**Upgrading Your Network Components**

**Note**

When we use the term “upgrade” in this documentation, we’re primarily referring to changing the version of a component (for example, going from a v1.3 binary to a v1.4.x binary). The term “update,” on the other hand, refers not to versions but to configuration changes, such as updating a channel configuration or a deployment script. As there is no data migration, technically speaking, in Fabric, we will not use the term “migration” or “migrate” here.

**Note**

Also, if your network is not yet at Fabric v1.3, follow the instructions for [Upgrading Your Network to v1.3](http://hyperledger-fabric.readthedocs.io/en/release-1.3/upgrading_your_network_tutorial.html). The instructions in this documentation only cover moving from v1.3 to v1.4.x or from an earlier version of v1.4.x to a later version to v1.4.x.

**Overview**

If you’re unfamiliar with capabilities, check out [Channel capabilities](https://hyperledger-fabric.readthedocs.io/en/release-1.4/capabilities_concept.html) before proceeding.

While upgrading to v1.4.0 does not require any capabilities to be enabled, v1.4.2 and v1.4.3 offer new capabilities.

Specifically, the v1.4.2 and v1.4.3 capabilities enable the following features:

* Migration from Kafka to Raft consensus (requires v1.4.2 orderer and channel capabilities)
* Ability to specify orderer endpoints per organization (requires v1.4.2 channel capability)
* Ability to store private data for invalidated transactions (requires v1.4.2 application capability)
* Node OU support for admin and orderer identity classifications (extends the existing Node OU support for clients and peers) (requires v1.4.3 channel capability)

Because not all users need these new features, enabling the v1.4.2 and v1.4.3 capabilities is considered optional (though recommended), and will be detailed in a section after the main body of this tutorial.

Because the [Building Your First Network](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html) (BYFN) tutorial defaults to the “latest” binaries, if you have run it since the latest release of v1.4.x, your machine should have the latest binaries and tools installed on it and you will not be able to upgrade them.

As a result, this tutorial will provide a network based on Hyperledger Fabric v1.3 binaries as well as the v1.4.x binaries you will be upgrading to.

At a high level, our upgrade tutorial will perform the following steps:

1. Backup the ledger and MSPs.
2. Upgrade the orderer binaries to Fabric v1.4.x.
3. Upgrade the peer binaries to Fabric v1.4.x.
4. Update channel capabilities to v1.4.2 and 1.4.3 (optional).

This tutorial will demonstrate how to perform each of these steps individually with CLI commands. Instructions for both scripted execution and manual execution are included.

**Note**

Because BYFN uses a single node ordering service by default, our script brings down the entire network. However, in production environments, the ordering nodes and peers can be upgraded simultaneously and on a rolling basis. In other words, you can upgrade the binaries in any order without bringing down the network.

Because BYFN does not utilize the following components by default, our script for upgrading BYFN will not cover them:

* **Fabric CA**
* **Kafka**
* **CouchDB**
* **SDK**

The process for upgrading these components — if necessary — will be covered in a section following the tutorial. We will also show how to upgrade the Node chaincode shim.

From an operational perspective, it’s worth noting that the process for specifying logging levels has changed in v1.4.x, from CORE\_LOGGING\_LEVEL (for the peer) and ORDERER\_GENERAL\_LOGLEVEL (for the ordering nodes) in v1.3.0 to FABRIC\_LOGGING\_SPEC in v1.4.x. For more information, check out the [Fabric release notes](https://github.com/hyperledger/fabric/releases/tag/v1.4.0) for v1.4.0, when the operations service was introduced.

**Prerequisites**

If you haven’t already done so, ensure you have all of the dependencies on your machine as described in [Prerequisites](https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.html). This will include pulling the latest binaries, which you will use when upgrading.

**Launch a v1.3 network**

Before we can upgrade to v1.4.x, we must first provision a network running Fabric v1.3 images.

Just as in the BYFN tutorial, we will be operating from the first-network subdirectory within your local clone of fabric-samples. Change into that directory now. You will also want to open a few extra terminals for ease of use.

**Clean up**

We want to operate from a known state, so we will use the byfn.sh script to kill any active or stale docker containers and remove any previously generated artifacts. Run:

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

**Generate the crypto and bring up the network**

With a clean environment, launch our v1.3 BYFN network using these four commands:

git fetch origin

git [checkout](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrading_your_network_tutorial.html) v1**.**3.0

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

**./**byfn**.**sh up **-**t 3000 **-**i 1.3**.**0

**Note**

If you have locally built v1.3 images, they will be used by the example. If you get errors, please consider cleaning up your locally built v1.3 images and running the example again. This will [download](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrading_your_network_tutorial.html) v1.3 images from docker hub.

If BYFN has launched properly, you will see:

**=====================** All GOOD, BYFN execution completed **=====================**

We are now ready to upgrade our network to Hyperledger Fabric v1.4.x.

**Get the newest samples**

**Note**

The instructions below pertain to whatever is the most recently published version of v1.4.x. Please substitute 1.4.x with the version identifier of the published release that you are testing, for example, replace ‘1.4.x’ with ‘1.4.3’.

Before completing the rest of the tutorial, it’s important to switch to the v1.4.x (for example, 1.4.3) version of the samples you are upgrading to. For v1.4.3, this would be:

git checkout v1**.**4.3

**Want to upgrade now?**

Our scripts will upgrade all of the components in BYFN as well as enable any capabilities that are needed. If you are running a production network, or are an administrator of some part of a network, this script can serve as a template for performing your own upgrades.

Afterwards, we will walk you through the steps in the script and describe what each piece of code is doing in the upgrade process.

If you are updating from v1.3, you will need to set the correct system channel name, which you can do by issuing:

export CH\_NAME**=**testchainid

If you are updating from a previous version of v1.4, you will need to set a different system channel name:

export CH\_NAME**=**byfn**-**sys**-**channel

Once you have set the correct system channel name, issue these commands (substituting your preferred release number for x). Note that the script to upgrade to v1.4.3 will also upgrade the channel capabilities.

**./**byfn**.**sh upgrade **-**i 1.4**.**3

If the upgrade is successful, you should see the following:

**=====================** All GOOD, End**-**2**-**End UPGRADE Scenario execution completed **=====================**

If you want to upgrade the network manually, simply run ./byfn.sh down again and perform the steps up to — but not including — the ./byfn.sh upgrade step. Then proceed to the next section.

Note that many of the commands you’ll run in this section will not result in any output. In general, assume no output is good output.

**Upgrade the orderer containers**

Orderer containers should be upgraded in a rolling fashion (one at a time). At a high level, the orderer upgrade process goes as follows:

1. Stop the orderer.
2. [Back](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrading_your_network_tutorial.html) up the orderer’s ledger and MSP.
3. Restart the orderer with the latest images.
4. Verify upgrade completion.

As a consequence of leveraging BYFN, we have a single node orderer setup, therefore, we will only perform this process once. In a Kafka or Raft setup, however, this process will have to be repeated on each orderer.

**Note**

This tutorial uses a docker deployment. For native deployments, replace the file orderer with the one from the release artifacts. Backup the orderer.yaml and replace it with the orderer.yaml file from the release artifacts. Then port any modified variables from the backed up orderer.yaml to the new one. Utilizing a utility like diff may be helpful.

Let’s begin the upgrade process by **bringing down the orderer**:

docker stop orderer.example.com

export LEDGERS\_BACKUP=./ledgers-backup

# Note, replace '1.4.x' with a specific version, for example '1.4.3'.

# Set IMAGE\_TAG to 'latest' if you prefer to default to the images tagged 'latest' on your system.

export IMAGE\_TAG=$(go env GOARCH)-1.4.x

We have created a variable for a directory to put file backups into, and exported the IMAGE\_TAG we’d like to move to.

Once the orderer is down, you’ll want to **backup its ledger and MSP**:

mkdir -p $LEDGERS\_BACKUP

docker cp orderer.example.com:/var/hyperledger/production/orderer/ ./$LEDGERS\_BACKUP/orderer.example.com

In a production network this process would be repeated for each of the Kafka-based or Raft-based orderers in a rolling fashion.

Now **download and restart the orderer** with our new fabric image:

docker**-**compose **-**f docker**-**compose**-**cli**.**yaml up **-**d **--**no**-**deps orderer**.**example**.**com

Because our sample uses a Solo ordering service, there are no other orderers in the network that the restarted orderer must sync up to. However, in a production network leveraging Kafka or Raft, it will be a best practice to issue peer channel fetch <blocknumber> after restarting the orderer to verify that it has caught up to the other orderers.

**Upgrade the peer containers**

Next, let’s look at how to upgrade peer containers to Fabric v1.4.x. Peer containers should, like the orderers, be upgraded in a rolling fashion (one at a time). As mentioned during the orderer upgrade, orderers and peers may be upgraded in parallel, but for the purposes of this tutorial we’ve separated the processes out. At a high level, we will perform the following steps:

1. Stop the peer.
2. Back up the peer’s ledger and MSP.
3. Remove chaincode containers and images.
4. Restart the peer with latest image.
5. Verify upgrade completion.

We have four peers running in our network. We will perform this process once for each peer, totaling four upgrades.

**Note**

Again, this tutorial utilizes a docker deployment. For **native** deployments, replace the file peer with the one from the release artifacts. Backup your core.yaml and replace it with the one from the release artifacts. Port any modified variables from the backed up core.yaml to the new one. Utilizing a utility like diff may be helpful.

Let’s **bring down the first peer** with the following command:

export PEER=peer0.org1.example.com

docker stop $PEER

We can then **backup the peer’s ledger and MSP**:

mkdir -p $LEDGERS\_BACKUP

docker cp $PEER:/var/hyperledger/production ./$LEDGERS\_BACKUP/$PEER

With the peer stopped and the ledger backed up, **remove the peer chaincode containers**:

CC\_CONTAINERS=$(docker ps | grep dev-$PEER | awk '{print $1}')

if [ -n "$CC\_CONTAINERS" ] ; then docker rm -f $CC\_CONTAINERS ; fi

And the peer chaincode images:

CC\_IMAGES=$(docker images | grep dev-$PEER | awk '{print $1}')

if [ -n "$CC\_IMAGES" ] ; then docker rmi -f $CC\_IMAGES ; fi

Now we’ll re-launch the peer using the v1.4.x image tag:

docker-compose -f docker-compose-cli.yaml up -d --no-deps $PEER

**Note**

Although, BYFN supports using CouchDB, we opted for a simpler implementation in this tutorial. If you are using CouchDB, however, issue this command instead of the one above:

docker-compose -f docker-compose-cli.yaml -f docker-compose-couch.yaml up -d --no-deps $PEER

**Note**

You do not need to relaunch the chaincode container. When the peer gets a request for a chaincode, (invoke or query), it first checks if it has a copy of that chaincode running. If so, it uses it. Otherwise, as in this case, the peer launches the chaincode (rebuilding the image if required).

**Verify peer upgrade completion**

We’ve completed the upgrade for our first peer, but before we move on let’s check to ensure the upgrade has been completed properly with a chaincode invoke.

**Note**

Before you attempt this, you may want to upgrade peers from enough organizations to satisfy your endorsement policy. However, this is only mandatory if you are updating your chaincode as part of the upgrade process. If you are not updating your chaincode as part of the upgrade process, it is possible to get endorsements from peers running at different Fabric versions.

Before we get into the CLI container and issue the invoke, make sure the CLI is updated to the most current version by issuing:

docker**-**compose **-**f docker**-**compose**-**cli**.**yaml stop cli

docker**-**compose **-**f docker**-**compose**-**cli**.**yaml up **-**d **--**no**-**deps cli

Then, get back into the CLI container:

docker exec **-**it cli bash

Now you’ll need to set two environment variables — the name of the channel and the location of the ORDERER\_CA TLS certificate:

CH\_NAME**=**mychannel

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

Now you can issue the invoke:

peer chaincode invoke -o orderer.example.com:7050 --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --tls --cafile $ORDERER\_CA -C $CH\_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'

Our query earlier revealed a to have a value of 90 and we have just removed 10 with our invoke. Therefore, a query against a should reveal 80. Let’s see:

peer chaincode query **-**C mychannel **-**n mycc **-**c '{"Args":["query","a"]}'

We should see the following:

Query Result: 80

After verifying the peer was upgraded correctly, make sure to issue an exit to leave the CLI container before continuing to upgrade your peers. You can do this by repeating the process above with a different peer name exported.

export PEER**=**peer1**.**org1**.**example**.**com

export PEER**=**peer0**.**org2**.**example**.**com

export PEER**=**peer1**.**org2**.**example**.**com

**Update channel capabilities to v1.4.2 and v1.4.3 (optional)**

**Note**

While we show how to enable v1.4.2 and v1.4.3 capabilities as part of this tutorial, this is an optional step UNLESS you are leveraging the v1.4.2 or v1.4.3 features that require the capabilities.

Although Fabric binaries can and should be upgraded in a rolling fashion, it is important to finish upgrading binaries before enabling capabilities. Any binaries which are not upgraded to at least the level of the relevant capabilities may intentionally crash to indicate a misconfiguration which could otherwise result in a forked blockchain.

Once a capability has been enabled, it becomes part of the permanent record for that channel. This means that even after disabling the capability, old binaries will not be able to participate in the channel because they cannot process beyond the block which enabled the capability to get to the block which disables it. As a result, once a capability has been enabled, disabling it is neither recommended nor supported.

For this reason, think of enabling channel capabilities as a point of no return. Please experiment with the new capabilities in a test setting and be confident before proceeding to enable them in production.

Capabilities are enabled through a channel configuration transaction. For more information on updating channel configs, check out [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) or the doc on [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html).

To learn about what the new capabilities are in v1.4.2 and v1.4.3 and what they enable, refer back to the [Overview](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrading_your_network_tutorial.html#overview).

We will enable these capabilities in the following order:

1. Orderer System Channel
2. Orderer Group
3. Channel Group
4. Individual Channels
5. Orderer Group
6. Channel Group
7. Application Group

**Note**

The channel capabilities will be updated to v1.4.3. All other capabilities will be updated to v1.4.2, the latest capability level for those groups.

Updating a channel configuration is a three step process:

1. Get the latest channel config
2. Create a modified channel config
3. Create a config update transaction

**Note**

In a real world production network, these channel config updates would be handled by the admins for each channel. Because BYFN all exists on a single machine, it is possible for us to update each of these channels.

For more information on updating channel configs, click on [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) or the doc on [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html).

**Orderer System Channel Capabilities**

Make sure you are in the CLI container:

docker exec **-**it cli bash

Because only ordering organizations admins can update the ordering system channel, we need set environment variables for the system channel that will allow us to carry out these tasks. Issue each of these commands:

CORE\_PEER\_LOCALMSPID**=**"OrdererMSP"

CORE\_PEER\_TLS\_ROOTCERT\_FILE**=/**opt**/**gopath**/**src**/**github**.**com**/**hyperledger**/**fabric**/**peer**/**crypto**/**ordererOrganizations**/**example**.**com**/**orderers**/**orderer**.**example**.**com**/**msp**/**tlscacerts**/**tlsca**.**example**.**com**-**cert**.**pem

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

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

If we’re upgrading from v1.3 to v1.4.3, we need to set the system channel name to testchainid:

CH\_NAME**=**testchainid

If we’re upgrading from v1.4.1 to v1.4.3, we need to set the system channel name to byfn-sys-channel:

CH\_NAME**=**byfn**-**sys**-**channel

**Orderer Group**

The first step in updating a channel configuration is getting the latest config block:

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

configtxlator proto\_decode --input config\_block.pb --type common.Block --output config\_block.json

jq .data.data[0].payload.data.config config\_block.json > config.json

Next, add capabilities to the orderer group. The following command will create a copy of the config file and change the capability level:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].orderer}}}}}' config**.**json **./**scripts**/**capabilities**.**json **>** modified\_config**.**json

Now we can create the config update:

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

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

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

Package the config update into a transaction:

configtxlator proto\_decode --input config\_update.pb --type common.ConfigUpdate --output config\_update.json

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

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

Submit the config update transaction:

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

Our config update transaction represents the difference between the original config and the modified one, but the ordering service will translate this into a full channel config.

**Channel Group**

Now let’s move on to updating the capability level for the channel group at the orderer system level.

The first step, as before, is to get the latest channel configuration.

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

configtxlator proto\_decode --input config\_block.pb --type common.Block --output config\_block.json

jq .data.data[0].payload.data.config config\_block.json > config.json

Next, create a modified channel config:

jq **-**s '.[0] \* {"channel\_group":{"values": {"Capabilities": .[1].channel}}}' config**.**json **./**scripts**/**capabilities**.**json **>** modified\_config**.**json

Create the config update transaction:

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

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

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

Package the config update into a transaction:

configtxlator proto\_decode --input config\_update.pb --type common.ConfigUpdate --output config\_update.json

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

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

Submit the config update transaction:

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

**Enabling Capabilities on Existing Channels**

Now that we have updating the capabilities on the ordering system channel, we need to updating the configuration of any existing application channels. We only have one application channel: mychannel. So let’s set that name as an environment variable.

CH\_NAME**=**mychannel

**Orderer Group**

Like the ordering system channel, our application channel also has an orderer group.

Get the channel config:

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

configtxlator proto\_decode --input config\_block.pb --type common.Block --output config\_block.json

jq .data.data[0].payload.data.config config\_block.json > config.json

Change the capability level of the orderer group:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Orderer": {"values": {"Capabilities": .[1].orderer}}}}}' config**.**json **./**scripts**/**capabilities**.**json **>** modified\_config**.**json

Create the config update:

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

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

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

Package the config update into a transaction:

configtxlator proto\_decode --input config\_update.pb --type common.ConfigUpdate --output config\_update.json

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

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

Submit the config update transaction:

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

**Channel Group**

Now we need to change the capability of the channel group of our application channel.

As before, fetch, decode, and scope the config:

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

configtxlator proto\_decode --input config\_block.pb --type common.Block --output config\_block.json

jq .data.data[0].payload.data.config config\_block.json > config.json

Create a modified config:

jq **-**s '.[0] \* {"channel\_group":{"values": {"Capabilities": .[1].channel}}}' config**.**json **./**scripts**/**capabilities**.**json **>** modified\_config**.**json

Create the config update:

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

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

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

Package the config update into a transaction:

configtxlator proto\_decode --input config\_update.pb --type common.ConfigUpdate --output config\_update.json

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

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

Because we’re updating the config of the channel group, the relevant orgs — Org1, Org2, and the OrdererOrg — need to sign it. This task would usually be performed by the individual org admins, but in BYFN, as we’ve said, this task falls to us.

First, switch into Org1 and sign the update:

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

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

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

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

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

And do the same as Org2:

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

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

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

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

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

And as the OrdererOrg:

CORE\_PEER\_LOCALMSPID="OrdererMSP"

CORE\_PEER\_TLS\_ROOTCERT\_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem

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

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

**Application Group**

For the application group, we will need to reset the environment variables as one organization:

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

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

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

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

Now, get the latest channel config (this process should be very familiar by now):

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

configtxlator proto\_decode --input config\_block.pb --type common.Block --output config\_block.json

jq .data.data[0].payload.data.config config\_block.json > config.json

Create a modified channel config:

jq **-**s '.[0] \* {"channel\_group":{"groups":{"Application": {"values": {"Capabilities": .[1].application}}}}}' config**.**json **./**scripts**/**capabilities**.**json **>** modified\_config**.**json

Note what we’re changing here: Capabilities are being added as a value of the Application group under channel\_group (in mychannel).

Create a config update transaction:

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

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

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

Package the config update into a transaction:

configtxlator proto\_decode --input config\_update.pb --type common.ConfigUpdate --output config\_update.json

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

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

Org1 signs the transaction:

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

Set the environment variables as Org2:

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

Org2 submits the config update transaction with its signature:

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

Congratulations! You have now enabled capabilities on all of your channels.

**Verify a transaction after Capabilities have been Enabled**

But let’s test just to make sure by moving 10 from a to b, as before:

peer chaincode invoke -o orderer.example.com:7050 --peerAddresses peer0.org1.example.com:7051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt --peerAddresses peer0.org2.example.com:9051 --tlsRootCertFiles /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt --tls --cafile $ORDERER\_CA -C $CH\_NAME -n mycc -c '{"Args":["invoke","a","b","10"]}'

And then querying the value of a, which should reveal a value of 70. Let’s see:

peer chaincode query **-**C mychannel **-**n mycc **-**c '{"Args":["query","a"]}'

We should see the following:

Query Result: 70

In which case we have successfully added capabilities to all of our channels.

**Upgrading components BYFN does not support**

Although this is the end of our update tutorial, there are other components that exist in production networks that are not covered in this tutorial. In this section, we’ll talk through the process of updating them.

**Fabric CA container**

To learn how to upgrade your Fabric CA server, click over to the [CA documentation](http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html#upgrading-the-server).

**Upgrade Node SDK clients**

**Note**

Upgrade Fabric and Fabric CA before upgrading Node SDK clients. Fabric and Fabric CA are tested for backwards compatibility with older SDK clients. While newer SDK clients often work with older Fabric and Fabric CA releases, they may expose features that are not yet available in the older Fabric and Fabric CA releases, and are not tested for full compatibility.

Use NPM to upgrade any Node.js client by executing these commands in the root directory of your application:

npm install fabric**-**client@latest

npm install fabric**-**ca**-**client@latest

These commands install the new version of both the Fabric client and Fabric-CA client and write the new versions package.json.

**Upgrading the Kafka cluster**

**Note**

If you intend to migrate from a Kafka-based ordering service to a Raft-based ordering service, check out [Migrating from Kafka to Raft](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html).

It is not required, but it is recommended that the Kafka cluster be upgraded and kept up to date along with the rest of Fabric. Newer versions of Kafka support older protocol versions, so you may upgrade Kafka before or after the rest of Fabric.

If you followed the [Upgrading Your Network to v1.3 tutorial](http://hyperledger-fabric.readthedocs.io/en/release-1.3/upgrading_your_network_tutorial.html), your Kafka cluster should be at v1.0.0. If it isn’t, refer to the official Apache Kafka documentation on [upgrading Kafka from previous versions](https://kafka.apache.org/documentation/#upgrade) to upgrade the Kafka cluster brokers.

**Upgrading Zookeeper**

An Apache Kafka cluster requires an Apache Zookeeper cluster. The Zookeeper API has been stable for a long time and, as such, almost any version of Zookeeper is tolerated by Kafka. Refer to the [Apache Kafka upgrade](https://kafka.apache.org/documentation/#upgrade) documentation in case there is a specific requirement to upgrade to a specific version of Zookeeper. If you would like to upgrade your Zookeeper cluster, some information on upgrading Zookeeper cluster can be found in the [Zookeeper FAQ](https://cwiki.apache.org/confluence/display/ZOOKEEPER/FAQ).

**Upgrading CouchDB**

If you are using CouchDB as state database, you should upgrade the peer’s CouchDB at the same time the peer is being upgraded. CouchDB v2.2.0 has been tested with Fabric v1.4.x.

To upgrade CouchDB:

1. Stop CouchDB.
2. Backup CouchDB data directory.
3. Install CouchDB v2.2.0 binaries or update deployment scripts to use a new Docker image (CouchDB v2.2.0 pre-configured Docker image is provided alongside Fabric v1.4).
4. Restart CouchDB.

**Upgrade Node chaincode shim**

To move to the new version of the Node chaincode shim a developer would need to:

1. Change the level of fabric-shim in their chaincode package.json from 1.3 to 1.4.x.
2. Repackage this new chaincode package and install it on all the endorsing peers in the channel.
3. Perform an upgrade to this new chaincode. To see how to do this, check out [peer chaincode](https://hyperledger-fabric.readthedocs.io/en/release-1.4/commands/peerchaincode.html).

**Note**

This flow isn’t specific to moving from 1.3 to 1.4.x It is also how one would upgrade from any incremental version of the node fabric shim.

**Upgrade Chaincodes with vendored shim**

**Note**

The v1.3.0 shim is compatible with the v1.4.x peer, but, it is still best practice to upgrade the chaincode shim to match the current level of the peer.

A number of third party tools exist that will allow you to vendor a chaincode shim. If you used one of these tools, use the same one to update your vendoring and re-package your chaincode.

If your chaincode vendors the shim, after updating the shim version, you must install it to all peers which already have the chaincode. Install it with the same name, but a newer version. Then you should execute a chaincode upgrade on each channel where this chaincode has been deployed to move to the new version.

If you did not vendor your chaincode, you can skip this step entirely.