**Adding an Org to a Channel**

**Note**

Ensure that you have downloaded the appropriate images and binaries as outlined in [Install Samples, Binaries and Docker Images](https://hyperledger-fabric.readthedocs.io/en/release-1.4/install.html) and [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html) that conform to the version of this documentation (which can be found at the bottom of the table of contents to the left). In particular, your version of the fabric-samples folder must include the eyfn.sh (“Extending Your First Network”) script and its related scripts.

This tutorial serves as an extension to the [Building Your First Network](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html) (BYFN) tutorial, and will demonstrate the addition of a new organization – Org3 – to the application channel (mychannel) autogenerated by BYFN. It assumes a strong understanding of BYFN, including the usage and functionality of the aforementioned utilities.

While we will focus solely on the integration of a new organization here, the same approach can be adopted when performing other channel configuration updates (updating modification policies or altering batch size, for example). To learn more about the process and possibilities of channel config updates in general, check out [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html)). It’s also worth noting that channel configuration updates like the one demonstrated here will usually be the responsibility of an organization admin (rather than a chaincode or application developer).

**Note**

Make sure the automated byfn.sh script runs without error on your machine before continuing. If you have exported your binaries and the related tools (cryptogen, configtxgen, etc) into your PATH variable, you’ll be able to modify the commands accordingly without passing the fully qualified path.

**Setup the Environment**

We will be operating from the root of the first-network subdirectory within your local clone of fabric-samples. Change into that directory now. You will also want to [open](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) a few extra terminals for ease of use.

First, use the byfn.sh script to tidy up. This command will kill any active or stale docker containers and remove previously generated artifacts. It is by no means **necessary** to bring down a Fabric network in order to perform channel configuration update tasks. However, for the sake of this tutorial, we want to operate from a known initial state. Therefore let’s run the following command to clean up any previous environments:

**./**byfn**.**sh down

Now generate the default BYFN artifacts:

**./**byfn**.**sh generate

And launch the network making use of the scripted execution within the CLI container:

**./**byfn**.**sh up

Now that you have a clean version of BYFN running on your machine, you have two different paths you can pursue. First, we offer a fully commented script that will carry out a config transaction update to bring Org3 into the network.

Also, we will show a “manual” version of the same process, showing each step and explaining what it accomplishes (since we show you how to bring down your network before this manual process, you could also run the script and then look at each step).

**Bring Org3 into the Channel with the Script**

You should be in first-network. To use the script, simply issue the following:

**./**eyfn**.**sh up

The output here is well worth reading. You’ll see the Org3 crypto material being added, the config update being created and signed, and then chaincode being installed to allow Org3 to execute ledger queries.

If everything goes well, you’ll get this message:

**=========** All GOOD, EYFN test execution completed **===========**

eyfn.sh can be used with the same Node.js chaincode and database options as byfn.sh by issuing the following (instead of ./byfn.sh up):

**./**byfn**.**sh up **-**c testchannel **-**s couchdb **-**l node

And then:

**./**eyfn**.**sh up **-**c testchannel **-**s couchdb **-**l node

For those who want to take a closer look at this process, the rest of the [doc](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) will show you each command for making a channel update and what it does.

**Bring Org3 into the Channel Manually**

**Note**

The manual steps outlined below assume that the FABRIC\_LOGGING\_SPEC in the cli and Org3cli containers is set to DEBUG.

For the cli container, you can set this by modifying the docker-compose-cli.yaml file in the first-network directory. e.g.

cli:

container\_name: cli

image: hyperledger/fabric-tools:$IMAGE\_TAG

tty: true

stdin\_open: true

environment:

- GOPATH=/opt/gopath

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

#- FABRIC\_LOGGING\_SPEC=INFO

- FABRIC\_LOGGING\_SPEC=DEBUG

For the Org3cli container, you can set this by modifying the docker-compose-org3.yaml file in the first-network directory. e.g.

Org3cli:

container\_name: Org3cli

image: hyperledger/fabric-tools:$IMAGE\_TAG

tty: true

stdin\_open: true

environment:

- GOPATH=/opt/gopath

- CORE\_VM\_ENDPOINT=unix:///host/var/run/docker.sock

#- FABRIC\_LOGGING\_SPEC=INFO

- FABRIC\_LOGGING\_SPEC=DEBUG

If you’ve used the eyfn.sh script, you’ll need to bring your network down. This can be done by issuing:

**./**eyfn**.**sh down

This will bring down the network, delete all the containers and undo what we’ve done to add Org3.

When the network is down, bring it [back](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) up again.

**./**byfn**.**sh generate

Then:

**./**byfn**.**sh up

This will bring your network back to the same state it was in before you executed the eyfn.sh script.

Now we’re ready to add Org3 manually. As a first step, we’ll need to generate Org3’s crypto material.

**Generate the Org3 Crypto Material**

In another terminal, change into the org3-artifacts subdirectory from first-network.

cd org3**-**artifacts

There are two yaml files of interest here: org3-crypto.yaml and configtx.yaml. First, generate the crypto material for Org3:

**../../**bin**/**cryptogen generate **--**config**=./**org3**-**crypto**.**yaml

This command reads in our new crypto yaml file – org3-crypto.yaml – and leverages cryptogen to generate the keys and certificates for an Org3 CA as well as two peers bound to this new Org. As with the BYFN implementation, this crypto material is put into a newly generated crypto-config folder within the present working directory (in our case, org3-artifacts).

Now use the configtxgen utility to print out the Org3-specific configuration material in JSON. We will preface the command by telling the tool to look in the current directory for the configtx.yaml file that it needs to ingest.

export FABRIC\_CFG\_PATH=$PWD && ../../bin/configtxgen -printOrg Org3MSP > ../channel-artifacts/org3.json

The above command creates a JSON file – org3.json – and outputs it into the channel-artifacts subdirectory at the root of first-network. This file contains the policy definitions for Org3, as well as three important certificates presented in base 64 format: the admin [user](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) certificate (which will be needed to act as the admin of Org3 later on), a CA root cert, and a TLS root cert. In an upcoming step we will append this JSON file to the channel configuration.

Our final piece of housekeeping is to port the Orderer Org’s MSP material into the Org3 crypto-config directory. In particular, we are concerned with the Orderer’s TLS root cert, which will allow for secure communication between Org3 entities and the network’s ordering node.

cd **../** **&&** cp **-**r crypto**-**config**/**ordererOrganizations org3**-**artifacts**/**crypto**-**config**/**

Now we’re ready to update the channel configuration…

**Prepare the CLI Environment**

The update process makes use of the configuration translator tool – configtxlator. This tool provides a stateless REST API independent of the SDK. Additionally it provides a CLI, to simplify configuration tasks in Fabric networks. The tool allows for the easy conversion between different equivalent data representations/formats (in this case, between protobufs and JSON). Additionally, the tool can compute a configuration update transaction based on the differences between two channel configurations.

First, exec into the CLI container. Recall that this container has been mounted with the BYFN crypto-config library, giving us access to the MSP material for the two original peer organizations and the Orderer Org. The bootstrapped identity is the Org1 admin user, meaning that any steps where we want to act as Org2 will require the export of MSP-specific environment variables.

docker exec **-**it cli bash

Export the ORDERER\_CA and CHANNEL\_NAME variables:

export ORDERER\_CA**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**ordererOrganizations**/**example**.**com**/**orderers**/**orderer**.**example**.**com**/**msp**/**tlscacerts**/**tlsca**.**example**.**com**-**cert**.**pem **&&** export CHANNEL\_NAME**=**mychannel

Check to make sure the variables have been properly set:

echo $ORDERER\_CA && echo $CHANNEL\_NAME

**Note**

If for any reason you need to restart the CLI container, you will also need to re-export the two environment variables – ORDERER\_CA and CHANNEL\_NAME.

**Fetch the Configuration**

Now we have a CLI container with our two key environment variables – ORDERER\_CA and CHANNEL\_NAME exported. Let’s go fetch the most recent config block for the channel – mychannel.

The reason why we have to pull the latest version of the config is because channel config elements are versioned. Versioning is important for several reasons. It prevents config changes from being repeated or replayed (for instance, reverting to a channel config with old CRLs would represent a security risk). Also it helps ensure concurrency (if you want to remove an Org from your channel, for example, after a new Org has been added, versioning will help prevent you from removing both Orgs, instead of just the Org you want to remove).

peer channel fetch config config\_block.pb -o orderer.example.com:7050 -c $CHANNEL\_NAME --tls --cafile $ORDERER\_CA

This command saves the binary protobuf channel configuration block to config\_block.pb. Note that the choice of name and file extension is arbitrary. However, following a convention which identifies both the type of object being represented and its encoding (protobuf or JSON) is recommended.

When you issued the peer channel fetch command, there was a decent amount of output in the terminal. The last line in the logs is of interest:

2017**-**11**-**07 17:17:57.383 UTC [channelCmd] readBlock **->** DEBU 011 Received block: 2

This is telling us that the most recent configuration block for mychannel is actually block 2, **NOT** the genesis block. By default, the peer channel fetch config command returns the most **recent** configuration block for the targeted channel, which in this case is the third block. This is because the BYFN script defined anchor peers for our two organizations – Org1 and Org2 – in two separate channel update transactions.

As a result, we have the following configuration sequence:

* block 0: genesis block
* block 1: Org1 anchor peer update
* block 2: Org2 anchor peer update

**Convert the Configuration to JSON and Trim It Down**

Now we will make use of the configtxlator tool to decode this channel configuration block into JSON format (which can be read and modified by humans). We also must strip away all of the headers, metadata, creator signatures, and so on that are irrelevant to the change we want to make. We accomplish this by means of the jq tool:

configtxlator proto\_decode **--**input config\_block**.**pb **--**type common**.**Block **|** jq **.**data**.**data[0]**.**payload**.**data**.**config **>** config**.**json

This leaves us with a trimmed down JSON object – config.json, located in the fabric-samples folder inside first-network – which will serve as the baseline for our config update.

Take a moment to open this file inside your text editor of choice (or in your browser). Even after you’re done with this tutorial, it will be worth studying it as it reveals the underlying configuration structure and the other kind of channel updates that can be made. We discuss them in more detail in [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html).

**Add the Org3 Crypto Material**

**Note**

The steps you’ve taken up to this point will be nearly identical no matter what kind of config update you’re trying to make. We’ve chosen to add an org with this tutorial because it’s one of the most complex channel configuration updates you can attempt.

We’ll use the jq tool once more to append the Org3 configuration definition – org3.json – to the channel’s application groups field, and name the output – modified\_config.json.

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Application":{"groups": {"Org3MSP":.[1]}}}}}' config**.**json **./**channel**-**artifacts**/**org3**.**json **>** modified\_config**.**json

Now, within the CLI container we have two JSON files of interest – config.json and modified\_config.json. The initial file contains only Org1 and Org2 material, whereas “modified” file contains all three Orgs. At this point it’s simply a matter of re-encoding these two JSON files and calculating the delta.

First, translate config.json back into a protobuf called config.pb:

configtxlator proto\_encode **--**input config**.**json **--**type common**.**Config **--**output config**.**pb

Next, encode modified\_config.json to modified\_config.pb:

configtxlator proto\_encode **--**input modified\_config**.**json **--**type common**.**Config **--**output modified\_config**.**pb

Now use configtxlator to calculate the delta between these two config protobufs. This command will output a new protobuf binary named org3\_update.pb:

configtxlator compute\_update --channel\_id $CHANNEL\_NAME --original config.pb --updated modified\_config.pb --output org3\_update.pb

This new proto – org3\_update.pb – contains the Org3 definitions and high level pointers to the Org1 and Org2 material. We are able to forgo the extensive MSP material and modification policy information for Org1 and Org2 because this data is already present within the channel’s genesis block. As such, we only need the delta between the two configurations.

Before submitting the channel update, we need to perform a few final steps. First, let’s decode this object into editable JSON format and call it org3\_update.json:

configtxlator proto\_decode **--**input org3\_update**.**pb **--**type common**.**ConfigUpdate **|** jq **.** **>** org3\_update**.**json

Now, we have a decoded update file – org3\_update.json – that we need to wrap in an envelope message. This step will give us back the header field that we stripped away earlier. We’ll name this file org3\_update\_in\_envelope.json:

echo '{"payload":{"header":{"channel\_header":{"channel\_id":"'$CHANNEL\_NAME'", "type":2}},"data":{"config\_update":'$(cat org3\_update.json)'}}}' | jq . > org3\_update\_in\_envelope.json

Using our properly formed JSON – org3\_update\_in\_envelope.json – we will leverage the configtxlator tool one last time and convert it into the fully fledged protobuf format that Fabric requires. We’ll name our final update object org3\_update\_in\_envelope.pb:

configtxlator proto\_encode **--**input org3\_update\_in\_envelope**.**json **--**type common**.**Envelope **--**output org3\_update\_in\_envelope**.**pb

**Sign and Submit the Config Update**

Almost done!

We now have a protobuf binary – org3\_update\_in\_envelope.pb – within our CLI container. However, we need signatures from the requisite Admin users before the config can be written to the ledger. The modification policy (mod\_policy) for our channel Application group is set to the default of “MAJORITY”, which means that we need a majority of existing org admins to sign it. Because we have only two orgs – Org1 and Org2 – and the majority of two is two, we need both of them to sign. Without both signatures, the ordering service will reject the transaction for failing to fulfill the policy.

First, let’s sign this update proto as the Org1 Admin. Remember that the CLI container is bootstrapped with the Org1 MSP material, so we simply need to issue the peer channel signconfigtx command:

peer channel signconfigtx **-**f org3\_update\_in\_envelope**.**pb

The final step is to switch the CLI container’s identity to reflect the Org2 Admin user. We do this by exporting four environment variables specific to the Org2 MSP.

**Note**

Switching between organizations to sign a config transaction (or to do anything else) is not reflective of a real-world Fabric operation. A single container would never be mounted with an entire network’s crypto material. Rather, the config update would need to be securely passed out-of-band to an Org2 Admin for inspection and approval.

Export the Org2 environment variables:

*# you can issue all of these commands at once*

export CORE\_PEER\_LOCALMSPID**=**"Org2MSP"

export CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**peers**/**peer0**.**org2**.**example**.**com**/**tls**/**ca**.**crt

export CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org2**.**example**.**com**/**users**/**Admin@org2**.**example**.**com**/**msp

export CORE\_PEER\_ADDRESS**=**peer0**.**org2**.**example**.**com:9051

Lastly, we will issue the peer channel update command. The Org2 Admin signature will be attached to this call so there is no need to manually sign the protobuf a second time:

**Note**

The upcoming update call to the ordering service will undergo a series of systematic signature and policy checks. As such you may find it useful to stream and inspect the ordering node’s logs. From another shell, issue a docker logs -f orderer.example.com command to display them.

Send the update call:

peer channel update -f org3\_update\_in\_envelope.pb -c $CHANNEL\_NAME -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA

You should see a message digest indication similar to the following if your update has been submitted successfully:

2018**-**02**-**24 18:56:33.499 UTC [msp**/**identity] Sign **->** DEBU 00f Sign: digest: 3207B24E40DE2FAB87A2E42BC004FEAA1E6FDCA42977CB78C64F05A88E556ABA

You will also see the submission of our configuration transaction:

2018**-**02**-**24 18:56:33.499 UTC [channelCmd] update **->** INFO 010 Successfully submitted channel update

The successful channel update call returns a new block – block 5 – to all of the peers on the channel. If you remember, blocks 0-2 are the initial channel configurations while blocks 3 and 4 are the instantiation and invocation of the mycc chaincode. As such, block 5 serves as the most recent channel configuration with Org3 now defined on the channel.

Inspect the logs for peer0.org1.example.com:

docker logs **-**f peer0**.**org1**.**example**.**com

Follow the demonstrated process to fetch and decode the new config block if you wish to inspect its contents.

**Configuring Leader Election**

**Note**

This section is included as a general reference for understanding the leader election settings when adding organizations to a network after the initial channel configuration has completed. This sample defaults to dynamic leader election, which is set for all peers in the network in *peer-base.yaml*.

Newly joining peers are bootstrapped with the genesis block, which does not contain information about the organization that is being added in the channel configuration update. Therefore new peers are not able to utilize gossip as they cannot verify blocks forwarded by other peers from their own organization until they get the configuration transaction which added the organization to the channel. Newly added peers must therefore have one of the following configurations so that they receive blocks from the ordering service:

1. To utilize static leader mode, configure the peer to be an organization leader:

CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**false

CORE\_PEER\_GOSSIP\_ORGLEADER**=**true

**Note**

This configuration must be the same for all new peers added to the channel.

2. To utilize dynamic leader election, configure the peer to use leader election:

CORE\_PEER\_GOSSIP\_USELEADERELECTION**=**true

CORE\_PEER\_GOSSIP\_ORGLEADER**=**false

**Note**

Because peers of the newly added organization won’t be able to form membership view, this option will be similar to the static configuration, as each peer will start proclaiming itself to be a leader. However, once they get updated with the configuration transaction that adds the organization to the channel, there will be only one active leader for the organization. Therefore, it is recommended to leverage this option if you eventually want the organization’s peers to utilize leader election.

**Join Org3 to the Channel**

At this point, the channel configuration has been updated to include our new organization – Org3 – meaning that peers attached to it can now join mychannel.

First, let’s launch the containers for the Org3 peers and an Org3-specific CLI.

Open a new terminal and from first-network kick off the Org3 docker compose:

docker**-**compose **-**f docker**-**compose**-**org3**.**yaml up **-**d

This new compose file has been configured to bridge across our initial network, so the two peers and the CLI container will be able to resolve with the existing peers and ordering node. With the three new containers now running, exec into the Org3-specific CLI container:

docker exec **-**it Org3cli bash

Just as we did with the initial CLI container, export the two key environment variables: ORDERER\_CA and CHANNEL\_NAME:

export ORDERER\_CA**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**ordererOrganizations**/**example**.**com**/**orderers**/**orderer**.**example**.**com**/**msp**/**tlscacerts**/**tlsca**.**example**.**com**-**cert**.**pem **&&** export CHANNEL\_NAME**=**mychannel

Check to make sure the variables have been properly set:

echo $ORDERER\_CA && echo $CHANNEL\_NAME

Now let’s send a call to the ordering service asking for the genesis block of mychannel. The ordering service is able to verify the Org3 signature attached to this call as a result of our successful channel update. If Org3 has not been successfully appended to the channel config, the ordering service should reject this request.

**Note**

Again, you may find it useful to stream the ordering node’s logs to reveal the sign/verify logic and policy checks.

Use the peer channel fetch command to retrieve this block:

peer channel fetch 0 mychannel.block -o orderer.example.com:7050 -c $CHANNEL\_NAME --tls --cafile $ORDERER\_CA

Notice, that we are passing a 0 to indicate that we want the first block on the channel’s ledger (i.e. the genesis block). If we simply passed the peer channel fetch config command, then we would have received block 5 – the updated config with Org3 defined. However, we can’t begin our ledger with a downstream block – we must start with block 0.

Issue the peer channel join command and pass in the genesis block – mychannel.block:

peer channel join **-**b mychannel**.**block

If you want to join the second peer for Org3, export the TLS and ADDRESS variables and reissue the peer channel join command:

export CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org3**.**example**.**com**/**peers**/**peer1**.**org3**.**example**.**com**/**tls**/**ca**.**crt **&&** export CORE\_PEER\_ADDRESS**=**peer1**.**org3**.**example**.**com:12051

peer channel join **-**b mychannel**.**block

**Upgrade and Invoke Chaincode**

The final piece of the puzzle is to increment the chaincode version and update the endorsement policy to include Org3. Since we know that an upgrade is coming, we can forgo the futile exercise of installing version 1 of the chaincode. We are solely concerned with the new version where Org3 will be part of the endorsement policy, therefore we’ll jump directly to version 2 of the chaincode.

From the Org3 CLI:

peer chaincode install **-**n mycc **-**v 2.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

Modify the environment variables accordingly and reissue the command if you want to install the chaincode on the second peer of Org3. Note that a second installation is not mandated, as you only need to install chaincode on peers that are going to serve as endorsers or otherwise interface with the ledger (i.e. query only). Peers will still run the validation logic and serve as committers without a running chaincode container.

Now jump back to the **original** CLI container and install the new version on the Org1 and Org2 peers. We submitted the channel update call with the Org2 admin identity, so the container is still acting on behalf of peer0.org2:

peer chaincode install **-**n mycc **-**v 2.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

Flip to the peer0.org1 identity:

export CORE\_PEER\_LOCALMSPID**=**"Org1MSP"

export CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org1**.**example**.**com**/**peers**/**peer0**.**org1**.**example**.**com**/**tls**/**ca**.**crt

export CORE\_PEER\_MSPCONFIGPATH**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**peerOrganizations**/**org1**.**example**.**com**/**users**/**Admin@org1**.**example**.**com**/**msp

export CORE\_PEER\_ADDRESS**=**peer0**.**org1**.**example**.**com:7051

And install again:

peer chaincode install **-**n mycc **-**v 2.0 **-**p github**.**com**/**chaincode**/**chaincode\_example02**/**go**/**

Now we’re ready to upgrade the chaincode. There have been no modifications to the underlying source code, we are simply adding Org3 to the endorsement policy for a chaincode – mycc – on mychannel.

**Note**

Any identity satisfying the chaincode’s instantiation policy can issue the upgrade call. By default, these identities are the channel Admins.

Send the call:

peer chaincode upgrade -o orderer.example.com:7050 --tls $CORE\_PEER\_TLS\_ENABLED --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc -v 2.0 -c '{"Args":["init","a","90","b","210"]}' -P "OR ('Org1MSP.peer','Org2MSP.peer','Org3MSP.peer')"

You can see in the above command that we are specifying our new version by means of the v flag. You can also see that the endorsement policy has been modified to -P "OR ('Org1MSP.peer','Org2MSP.peer','Org3MSP.peer')", reflecting the addition of Org3 to the policy. The final area of interest is our constructor request (specified with the c flag).

As with an instantiate call, a chaincode upgrade requires usage of the init method. **If** your chaincode requires arguments be passed to the init method, then you will need to do so here.

The upgrade call adds a new block – block 6 – to the channel’s ledger and allows for the Org3 peers to execute transactions during the endorsement phase. Hop back to the Org3 CLI container and issue a query for the value of a. This will take a bit of time because a chaincode image needs to be built for the targeted peer, and the container needs to start:

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

We should see a response of Query Result: 90.

Now issue an invocation to move 10 from a to b:

peer chaincode invoke -o orderer.example.com:7050 --tls $CORE\_PEER\_TLS\_ENABLED --cafile $ORDERER\_CA -C $CHANNEL\_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'

Query one final time:

peer chaincode query -C $CHANNEL\_NAME -n mycc -c '{"Args":["query","a"]}'

We should see a response of Query Result: 80, accurately reflecting the update of this chaincode’s world state.

**Conclusion**

The channel configuration update process is indeed quite involved, but there is a logical method to the various steps. The endgame is to form a delta transaction object represented in protobuf binary format and then acquire the requisite number of admin signatures such that the channel configuration update transaction fulfills the channel’s modification policy.

The configtxlator and jq tools, along with the ever-growing peer channel commands, provide us with the functionality to accomplish this task.

**Updating the Channel Config to include an Org3 Anchor Peer (Optional)**

The Org3 peers were able to establish gossip connection to the Org1 and Org2 peers since Org1 and Org2 had anchor peers defined in the channel configuration. Likewise newly added organizations like Org3 should also define their anchor peers in the channel configuration so that any new peers from other organizations can directly discover an Org3 peer.

Continuing from the Org3 CLI, we will make a channel configuration update to define an Org3 anchor peer. The process will be similar to the previous configuration update, therefore we’ll go faster this time.

As before, we will fetch the latest channel configuration to get started. Inside the CLI container for Org3 fetch the most recent config block for the channel, using the peer channel fetch command.

peer channel fetch config config\_block.pb -o orderer.example.com:7050 -c $CHANNEL\_NAME --tls --cafile $ORDERER\_CA

After fetching the config block we will want to convert it into JSON format. To do this we will use the configtxlator tool, as done previously when adding Org3 to the channel. When converting it we need to remove all the headers, metadata, and signatures that are not required to update Org3 to include an anchor peer by using the jq tool. This information will be reincorporated later before we proceed to update the channel configuration.

configtxlator proto\_decode **--**input config\_block**.**pb **--**type common**.**Block **|** jq **.**data**.**data[0]**.**payload**.**data**.**config **>** config**.**json

The config.json is the now trimmed JSON representing the latest channel configuration that we will update.

Using the jq tool again, we will update the configuration JSON with the Org3 anchor peer we want to add.

jq '.channel\_group.groups.Application.groups.Org3MSP.values += {"AnchorPeers":{"mod\_policy": "Admins","value":{"anchor\_peers": [{"host": "peer0.org3.example.com","port": 11051}]},"version": "0"}}' config**.**json **>** modified\_anchor\_config**.**json

We now have two JSON files, one for the current channel configuration, config.json, and one for the desired channel configuration modified\_anchor\_config.json. Next we convert each of these back into protobuf format and calculate the delta between the two.

Translate config.json back into protobuf format as config.pb

configtxlator proto\_encode **--**input config**.**json **--**type common**.**Config **--**output config**.**pb

Translate the modified\_anchor\_config.json into protobuf format as modified\_anchor\_config.pb

configtxlator proto\_encode **--**input modified\_anchor\_config**.**json **--**type common**.**Config **--**output modified\_anchor\_config**.**pb

Calculate the delta between the two protobuf formatted configurations.

configtxlator compute\_update --channel\_id $CHANNEL\_NAME --original config.pb --updated modified\_anchor\_config.pb --output anchor\_update.pb

Now that we have the desired update to the channel we must wrap it in an envelope message so that it can be properly read. To do this we must first convert the protobuf back into a JSON that can be wrapped.

We will use the configtxlator command again to convert anchor\_update.pb into anchor\_update.json

configtxlator proto\_decode **--**input anchor\_update**.**pb **--**type common**.**ConfigUpdate **|** jq **.** **>** anchor\_update**.**json

Next we will wrap the update in an envelope message, restoring the previously stripped away header, outputting it to anchor\_update\_in\_envelope.json

echo '{"payload":{"header":{"channel\_header":{"channel\_id":"'$CHANNEL\_NAME'", "type":2}},"data":{"config\_update":'$(cat anchor\_update.json)'}}}' | jq . > anchor\_update\_in\_envelope.json

Now that we have reincorporated the envelope we need to convert it to a protobuf so it can be properly signed and submitted to the orderer for the update.

configtxlator proto\_encode **--**input anchor\_update\_in\_envelope**.**json **--**type common**.**Envelope **--**output anchor\_update\_in\_envelope**.**pb

Now that the update has been properly formatted it is time to sign off and submit it. Since this is only an update to Org3 we only need to have Org3 sign off on the update. As we are in the Org3 CLI container there is no need to switch the CLI containers identity, as it is already using the Org3 identity. Therefore we can just use the peer channel update command as it will also sign off on the update as the Org3 admin before submitting it to the orderer.

peer channel update -f anchor\_update\_in\_envelope.pb -c $CHANNEL\_NAME -o orderer.example.com:7050 --tls --cafile $ORDERER\_CA

The orderer receives the config update request and cuts a block with the updated configuration. As peers receive the block, they will process the configuration updates.

Inspect the logs for one of the peers. While processing the configuration transaction from the new block, you will see gossip re-establish connections using the new anchor peer for Org3. This is proof that the configuration update has been successfully applied!

docker logs **-**f peer0**.**org1**.**example**.**com

2019**-**06**-**12 17:08:57.924 UTC [gossip**.**gossip] learnAnchorPeers **->** INFO 89a Learning about the configured anchor peers of Org1MSP **for** channel mychannel : [{peer0**.**org1**.**example**.**com 7051}]

2019**-**06**-**12 17:08:57.926 UTC [gossip**.**gossip] learnAnchorPeers **->** INFO 89b Learning about the configured anchor peers of Org2MSP **for** channel mychannel : [{peer0**.**org2**.**example**.**com 9051}]

2019**-**06**-**12 17:08:57.926 UTC [gossip**.**gossip] learnAnchorPeers **->** INFO 89c Learning about the configured anchor peers of Org3MSP **for** channel mychannel : [{peer0**.**org3**.**example**.**com 11051}]

Congratulations, you have now made two configuration updates — one to add Org3 to the channel, and a second to define an anchor peer for Org3.