Unique container name (For example, this new container will be called **couchdbOrg1Peer1**)
2. Different Port specification for Host in your port mapping section (In our example, **couchDB 2** may have **6984:5984**). We adjust the HOST port on our Port mapping to make sure we avoid an internal network port conflict for the couchDB instances.

Assigning Dependency

Now we must assign the container dependency (aka Docker container linking) for each peer and its respective container. This is so that peers never start up without the appropriate state database container.

Since we named our CouchDB verbosely, it should be easy to figure out which container databases are meant to be linked to their respective peer. Under the *services* section in the **peer0.org1.example.com** definition, add in the **depends_on** section:

```
  - couchdbOrg1Peer0
```

Now we can do the same for **peer1.org1.example.com**. Under **depends_on** add:

```
  - couchdbOrg1Peer1
```

Override the Core Peer Environment Variables

Lastly, to override the default values (relating to the database configuration) for each peer that arises from **`core.yaml`**, we must add environment variables to each peer's container definition. For **`peer0.Org1`**, we would add this to the environment section:

```
- CORE_LEDGER_STATE_STATEDATABASE=CouchDB
- CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdbOrg1Peer0:5
  984
```

Please add two more variables, which secures database access to only those with the Username, and password.

```
- CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=peer0.Org1
- CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=password
```

Try it Yourself: Please provide the same four environment variables for **`peer1.Org1`** as well**.** Just remember to replace the unique container name on the **`COUCHDBADDRESS`** variable.

Still Confused?

For a full reference of how your **`docker-compose.yml`** file should currently look, please check out the *Reference* section at the bottom of this document, for a full documentation of how the file should look like.

Test It Out

Now we can start Fabric, and test this out.

```
docker-compose -f docker-compose.yml up \
-d ca.example.com orderer.example.com couchdbOrg1Peer0 \
peer0.org1.example.com couchdbOrg1Peer1 peer1.org1.example.com cli
```

Note: Since our peers are dependent on their respective **couchDB** containers, we always start them up before the peer.

Now, let's do a search to list our running **couchDB** containers.

```
docker ps --filter "name=couchdb"
```

Lastly, please confirm your peers are still a part of the channel. If they are not, it may be as result of us re-starting our peers, we may need to rerun the commands:

```
docker exec -it peer0.org1.example.com bash
```

```
peer channel fetch oldest allarewelcome.block -c allarewelcome \
--orderer orderer.example.com:7050
```

```
peer channel join -b allarewelcome.block
```

Try it Yourself: Now rerun the same process for peer1:

Extra: From your host, you can run **curl http://localhost:5984** to return whether **couchdb** is up and running to your command line. Or, just visit it in the browser to see the Fauxton interface.

SUCCESS!

REFERENCE:

```
#
# Copyright IBM Corp All Rights Reserved
#
# SPDX-License-Identifier: Apache-2.0
#
version: '2'

networks:
  basic:

services:

  ca.example.com:
    image: hyperledger/fabric-ca
    environment:
      - FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server
      - FABRIC_CA_SERVER_CA_NAME=ca.example.com
      -
FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/
ca.org1.example.com-cert.pem
      -
FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/3
29dac791fb648e0121c8fc7e787287c3085a5f7200750a47a7a468c2a11f32d_sk
    ports:
      - "7054:7054"
    volumes:
      -
./crypto-config/peerOrganizations/org1.example.com/ca/:/etc/hyperledge
r/fabric-ca-server-config
    container_name: ca.example.com
    networks:
      - basic
```

```
  orderer.example.com:
    container_name: orderer.example.com
    image: hyperledger/fabric-orderer
    environment:
      - ORDERER_GENERAL_LOGLEVEL=info
      - ORDERER_GENERAL_LISTENADDRESS=0.0.0.0
      - ORDERER_GENERAL_GENESISMETHOD=file
      -
ORDERER_GENERAL_GENESISFILE=/etc/hyperledger/configtx/genesis.block
      - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
      - ORDERER_GENERAL_LOCALMSPDIR=/etc/hyperledger/msp/orderer/msp
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/orderer
    command: orderer
    ports:
      - 7050:7050
    volumes:
        - ./config/:/etc/hyperledger/configtx
        -
./crypto-config/ordererOrganizations/example.com/orderers/orderer.exam
ple.com/:/etc/hyperledger/msp/orderer
        -
./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.ex
ample.com/:/etc/hyperledger/msp/peerOrg1
    networks:
      - basic

  peer0.org1.example.com:
    container_name: peer0.org1.example.com
    image: hyperledger/fabric-peer
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer0.org1.example.com
      - CORE_LOGGING_PEER=info
      - CORE_CHAINCODE_LOGGING_LEVEL=info
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
```

```
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      # # the following setting starts chaincode containers on the
same
      # # bridge network as the peers
      # # https://docs.docker.com/compose/networking/
      -
CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_basic
       - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
       -
      CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdbOrg1Peer0:5
      984
       - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=peer0.Org1
       - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=password
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start
    # command: peer node start --peer-chaincodedev=true
    ports:
      - 7051:7051
      - 7053:7053
    volumes:
        - /var/run/:/host/var/run/
        -
./crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.ex
ample.com/msp:/etc/hyperledger/msp/peer
        -
./crypto-config/peerOrganizations/org1.example.com/users:/etc/hyperled
ger/msp/users
        - ./config:/etc/hyperledger/configtx
    depends_on:
      - orderer.example.com
      - couchdbOrg1Peer0
    networks:
      - basic

  couchdbOrg1Peer0:
```

```
    container_name: couchdbOrg1Peer0
    image: hyperledger/fabric-couchdb
    environment:
      - COUCHDB_USER=peer0.Org1
      - COUCHDB_PASSWORD=password
    ports:
      - "5984:5984"
    networks:
      - basic

  peer1.org1.example.com:
    container_name: peer1.org1.example.com
    image: hyperledger/fabric-peer
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_PEER_ID=peer1.org1.example.com
      - CORE_LOGGING_PEER=info
      - CORE_CHAINCODE_LOGGING_LEVEL=info
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/
      - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
       - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
       -
      CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdbOrg1Peer1:5
      984
       - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=peer1.Org1
       - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=password

       # # the following setting starts chaincode containers on the
same
       # # bridge network as the peers
       # # https://docs.docker.com/compose/networking/
       -
CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_basic
```

```
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric
    command: peer node start
    # command: peer node start --peer-chaincodedev=true
    ports:
      - 8051:7051
      - 8053:7053
    volumes:
        - /var/run/:/host/var/run/
        -
./crypto-config/peerOrganizations/org1.example.com/peers/peer1.org1.ex
ample.com/msp:/etc/hyperledger/msp/peer
        -
./crypto-config/peerOrganizations/org1.example.com/users:/etc/hyperled
ger/msp/users
        - ./config:/etc/hyperledger/configtx
    depends_on:
      - orderer.example.com
      - couchdbOrg1Peer1
    networks:
      - basic

  couchdbOrg1Peer1:
    container_name: couchdbOrg1Peer1
    image: hyperledger/fabric-couchdb
    environment:
      - COUCHDB_USER=peer1.Org1
      - COUCHDB_PASSWORD=password
    ports:
      - "6984:5984"
    networks:
      - basic

    cli:
    container_name: cli
    image: hyperledger/fabric-tools
    tty: true
```

```
    environment:
      - GOPATH=/opt/gopath
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_LOGGING_SPEC=info
      - CORE_PEER_ID=cli
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      - CORE_PEER_LOCALMSPID=Org1MSP
      -
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.exampl
e.com/msp
      - CORE_CHAINCODE_KEEPALIVE=10
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: /bin/bash
    volumes:
        - /var/run/:/host/var/run/
        - ./../chaincode/:/opt/gopath/src/github.com/
        -
./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/cry
pto/
        -
./config:/opt/gopath/src/github.com/hyperledger/fabric/peer/config/
    networks:
        - basic
    #depends_on:
    #  - orderer.example.com
    #  - peer0.org1.example.com
    #  - couchdb
```

Lab 4. Chaincode Management

Objectives

- Install Chaincode
- Instantiate Chaincode
- Configure Endorsement Policies

Installing Chaincode

The tools for performing chaincode operations are accessed by using the **`peer`** command in the network CLI Docker container.

First, we must decide which peer we would like to install our chaincode on. Let's choose p**`eer0.Org1`**. Next, we need to access the CLI Docker container.

Note: Please make sure the CLI container is running. If not, run:

```
docker-compose -f docker-compose.yml up -d cli
docker exec -it cli bash
```

and set the local environment variables with the appropriate pathing for the specified peer (In this case, **`peer0`**).

```
export CORE_PEER_LOCALMSPID=Org1MSP

export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

```
export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.exampl
e.com/msp
```

Remember the volumes we set in docker-compose?

Before we try to install anything, we must first see where our chaincode is, so that we can access it for installation. Below is the path to the sample chaincode directory:

```
ls /opt/gopath/src/github.com/sacc
```

Now we can finally begin the chaincode installation process for our first peer. The command to install is as follows:

```
peer chaincode install -n ccForAll -p github.com/sacc -v 1.0
```

Your Response: `Installed remotely response:<status:200 payload:"OK" >`

NOTE: It's important to note that this specific chaincode was written in Go, which is what Hyperledger Fabric natively expects. In the v1.4 release of Hyperledger Fabric, chaincode can also be written in Java or Node.js, so it is important to know the type of chaincode you are installing. If you are installing a separate language other than Golang, it is important to add the `--lang (-l)` flag to specify what kind of language you're using. Here is an example (don't run):

```
peer chaincode install -l java -n ccForAll -p ./SimpleChaincode.java
```

Here is a breakdown of each argument we specify on the `peer chaincode install` command:

`-n` is for the name of the chaincode we want to install on our peer
`-p` is the path to our chaincode file directory

Lastly, now we can check and see if our chaincode is installed on the container by running `peer chaincode list` and adding the flag `--installed`:

```
peer chaincode list --installed
```

Additionally, we must perform the same operation on the rest of the peers in the network.

TEST YOURSELF

There is one more peer left in Organization 1. Using the above walkthrough, try and install chaincode on **peer1.org1**.

Instantiate Chaincode

Now that chaincode is finally installed on all the peers in the Organization, chaincode instantiation is the next step in properly administering peer operations if a peer should be active on the network. We must remember that installing chaincode only means that the source code is being provided, but instantiating actually deploys the chaincode.

First, we must switch back to the first peer we would like to instantiate our chaincode on, which is **peer0**. Once again, we must return back to **peer0.Org1**'s environment variable pathing:

```
export CORE_PEER_LOCALMSPID=Org1MSP

export CORE_PEER_ADDRESS=peer0.org1.example.com:7051

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.exampl
e.com/msp
```

Now, this is the part where we typically run our **instantiate** command on **peer0**. Before we actually run it, let's analyze it. The command is listed below.

```
peer chaincode instantiate -n ccForAll  -v 1.0 -o \
orderer.example.com:7050 -C allarewelcome  \
-c '{"Args":["Mach","50"]}'
```

NOTE: IT MAY HANG FOR A MOMENT, BUT IT IS INVOKING THE CONSTRUCTOR FUNCTION, and initiating the chaincode lifecycle, so it will come back.

Let's walk through and see the total breakdown of what we are actually doing in this command:

-n name of our chaincode
-v version of the chaincode we want to install, more on this in later labs
-C the channel name (This is what we created in a previous lab)
-c The arguments we need to send through to initialize the constructor function of our chaincode. Please note that the constructor arguments depend on the chaincode init method, so it's imperative that the SysAdmin works closely with the Developer to understand its purpose. Otherwise, an alternative is reading the source code and figuring it out from there.

Now, before we send this command through, let's analyze and add one more flag to this command.

Practice with Endorsement Policies

Chaincode instantiation is synonymous with endorsement policy specification because they are both committed simultaneously. As we have discussed thus far in the course, endorsement policies are extremely important when specifying chaincode operations, because they determine who can execute the chaincode and who just has the chaincode installed for visibility, transparency, but not executability.

Now that we have our **`peer0`** configured, and our chaincode (source code) installed, we can specify an endorsement policy. We use the **`--policy`** flag to specify the endorsement policy for the specific chaincode we are instantiating.

```
--policy "AND('Org1.peer', 'Org2.peer' OR ('Org1.peer, Org2.peer,
Org3.peer))"
```

Now, let's break down this statement. We use the **`AND`** statement to indicate when the chaincode is run/invoked, these arguments must be approved of it (endorsing). The **`OR`** statement allows us to specify an alternate group of endorsers if the original requirements are not met. This is extremely similar to traditional conditional in Software Programming/Web Dev. Our complete command (ready to be entered) should be:

```
peer chaincode instantiate -n ccForAll -v 1.0 \
-o orderer.example.com:7050 -C allarewelcome \
-c '{"Args":["Mach","50"]}' \
--policy "AND('Org1.peer', OR ('Org1.member))"
```

Next, let's confirm that the chaincode is properly installed on the peer.

```
peer chaincode list --installed
```

Second, let's confirm the instantiation on our channel:

```
peer chaincode list --instantiated -C allarewelcome
```

Lab 5. Updating the Network (Anchor Peers)

Objective

- Update anchor peers.

Overview

Although the definition of an update for most means version changing, an update in terms of Hyperledger means an “edit” to a configuration on the network. Most of our “updates” to the network will be submitted as Configuration Transactions. In this lab, we will show you how to make updates to anchor peers.

Updating Anchor Peers Using Configtxgen

Now that we have two members in our organization, we can redefine our anchor peer for Org1, switching it from **`peer0`** to **`peer1`**.

Edit **`configtx.yaml`**, and change the anchor peer for Org1 (in the *Organizations* section) from **`peer0`** to **`peer1`**.

```
AnchorPeers:
      - Host: peer1.org1.example.com
        Port: 7051
```

There is a native flag on the **configtxgen** command to output an anchor peer transaction.

```
../bin/configtxgen -profile OneOrgChannel -outputAnchorPeersUpdate
./config/changeanchorpeerorg1.tx -channelID allarewelcome -asOrg
Org1MSP
```

Map it so we can access it!

Please open **docker-compose.yml** once more, and, under the *services.cli.volumes* section, please add:

```
- ./config:/opt/gopath/src/github.com/hyperledger/fabric/peer/confi
  g/
```

This allows us to access that configuration folder we just stored our organization definition in the CLI's working directory.

Next, we need to actually implement this updated transaction. We will use the CLI container.

```
docker container rm -f cli

docker-compose -f docker-compose.yml up -d cli


docker exec -it cli bash
```

Now let's run our command, referencing the anchor peer transaction we have created.

```
peer channel update -o orderer.example.com:7050 -c allarewelcome \
-f ./config/changeanchorpeerorg1.tx
```

Troubleshooting Note: If you encounter an error for the above command at all, you may need to reset yourself to the correct pathing for administrative privileges on org1. You can try:

```
export CORE_PEER_ADDRESS=peer1.org1.example.com:7051

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.exampl
e.com/msp
```

Now, let's check the logs output for **`peer1`**. We can see messages related to the anchor peer update by paying attention to the ones that start with the key **`learnAnchorPeers`**. We can check the peer logs and see this by exiting the container and running:

```
docker logs peer1.org1.example.com
```

Success!

Lab 6. Updating the Network (Chaincode)

Objective

- Update/Upgrade Chaincode.

Overview

Although the definition of an update for most means version changing, an update in terms of Hyperledger means an “edit” to a configuration on the network. Most of our “updates” to the network will be submitted as Configuration Transactions. In this lab we will show you how to update chaincode.

Updating/Upgrading Chaincode

We can imagine that our developer noticed an error in our **`ccForAll`** file and needs it to be updated, to prevent peer vulnerabilities.

```
export CORE_PEER_ADDRESS=peer0.org1.example.com:7051
```

Install the New (Upgraded) Chaincode on the Peer

```
peer chaincode install -n ccForAll -v 1.1 -p github.com/sacc
```

Now, we have the same chaincode installed on the peer, but multiple versions, which really helps if your company is dependent on version control as well. Now, let's put the new chaincode version in place of the old one.

Upgrade

```
peer chaincode upgrade -n ccForAll -v 1.1 -C allarewelcome \
-c '{"Args":["Mach","50"]}' \
-P "AND('Org1.peer', 'Org2.peer' OR ('Org1.peer, Org2.peer) )"
```

As you will instantly notice, the **`upgrade`** command is almost synonymous with the **`instantiate`** command, it's just suited for implementing new changes on a previously installed chaincode within a peer.

Lab 7. Multi-Organizational Management

Objective

- Scale our network from single organization to multi-organization.

Creating the Initial Information

This process of initial configuration will be very familiar because it is similar to how we added our second peer to Org1. We need to create a template for the **`crypto-certificate`** generation. Please navigate to and open the **`crypto-config.yaml`** file and append this to the bottom of the *PeerOrgs* section.

```
#
---------------------------------------------------------------------
---
 # Org2
---------------------------------------------------------------------
---
 - Name: Org2
   Domain: org2.example.com
   EnableNodeOUs: true
   Template:
     Count: 2
   Users:
     Count: 1
```

Now that we have defined the template for how our cryptographic certificates should be generated, let's add Org 2's peer container definition to our existing **`docker-compose.yml`** file. Open **`docker-compose.yml`** in your favorite code editor, and add the following values in the *services* section.

peer0.Org2.example.com	**peer1.Org2.example.com**
peer0.org2.example.com: **container_name: peer0.org2.example.com** **image: hyperledger/fabric-peer** **environment:** **-** **CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock** **- CORE_PEER_ID=peer0.org2.example.com** **- CORE_LOGGING_PEER=info** **- CORE_CHAINCODE_LOGGING_LEVEL=debug** **- CORE_PEER_LOCALMSPID=Org2MSP** **-** **CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/** **-** **CORE_PEER_ADDRESS=peer0.org2.example.com:7051** **-** **CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_basic** **- CORE_LEDGER_STATE_STATEDATABASE=CouchDB** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdbOrg2Peer0:5984** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=peer1.Org2** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=password** **working_dir:** **/opt/gopath/src/github.com/hyperledger/fabric** **command: peer node start** **ports:** **- 9051:7051** **- 9053:7053** **volumes:**	**peer1.org2.example.com:** **container_name: peer1.org2.example.com** **image: hyperledger/fabric-peer** **environment:** **-** **CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock** **- CORE_PEER_ID=peer1.org2.example.com** **- CORE_LOGGING_PEER=info** **- CORE_CHAINCODE_LOGGING_LEVEL=debug** **- CORE_PEER_LOCALMSPID=Org2MSP** **-** **CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/peer/** **- CORE_PEER_ADDRESS=peer1.org2.example.com:7051** **-** **CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_basic** **- CORE_LEDGER_STATE_STATEDATABASE=CouchDB** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdbOrg2Peer1:5984** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=peer1.Org2** **-** **CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=password** **working_dir:** **/opt/gopath/src/github.com/hyperledger/fabric** **command: peer node start** **ports:** **- 10051:7051** **- 10053:7053** **volumes:** **- /var/run/:/host/var/run/**

```
        - /var/run/:/host/var/run/
        -
./crypto-config/peerOrganizations/org2.example.com/
peers/peer0.org2.example.com/msp:/etc/hyperledger/m
sp/peer
        -
./crypto-config/peerOrganizations/org2.example.com/
users:/etc/hyperledger/msp/users
        - ./config:/etc/hyperledger/configtx
    depends_on:
      - orderer.example.com
      - couchdbOrg2Peer0
    networks:
      - basic
```

```
        -
./crypto-config/peerOrganizations/org2.example.com/pe
ers/peer1.org2.example.com/msp:/etc/hyperledger/msp/p
eer
        -
./crypto-config/peerOrganizations/org2.example.com/us
ers:/etc/hyperledger/msp/users
        - ./config:/etc/hyperledger/configtx
    depends_on:
      - orderer.example.com
      - couchdbOrg2Peer1
    networks:
      - basic
```

Test Yourself: Additionally, we must add CouchDB definitions. Please take this time to create two more couchDB containers for **`peer0.Org2`** and **`peer1.Org2`**. You can use the peer definitions we just defined (and Lab 2) for hints.

Now we need to add Org 2 to our configuration file before updating it throughout the network. Please open **`configtx.yaml`** in your favorite code editor and, under the *organizations* section, add:

```
- &Org2
    Name: Org2MSP
    ID: Org2MSP
    MSPDir: crypto-config/peerOrganizations/org2.example.com/msp
    AnchorPeers:
        - Host: peer1.org2.example.com
          Port: 7051
```

Generate Artifacts & Certificates

First, let's use **cryptogen** to generate certificates (without removing the other certificates already generated):

```
../bin/cryptogen extend --config=./crypto-config.yaml
```

Now, we should see two crypto-certificates folders when we run the command (from the root of the basic network folder):

```
ls crypto-config/peerOrganizations
```

Let's take our org definition and put it in a file, so we can later reference it.

```
../bin/configtxgen -printOrg Org2MSP > ./config/org2_definition.json
```

Before we move on
We need to make sure that we bring up a CA container for Org2, and each CA has the correct **`signcerts`** to start up and administer the identity properly.

In **`docker-compose.yml`** remove the CA definition you currently have, and paste:

`Org1ca.example.com:`	`Org2ca.example.com:`

`Org1ca.example.com:`
`image: hyperledger/fabric-ca`
`environment:`
`- FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server`
`- FABRIC_CA_SERVER_CA_NAME=Org1ca.example.com`
`- FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem`
`- FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/4239aa0dcd76daeeb8ba0cda701851d14504d31aad1b2ddddbac6a57365e497`
	`Org2ca.example.com:`
	`image: hyperledger/fabric-ca`
	`environment:`
	`- FABRIC_CA_HOME=/etc/hyperledger/fabric-ca-server`
	`- FABRIC_CA_SERVER_CA_NAME=Org2ca.example.com`
	`- FABRIC_CA_SERVER_CA_CERTFILE=/etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem`
	`- FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/4239aa0dcd76daeeb8ba0cda701851d14504d31aad1b2ddddbac6a57365e497`

c_sk ports: - "7054:7054" command: sh -c 'fabric-ca-server start -b admin:adminpw' volumes: - ./crypto-config/peerOrganizations /org1.example.com/ca/:/etc/hyperl edger/fabric-ca-server-config container_name: Org1ca.example.com networks: - basic	**c_sk ports: - "8054:7054" Command: sh -c 'fabric-ca-server start -b admin:adminpw' volumes: - ./crypto-config/peerOrganizations /org2.example.com/ca/:/etc/hyperl edger/fabric-ca-server-config container_name: Org2ca.example.com networks: - basic**

cd crypto-config/peerOrganizations/org1.example.com/ca

And copy the **sk** file (ex: **a29e53098e0b4dee6f6e8d7abc07e07a6074db69e822ff3adcc4415f033a1e75_sk**)

Now, open up **docker-compose.yml** again, and replace the **FABRIC_CA_SERVER_CA_KEYFILE** sk file in the path with the one you just copied.

Ex.
FABRIC_CA_SERVER_CA_KEYFILE=/etc/hyperledger/fabric-ca-server-config/4239aa0dcd76daeeb8ba0cda701851d14504d31aad1b2ddddbac6a57365e497c_sk

Next, navigate to the **sk** file for Org2 and do the same thing. It will live in the path:

./crypto-config/peerOrganizations/org2.example.com/ca

Start Hyperledger Fabric Containers

**docker-compose -f docker-compose.yml up -d Org1ca.example.com \
Org2ca.example.com**

(Alternative, if needed) If you took down all of the other containers, start everything all over again.

**docker-compose -f docker-compose.yml up **

```
-d Org1ca.example.com Org2ca.example.com orderer.example.com \
couchdbOrg1Peer0 peer0.org1.example.com couchdbOrg1Peer1 \
peer1.org1.example.com cli
```

Adding a New Organization to Our Channel

Next, we can run our **peer** commands from the **cli** container.

```
docker exec -it cli bash
```

This will feel very familiar because it involves a lot of what we learned in the previous labs. Again, we must start off by grabbing the latest configuration definition from the network.

```
peer channel fetch config blockFetchedConfig.pb -o \
orderer.example.com:7050 -c allarewelcome
```

Next, we will decode it.

```
configtxlator proto_decode --input blockFetchedConfig.pb \
--type common.Block | jq .data.data[0].payload.data.config > configBlock.json
```

We must modify the current configuration file that is on the new to include all our newest orgs:

```
jq -s '.[0] * {"channel_group":{"groups":{"Application":{"groups":
{"Org2MSP":.[1]}}}}}' \
configBlock.json ./config/org2_definition.json > configChanges.json
```

Let's confirm that this exists in our directory.

```
ls
```

Let's confirm the contents were inserted.

```
cat configChanges.json
```

Now that we've made the appropriate changes, we can encode our files. But first let's encode the original configuration file back into protobuf

```
configtxlator proto_encode --input configBlock.json \
--type common.Config --output configBlock.pb
```

Now we can perform the same encoding on our files with the modifications.

```
configtxlator proto_encode --input configChanges.json \
--type common.Config --output configChanges.pb
```

Since we have seen how to make changes to our configuration block manually, it is useful to show how we can make use of an option called **compute_update** on the **configtxlator. compute_update** allows us to compare our original (unmodified) configuration and our newly modified version to determine changes between the two, rather than doing it manually.

```
configtxlator compute_update --channel_id allarewelcome \
--original configBlock.pb --updated configChanges.pb \
--output configProposal_Org2.pb
```

Once more, we must take this updated file, and decode it so we can add all of the header-related information on it.

```
configtxlator proto_decode --input configProposal_Org2.pb \
--type common.ConfigUpdate | jq . > configProposal_Org2.json
```

We are basically taking our newly created configuration file, and attaching their original header information around it (also called fitting it in an envelope). We use **cat** to echo out the file contents, resulting as an insertion into the data key array. Then, we take the total contents of that and stick it into a new and final file.

```
echo
'{"payload":{"header":{"channel_header":{"channel_id":"allarewelcome",
"type":2}},"data":{"config_update":'$(cat
configProposal_Org2.json)'}}}' | jq . > org2SubmitReady.json
```

Next, we must re-encode it so we can submit it to the network for a configuration.

```
configtxlator proto_encode --input org2SubmitReady.json \
--type common.Envelope --output org2SubmitReady.pb
```

Now before we move on, let's recall that signature requirements are dependent on the type of configuration update being proposed. Since this is an organization being added, we must gather signatures from the majority of admins of Org1 (the org already on the channel). Before we send it off for response from network, we must first sign it ourselves (since we are currently Org1s' admin).

```
peer channel signconfigtx -f org2SubmitReady.pb
```

Now that we had Org1's admin signature on the configuration update file, we can send our changes (aka our update) off to the network for approval:

```
peer channel update -f org2SubmitReady.pb -c allarewelcome -o \
orderer.example.com:7050
```

Join New Peers to the Channel and Install Chaincode

Now we can bring up our Org2 containers.

```
docker-compose -f docker-compose.yml up \
-d couchdbOrg2Peer0 peer0.org2.example.com couchdbOrg2Peer1 \
peer1.org2.example.com
```

Now that we have added the Org to the network, we would need to add a peer and also establish them into our channel.

```
docker exec -it cli bash

export CORE_PEER_LOCALMSPID=Org2MSP

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.exampl
e.com/msp
```

To join our channel, we must fetch the configuration:

```
peer channel fetch config Org2AddedConfig.pb \
-o orderer.example.com:7050 -c allarewelcome
```

As we discussed before, in order to add a peer to a channel through the CLI, we must specify the genesis block at command execution (using the `--blockpath` or `-b` flag):

```
peer channel join -b allarewelcome.block
```

Next, in order for the peer to be valid and in sync with all of our operations, we must install the chaincode on Peer 0 of Org2. Note that once we instantiate, we will be using a different endorsement policy.

```
peer chaincode install -n ccForAll -v 1.1 -n networkChaincode \
-p github.com/sacc.go
```

Try it Yourself: Do the same thing, and join `Peer1.Org2` to the channel, and install/instantiate the chaincode on it.

Upgrading the Endorsement Policy of Other Peers

Because we want to include a peer from Org2 as an endorser, we need to update the peer chaincode endorsement policy. To accomplish this, we will raise the version of our chaincode and run the modified endorsement policy on all related peers in the channel. If we run the command:

```
peer chaincode list --installed
```

It should return the following output:

```
Name:ccForAll, Version: 1.0, Path: github.com/sacc, Id:
79d2ed968d1494ba1d8ec299454dcc3eac5e23d15effaf41f6b3a111f592c1b9
```

As we can see, our chaincode is versioned at 1.0. Let's upgrade it with our new chaincode endorsement policy. We can version it however we like (v1.1 or v2.0), depending on if the release is major or minor. Since this is a minor change, let's update it to 1.1:

```
peer chaincode install -n mycc -v 1.1 -n networkChaincode \
-p github.com/sacc
```

Please run the above command on all of the peers for org1

Now we can upgrade (re-instantiate) the chaincode onto the channel.

```
peer chaincode upgrade -n ccForAll -v 1.1 \
-o orderer.example.com:7050  \
--policy "AND('Org1.peer', 'Org2.peer' OR ('Org1.admin') )" \
-c '{"Args":["Mach", "50"]}'
```

```
peer chaincode list --installed && peer chaincode list \
--instantiated -C allarewelcome
```

SUCCESS!

Lab 8. Advanced Channel Management

Objectives

- Create a channel.
- Change channel configuration.
- Add Orgs to the channel.

Create the Transaction Artifacts

Before we actually run commands, we must create channel definitions and use the **configtxgen** tool to create their network.

To practice, we are going to create three channels. These channels will be:

1.) All Org1 peers & two Org 2 peers
2.) All Org2 peers Only
3.) Org 1 Peers only

Open the **`configtx.yaml`** file, and append this to the profiles section:

```
OrgOneChannel:
    Consortium: SampleConsortium
    Application:
        <<: *ApplicationDefaults
```

```
            Organizations:
                - *Org1
    OrgTwoChannel:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org2
    AllAreWelcomeTwo:
        Consortium: SampleConsortium
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org1
                - *Org2
```

Next, go to the command line and, from the root of your project folder, run:

```
../bin/configtxgen -profile Org1Channel \
-outputCreateChannelTx ./config/OrgOneChannel.tx \
-channelID orgonechannel

../bin/configtxgen -profile Org2Channel \
-outputCreateChannelTx ./config/OrgTwoChannel.tx \
-channelID orgtwochannel

../bin/configtxgen -profile AllAreWelcomeTwo \
-outputCreateChannelTx ./config/AllAreWelcomeTwo.tx \
-channelID allarewelcometwo
```

Let’s check out where this lives.

```
cd config && ls
```

You should now see all three channel transaction artifacts. Next, we need to actually create the channels.

Create a New Channel from CLI

To review, we know that channels are created to help manage privacy and discretion within business networks. This allows deeper interpersonal/interorganizational interaction for companies permitted.

In order to create a new channel after the business network has been deployed, we need to use the **`peer channel`** command.

Let's analyze what we currently have:

- Two Organizations
- Four total peers

Before we start with channel operations, let's set the appropriate certification pathing for our orderer since we will be using it throughout most of our operations.

```
docker exec -it cli bash
```

```
export
CORE_PEER_MSPCONFIGPATH=/etc/hyperledger/msp/users/Admin@org1.example.
com/msp
```

Now that the orderer is properly configured, we can create a channel, and we can use the **`peer channel create`** command to create a new channel:

```
peer channel create -o orderer.example.com:7050 -f \
./config/AllAreWelcomeTwo.tx -c allarewelcometwo
```

For the second channel,

```
peer channel create -o orderer.example.com:7050 -f \
./config/OrgOneChannel.tx -c orgonechannel
```

For the third channel:

```
peer channel create -o orderer.example.com:7050 -f \
./config/OrgTwoChannel.tx -c orgtwochannel
```

You should get the error:

"Error: got unexpected status: BAD_REQUEST -- Attempted to include a member which is not in the consortium"

This is because you are currently doing this from Org1, and not Org2.

Let's switch to Org2 Administrator's certificates, and rerun this command:

```
export CORE_PEER_LOCALMSPID="Org2MSP"

export
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.exampl
e.com/msp

export CORE_PEER_ADDRESS=peer0.org2.example.com:7051
```

IMPORTANT NOTE*: Although the channel doesn't give an output that explicitly indicates its' successful creation, if you see that the initial genesis block was received (AKA Block 0), this means that the operation was successful.*

Add the Correct Peers to Appropriate Channels

Now that we have created the channels, and received all the blocks needed to join our channels, we still need to add peers to them. Since we have our pathing set to Org2, let's start with the channel **`Org2Only`**.

```
peer channel join -o orderer.example.com:7050 -b ./org2channel.block
```

Try it out: Now, switch back to the Org1 administrator and add both **`peer0`**, and **`peer1`** to the channel. Remember to adjust the **`core_peer_address`** appropriately.

Lab 9. Kafka

Objective

- Configure and bring up a Kafka-based ordering service.

Note: We will be structuring a very simple simple 2Kafka:3Zookeeper ratio in our cluster.

Note: In order to implement changes to the ordering service, we must reset the network.

Configtx File

First, we must make configuration changes to the *Orderer* section of our **`configtx.yaml`** file. First, we will change the value of the key **`OrdererType`** to be **`kafka`**.

```
OrdererType: kafka
```

Next, we need to add and specify the addresses of our Kafka broker containers that will be running in the network. We plan to just bring up two Kafka brokers since our network isn't that large. So, under **`brokers`**, we can delete the

```
- 127.0.0.1:9092
```

And enter

```
- kafkaA.example.com:9092
```

```
- kafkaB.example.com:9092
```

This means that we can find the kafka containers running at their respective container domain names, on port 9092.

Docker-compose

As with all of our network components/nodes, Kafka must be stood up in separate containers as well. We can do this in our **`docker-compose.yml`** file. We will have two kafka processes running, so we need to define each one with unique names (**`kafka1`** and **`kafka2`**). It's important to also note that we will be using Zookeeper to manage these brokers, so we must assign dependency, and connection to each server. Enter these two container definitions:

`kafkaA.example.com`

```
 kafka0.example.com:
   container_name: kafka0.example.com
   image: hyperledger/fabric-kafka
   restart: always
   environment:
     - KAFKA_MESSAGE_MAX_BYTES=103809024
     - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
     - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
     - KAFKA_MIN_INSYNC_REPLICAS=2
     - KAFKA_DEFAULT_REPLICATION_FACTOR=2
     -
KAFKA_ZOOKEEPER_CONNECT=zookeeper0.example.com:2181,
zookeeper1.example.com:2181,zookeeper2.example.com:2
181
     - KAFKA_BROKER_ID=0
   ports:
     - 9092:9092
     - 9093:9093
   networks:
     - basic
   depends_on:
     - zookeeper0.example.com
     - zookeeper1.example.com
     - zookeeper2.example.com
```

`kafkaB.example.com`

```
 kafka1.example.com:
   container_name: kafka1.example.com
   image: hyperledger/fabric-kafka
   restart: always
   environment:
     - KAFKA_MESSAGE_MAX_BYTES=103809024
     - KAFKA_REPLICA_FETCH_MAX_BYTES=103809024
     - KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE=false
     - KAFKA_MIN_INSYNC_REPLICAS=2
     - KAFKA_DEFAULT_REPLICATION_FACTOR=2
     -
KAFKA_ZOOKEEPER_CONNECT=zookeeper0.example.com:2181,
zookeeper1.example.com:2181,zookeeper2.example.com:2
181
     - KAFKA_BROKER_ID=1
   ports:
     - 10092:9092
     - 10093:9093
   networks:
     - basic
   depends_on:
     - zookeeper0.example.com
     - zookeeper1.example.com
     - zookeeper2.example.com
```


Standing Up Zookeeper

There must be an odd number of Zookeeper servers in order to implement Crash Fault Tolerance and keep the network running. Since we have two Kafka brokers, we can create three zookeeper servers, assigning custom ID's to each one. (Please note that the **ZOO_SERVERS** environment variable and its attached values should be all on the same line)

ZookeeperA.example.com	**ZookeeperB.example.com**	**ZookeeperC.example.com**
zookeeper0.example.com: container_name: zookeeper0.example.com image: hyperledger/fabric-zookeeper environment: - ZOO_MY_ID=1 - ZOO_SERVERS=server.1=zookeeper0.example.com:2888:3888 server.2=zookeeper1.example.com:2888:3888 server.3=zookeeper2.example.com:2888:3888 ports: - 2181:2181 - 2888:2888 - 3888:3888 networks: - basic	zookeeper1.example.com: container_name: zookeeper1.example.com image: hyperledger/fabric-zookeeper environment: - ZOO_MY_ID=2 - ZOO_SERVERS=server.1=zookeeper0.example.com:2888:3888 server.2=zookeeper1.example.com:2888:3888 server.3=zookeeper2.example.com:2888:3888 ports: - 12181:2181 - 12888:2888 - 13888:3888 networks: - basic	zookeeper2.example.com: container_name: zookeeper2.example.com image: hyperledger/fabric-zookeeper environment: - ZOO_MY_ID=3 - ZOO_SERVERS=server.1=zookeeper0.example.com:2888:3888 server.2=zookeeper1.example.com:2888:3888 server.3=zookeeper2.example.com:2888:3888 ports: - 22181:2181 - 22888:2888 - 23888:3888 networks: - basic

Start Your Engines!

Now that we have all of our configuration details handled, we can bring up our zookeeper and kafka services. Remember we need to start them in that order because our kafka brokers rely on Zookeeper management. Run this command to start:

```
docker-compose -f docker-compose.yml up \
-d zookeeper1.example.com zookeeper2.example.com \
zookeeper3.example.com kafkaA.example.com kafkaB.example.com
```

To allow everything to start up in a correct amount of time we will wait about 30 seconds, and then we can start up our normal orderer and peer nodes:

```
docker-compose -f docker-compose.yml up \
-d ca.example.com orderer.example.com peer0.example.com \
peer1.org1.example.com peer0.org2.example.com cli
```

Test Yourself: Network Channels Gone?!

Since we had to restart the network as a whole in order to implement the new Kafka-based ordering service, we will not have any channels implemented anymore. We must now reimplement our channels. Using what you learned in the previous steps, take the time now to recreate the channels we had earlier and join the appropriate organizations. Feel free to use the prior labs as reference, if you need assistance.

SUCCESS!

Lab10. CA Operations

Objectives

- Initialize and start the Fabric server.
- Enroll an Intermediate CA.
- Enroll an Identity.
- Register a new Identity.
- Remove an Identity.
- Revoke an Identity and the Certificate Revocation List (CRL).
- Get Identity information.

Note: Before finding and starting the Fabric server, please find this line in the *ca service* section of the **`docker-compose.yml`**

```
command: sh -c 'fabric-ca-server start -b admin:adminpw'
```

and replace it with the second command:

```
command: sh -c 'sleep 100000'
```

Going into the CA Container

First, to access the container, we must list all of the Docker containers so that we can find the appropriate container to execute. Run the following command:

```
docker ps
```

Next, after looking through the list of Docker containers, we will find a Docker container titled **`ca.org1.example.com`**. We can either use this name or the container's ID to **`exec`** into it. Run the following command:

```
docker exec -it ca.example.com bash
```

Now that we are in the container, we have access to the **`fabric-ca-client`** commands. We can confirm this by running:

```
fabric-ca-server --help
```

If you see a list of commands, you are all set and can move on to the next section.

Note: If you don't, please take down and restart the ca docker container. Also make sure you have the correct version of this CA image from Docker Hub.

Initializing the Server

In order to use the Fabric server, we must configure it and then start it up.
In order to generate the configuration file, we will be using the **`fabric-ca-server init`** command.

```
The CA key and certificate files already exist
Key file location: /etc/hyperledger/fabric-ca-server-config/329dac791fb648e0121c8fc7e787287c3085a5f7200750a47a7a468c2a11f32d_sk
Certificate file location: /etc/hyperledger/fabric-ca-server-config/ca.org1.example.com-cert.pem
Initialized sqlite3 database at /etc/hyperledger/fabric-ca-server/fabric-ca-server.db
The Idemix issuer public and secret key files already exist
   secret key file location: /etc/hyperledger/fabric-ca-server/msp/keystore/IssuerSecretKey
   public key file location: /etc/hyperledger/fabric-ca-server/IssuerPublicKey
The Idemix issuer revocation public and secret key files already exist
   private key file location: /etc/hyperledger/fabric-ca-server/msp/keystore/IssuerRevocationPrivateKey
   public key file location: /etc/hyperledger/fabric-ca-server/IssuerRevocationPublicKey
Home directory for default CA: /etc/hyperledger/fabric-ca-server
Listening on http://0.0.0.0:8080
```

Note: If you get any errors when initializing our server from the start, remove any previously initialized files artifacts by doing:

```
cd /etc/hyperledger/fabric-ca-server
rm ca-cert.pem ca-key.pem fabric-ca-server-config.yaml
```

Now, let's initialize our server:

```
fabric-ca-server init -b caServerAdmin:AdminsRock
```

Let's take a moment to look at the contents of our **`ca-server-config.yaml`** file.

```
cat fabric-ca-server-config.yaml
```

The contents in this file define how our CA server is structured and operates. For example, the *registry* section of this file contains the values we passed in for name and password.

```
# Contains identity information which is used when LDAP is disabled
identities:
   - name: CaServerAdmin
     pass: AdminsRock
     type: client
```

Now that our **`fabric-server`** is initialized and configured, we can start it by running:

```
fabric-ca-server start -b caServerAdmin:AdminsRock
```

If you get a port conflict, you can use the **`-p`** flag to specify a different port for startup.

```
fabric-ca-server start -b caServerAdmin:AdminsRock -p 8080
```

Open a new terminal window and enter back into the CA container. Now that we've started up the server, we need to set our server identity.

Starting an Intermediate CA Server

We can use the Root CA server we just started up to enroll an Intermediate CA server. We just need to specify the credentials for the Intermediate CA server and the port to run it on, as well as our Root CA server which is the one we started up previously. Open a new terminal window and enter:

```
fabric-ca-server start -b intermediateCAServer:ServerPassword \
-u http://caServerAdmin:AdminsRock@localhost:8080 -p 3000
```

Enrolling an Identity

Since the CA is in charge of all peers memberships, enrollment in the network will be a vital/essential daily tool. Let's look at how we can do an enrollment:

```
fabric-ca-client enroll -u <serverPath> <options>
```

Enrolling Ourselves as the Admin Authority

For simplicity purposes, we can use the **`enroll`** command to not only enroll peers, but also ourselves as a client administrator to perform identity operations. We do need to specify the path to our CA server, and login using our credentials, which we can specify in the beginning of the URL.

```
fabric-ca-client enroll \
-u http://caServerAdmin:AdminsRock@localhost:8080
```

Enrolling a Peer

Note: We want to give this admin privileges to this identity. These privileges will be the abilities to revoke organization-level certificates, generate CRL's, and register other peers. You can see these specifications under the **`id.attrs`** statement.

```
fabric-ca-client register \
--id.name Org1Administrator \
--id.affiliation org1 --id.type admin \
--id.attrs 'admin=true:ecert, hf.Revoker=true, hf.GenCRL=true,
hf.Registrar.Roles=peer' \
--id.secret Org1Rocks -u http://localhost:8080
```

Now we can enroll the admin:

```
fabric-ca-client enroll \
-u http://Org1Administrator:Org1Rocks@localhost:8080
```

Next, let's create two regular peer identities for Org1, and then enroll them:

```
fabric-ca-client register --id.name peerJohn \
--id.affiliation org1 --id.type peer \
--id.secret 'IAMAPEER!' -u http://localhost:8080

fabric-ca-client enroll -u http://peerJohn:'IAMAPEER!'@localhost:8080

fabric-ca-client register --id.name peerSam \
--id.affiliation org1 --id.type user \
--id.secret 'IAMAPEER2!' \
```

```
-u http://Org2Administrator:Org2Rocks@localhost:8080

fabric-ca-client enroll -u http://peerJohn:'IAMAPEER2!'@localhost:8080
```

As you can see, when we are registering/enrolling our node onto the network, at a minimum, we need to account for four important things (in this particular order):

1. Name
2. Organization the node belongs to
3. Their role-based capabilities
4. The node type.

Now that we have three new identities created, we can confirm that everything is correct and they exist by running the command:

```
fabric-ca-client enroll -u
http://caServerAdmin:AdminsRock@localhost:8080
```

Getting Identity Information

Now we can check to see if your identities have been created by running:

```
fabric-ca-client identity list *
```

If we want to specify our search for a specific user (like **`peerSam`**), we can run:

```
fabric-ca-client identity list --id peerSam
```

Revoking

Now that we have enrolled our peer onto the network, let's imagine that we made a major mistake by allowing this node in. We must reject its' certificate and let the network know this node is never allowed because of its' malicious activity. Let's use the **`revoke`** command:

```
fabric-ca-client revoke -e peerJohn -r 'keycompromise'
```

The **`-r`** flag allows us to specify a reason (from a list of available reasons) why we decided to revoke the certificate of the node. In this case, we will say that we revoked this node because of a key-related exposure issue. Now that we have a node properly revoked, we can move on.

Certificate Revocation List

As mentioned in this chapter, in order to keep ledgers secure, and prevent malicious activity on the network, we must be able to keep track of all the actors who have had their identities revoked. Now that we've revoked **peerSam**, we need to generate a CRL (Certificate Revocation List), to see it. There is a native command for achieving this called **gencrl**.

```
fabric-ca-client gencrl

cd /etc/hyperledger/fabric-ca-server/msp/crls/
```

Since it is a PEM-encoded file, in order to see it, we need to use the **openSSL** command to decrypt it:

```
(Optional) apt install openssl
openssl crl -inform PEM -text -in crl.pem
```

As we can see from the output, we have our revoked certificate in our CRL, and the date/time it was revoked.

```
Revoked Certificates:
    Serial Number: 1F454E2A29831C2438C66307E3A4ECCD235B4A03
        Revocation Date: Dec  3 07:11:43 2018 GMT
```

List Revoked Certs by Command

Let's list our revoked x509 certificates:

```
fabric-ca-client certificate list --revocation 2018-01-01::2019-12-30
```

You may be wondering what those dates attached to the **--revocation** flag are. This is the parameter for pulling certificates revoked within a certain timeframe, using the double colon (**::**) as a separator. Here is the complete syntax structure: **<OldestTime>::<LatestTime>**

Removing an Identity

Let’s assume that the peer by the name of John left the company to work for a different one outside of our network. Since he is not needed at all, as protocol, we can save space and remove all of his information from the network by running:

```
fabric-ca-client identity remove peerJohn \
--cfg.identities.allowremove -b caServerAdmin:AdminsRock
```

SUCCESS!

Lab 11. Transport Layer Security

Overview

Transport Layer Security is naturally turned off by default, which is okay for local host testing. But in a production environment, or even a pre-deployment multi-host setup, it is imperative that TLS be set up and turned on in order to secure all communication between nodes in the network. Both peers and orderers have options/settings for TLS security.

Setting up TLS

If we configure our network nodes to require TLS certificates, then all of our certificates for all nodes will live in the **`crypto-config`** folder. Each node has a specific folder titled “tls” in its subdirectory, which houses all of the x509 certificates generated for it.

In your network root directory, please navigate to:

`/crypto-config/peerOrganizations/<orgDomainName>/peers/<PEERNAME>/tls`

Example:

`/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls`

`crypto-config/ordererOrganizations/example.com/msp/tlscacerts`

Configuring TLS on a Peer Node

We can configure the environment variables in our **`docker-compose.yml`** file to use TLS authentication. We will start by adding these validating environment variables to all of our peers in the network. There are four things we need to account for:

- If TLS is enabled
- The TLS certificate file for the peer
- The Key file for the peer
- The Root certificate file

Before we enable TLS, let's do a quick check to see if we can return our channels.

```
peer channel list
```

Now, let's go into the **`docker-compose`** file and under each peer's definition, let's add the following environment variable to override the default TLS off setting (thus, turning on TLS):

```
CORE_PEER_TLS_ENABLED=true
```

TLS Identity Pathing

As seen in the previous command, we are once again overriding the **`core.yaml`** file initial value, by simply turning on the requirement for TLS operations, so that all communications are secured. Under that, we can add to the list. For this specific example, we will want to have administrative privileges, so we must use all of our admin certificates which will be found in the **`tls`** subdomain of the **`admin`** folder. Here is the path:

`/crypto-config/peerOrganizations/`[**`org1.example.com/users/Admin@org1.example.com/tls`**]

Now that we have our **`crypto-certs`** folder mapped, we must find it in the following path:

```
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/admin@org1.example.com/tls
```

These next three environment variables will come from the TLS subfolder of our Admin for Org1.

```
export
CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
```

```
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.exampl
e.com/tls/client.crt

export
CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/p
eer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example
.com/tls/client.key

export
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fab
ric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.ex
ample.com/tls/ca.crt
```

Now that we have all of our pathing to the proper TLS certificates corrected and implemented through our environment variables, we can, and will specify them at runtime. Additionally, we must specify the endpoint for the orderer we decide to use. Please run the following command from the CLI container:

```
peer channel list -o orderer.example.com:7050 --tls \
--cafile $CORE_PEER_TLS_ROOTCERT_FILE \
--certfile $CORE_PEER_TLS_CERT_FILE \
--keyfile $CORE_PEER_TLS_KEY_FILE
```

Invoke Chaincode Operations with TLS

To drive the concept home, we can try this out on a second operation. We will invoke (in other words, execute) our chaincode, using TLS to invoke the chaincode.

```
peer chaincode invoke {"Args":["Mach","50"]} \
--tls -o orderer.example.com:7050 \
--tls --cafile $CORE_PEER_TLS_ROOTCERT_FILE \
--certfile $CORE_PEER_TLS_CERT_FILE \
--keyfile $CORE_PEER_TLS_KEY_FILE
```

Lab 12. Service Discovery

Note: If you encounter yourself working on a legacy project using v1.2 or older, you must first update the network version from 1.2 to 1.4 because the **discover** binary was not available to use through the CLI container until v1.3+. You can grab the updated cli (as well as all the other ones containers) & updated **bin** folder for your update from inside this link (which we used in the beginning of the class).

```
curl -sSL http://bit.ly/2ysbOFE | bash -s 1.4.0
```

Make sure to replace the old projects `bin` folder with the one from the above link.

ConfigFile Creation

Let's use a command on the Discover CLI to create a configuration file that allows us to discover our peers dynamically.

`userKey` Flag

We need to grab the proper certification/key auth for the peers we would like to discover. Navigate to and grab the SK file name by running this:

```
cd
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganiza
tions/org1.example.com/users/User1@org1.example.com/msp/keystore
```

Here is the part where we add the SK file to our flag:

```
--userKey msp/keystore/<InsertYourCustomSKFILEhere>
```

So, your output result should look similar to:

EXAMPLE:

```
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganiza
tions/org1.example.com/users/User1@org1.example.com/msp/keystore/c75bd
6911aca808941c3557ee7c97e90f3952e379497dc55eb903f31b50abc83_sk
```

`UserCert` Flag

Now, we must specify the `--userCert` flag. Let's head to the proper folder so we can grab the `signcert` path for this flag:

```
cd
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganiza
tions/org1.example.com/users/User1@org1.example.com/msp/signcerts/
```

This is what the total command will look like:

```
discover --configFile discoveryConfig.yaml --userKey
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganiza
tions/org1.example.com/users/User1@org1.example.com/msp/keystore/c75bd
6911aca808941c3557ee7c97e90f3952e379497dc55eb903f31b50abc83_sk
--userCert
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganiza
tions/org1.example.com/users/User1@org1.example.com/msp/signcerts/User
1@org1.example.com-cert.pem --MSP Org1MSP saveConfig
```

Restructure and Declutter

Since this is extremely displeasing to the visual eye, and we have adequate experience with environment variables, let's restructure this command with some environment variables. We will create `$USERKEYFILE` and `$USERCERTFILE` variables to store our paths.

```
export
USERKEYFILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/ke
ystore/c75bd6911aca808941c3557ee7c97e90f3952e379497dc55eb903f31b50abc8
3_sk
```

```
export
USERCERTFILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
/peerOrganizations/org1.example.com/users/User1@org1.example.com/msp/s
igncerts/User1@org1.example.com-cert.pem
```

What It Looks like Now

Now, our command has been cut from numerous lines to a little over one line and looks like:

```
discover --configFile discoveryConfig.yaml \
--userKey $USERKEYFILE \
--userCert $USERCERTFILE \
--MSP Org1MSP saveConfig
```

Perform the **`ls`** command to see your newly created **`configFile`** in the directory, and let's output its contents.

```
ls && cat discoveryConfig.yaml
```

Now, in order to allow our peers to be discovered by our service, they must have their endpoints exposed to the outside for interaction. Let's set the environment variables for its' external endpoint. In your **`docker-compose`** file, please add the following environment variable under the peer service definitions.

```
CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.org1.example.com:7051
```

Put the Configuration File to Work

Now, using our newly generated configuration file, we can inquire dynamically generated information about the peers in our channel using the command:

```
discover peers --configFile conf.yaml  \
--channel allarewelcome --server peer0.org1.example.com:7051
```

Grab Endorser-Related Information

The process is similar to the peer request, except we must specify the chaincode we want to grab our Endorsement Policy information on. Let's grab the endorser information based on our previously-created Chaincode named **`ccForAll`**.

```
discover peers --configFile conf.yaml \
--channel allarewelcome \
--server peer0.org1.example.com:7051 --chaincode ccForAll


Success!
```