# Operations Guides

* [Upgrading to the Newest Version of Fabric](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrade_to_newest_version.html)
* [Setting up an ordering node](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer_deploy.html)
* [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html)
* [Membership Service Providers (MSP)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html)
* [Channel Configuration (configtx)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/configtx.html)
* [Defining capability requirements](https://hyperledger-fabric.readthedocs.io/en/release-1.4/capability_requirements.html)
* [Endorsement policies](https://hyperledger-fabric.readthedocs.io/en/release-1.4/endorsement-policies.html)
* [Pluggable transaction endorsement and validation](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html)
* [Access Control Lists (ACL)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html)
* [MSP Implementation with Identity Mixer](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html)
* [Identity Mixer MSP configuration generator (idemixgen)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemixgen.html)
* [The Operations Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/operations_service.html)
* [Metrics Reference](https://hyperledger-fabric.readthedocs.io/en/release-1.4/metrics_reference.html)
* [Error handling](https://hyperledger-fabric.readthedocs.io/en/release-1.4/error-handling.html)
* [Logging Control](https://hyperledger-fabric.readthedocs.io/en/release-1.4/logging-control.html)
* [Securing Communication With Transport Layer Security (TLS)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/enable_tls.html)
* [Configuring and operating a Raft ordering service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html)
* [Migrating from Kafka to Raft](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html)
* [Bringing up a Kafka-based Ordering Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html)

# Upgrading to the Newest Version of Fabric

At a high level, upgrading a Fabric network from v1.3 to v1.4 can be performed by following these steps:

* Upgrade the binaries for the ordering service, the Fabric CA, and the peers. These upgrades may be done in parallel.
* Upgrade client SDKs.
* If upgrading to v1.4.2, enable the v1.4.2 channel capabilities.
* (Optional) Upgrade the Kafka cluster.

To help understand this process, we’ve created the [Upgrading Your Network Components](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrading_your_network_tutorial.html) tutorial that will take you through most of the major upgrade steps, including upgrading peers, orderers, as well as the capabilities. We’ve included both a script as well as the individual steps to achieve these upgrades.

Because our tutorial leverages the [Building Your First Network](https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html) (BYFN) sample, it has certain limitations (it does not use Fabric CA, for example). Therefore we have included a section at the end of the tutorial that will show how to upgrade your CA, Kafka clusters, CouchDB, Zookeeper, vendored chaincode shims, and Node SDK clients.

While upgrade to v1.4.0 does not require any capabilities to be enabled, v1.4.2 offers new capabilities at the orderer, channel, and application levels. Specifically, the v1.4.2 capabilities enable the following [features](https://hyperledger-fabric.readthedocs.io/en/release-1.4/upgrade_to_newest_version.html):

* 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)

If you want to learn more about capability requirements, check out the [Defining capability requirements](https://hyperledger-fabric.readthedocs.io/en/release-1.4/capability_requirements.html) documentation.

# Setting up an ordering node

In this topic, we’ll describe the process for bootstrapping an ordering node. If you want more information about the different ordering service implementations and their relative strengths and weaknesses, check out our [conceptual documentation about ordering](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html).

Broadly, this topic will involve a few interrelated steps:

1. Creating the organization your ordering node belongs to (if you have not already done so)
2. Configuring your node (using orderer.yaml)
3. Creating the genesis block for the orderer system channel
4. Bootstrapping the orderer

Note: this topic assumes you have already pulled the Hyperledger Fabric orderer images from docker hub.

## Create an organization definition

Like peers, all orderers must belong to an organization that must be created before the orderer itself is created. This organization has a definition encapsulated by a [Membership Service Provider](https://hyperledger-fabric.readthedocs.io/en/release-1.4/membership/membership.html) (MSP) that is created by a Certificate Authority (CA) dedicated to creating the certificates and MSP for the organization.

For information about creating a CA and using it to [create](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer_deploy.html) users and an MSP, check out the [Fabric CA user’s guide](https://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html).

## Configure your node

The configuration of the orderer is handled through a yaml filed called orderer.yaml. The FABRIC\_CFG\_PATH environment variable is used to point to an orderer.yaml file you’ve configured, which will extract a series of files and certificates on your file system.

To look at a sample orderer.yaml, check out the [fabric-samples github repo](https://github.com/hyperledger/fabric/blob/release-1.4/sampleconfig/orderer.yaml), which **should be read and studied closely** before proceeding. Note in particular a few values:

* LocalMSPID — this is the name of the MSP, generated by your CA, of your orderer organization. This is where your orderer organization admins will be listed.
* LocalMSPDir — the place in your file system where the local MSP is located.
* # TLS enabled, Enabled: false. This is where you specify whether you want to [enable TLS](https://hyperledger-fabric.readthedocs.io/en/release-1.4/enable_tls.html). If you set this value to true, you will have to specify the locations of the relevant TLS certificates. Note that this is mandatory for Raft nodes.
* GenesisFile — this is the name of the genesis block you will generate for this ordering service.
* GenesisMethod — the method by which the genesis block is created. This can be either file, in which the file in the GenesisFile is specified, and provisional, in which the profile in GenesisProfile is used.

If you are deploying this node as part of a cluster (for example, as part of a cluster of Raft nodes), make note of the Cluster and Consensus sections.

If you plan to deploy a Kafka based ordering service, you will need to complete the Kafka section.

## Generate the genesis block of the orderer

The first block of a newly created channel is known as a “genesis block”. If this genesis block is being created as part of the creation of a **new network** (in other words, if the orderer being created will not be joined to an existing cluster of orderers), then this genesis block will be the first block of the “orderer system channel” (also known as the “ordering system channel”), a special channel managed by the orderer admins which includes a list of the organizations permitted to create channels. The genesis block of the orderer system channel is special: it must be created and included in the configuration of the node before the node can be started.

To learn how to create a genesis block using the configtxgen tool, check out [Channel Configuration (configtx)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/configtx.html).

## Bootstrap the ordering node

Once you have built the images, created the MSP, configured your orderer.yaml, and created the genesis block, you’re ready to [start](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer_deploy.html) your orderer using a command that will look similar to:

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

With the address of your orderer replacing orderer.example.com.

# Updating a Channel Configuration

## What is a Channel Configuration?

Channel configurations contain all of the information relevant to the administration of a channel. Most importantly, the channel configuration specifies which organizations are members of channel, but it also includes other channel-wide configuration information such as channel access policies and block batch sizes.

This configuration is stored on the ledger in a **block**, and is therefore known as a configuration (config) block. Configuration blocks contain a single configuration. The first of these blocks is known as the “genesis block” and contains the initial configuration required to bootstrap a channel. Each time the configuration of a channel changes it is done through a new configuration block, with the latest configuration block representing the current channel configuration. Orderers and peers keep the current channel configuration in memory to facilitate all channel operations such as cutting a new block and validating block transactions.

Because configurations are stored in blocks, updating a config happens through a process called a “configuration transaction” (even though the process is a little different from a normal transaction). Updating a config is a process of pulling the config, translating into a format that humans can read, modifying it and then submitting it for approval.

For a more in-depth look at the process for pulling a config and translating it into JSON, check out [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html). In this doc, we’ll be focusing on the different ways you can edit a config and the process for getting it signed.

## Editing a Config

Channels are highly configurable, but not infinitely so. Different configuration elements have different modification policies (which specify the group of identities required to [sign](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html) the config update).

To see the scope of what’s possible to change it’s important to look at a config in JSON format. The [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) tutorial generates one, so if you’ve gone through that [doc](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html) you can simply refer to it. For those who have not, we’ll provide one here (for ease of readability, it might be helpful to put this config into a viewer that supports JSON folding, like atom or Visual Studio).

**Click here to see the config**

A config might look intimidating in this form, but once you study it you’ll see that it has a logical structure.

Beyond the definitions of the policies – defining who can do certain things at the channel level, and who has the permission to change who can change the config – channels also have other kinds of [features](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html) that can be modified using a config update. [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) takes you through one of the most important – adding an org to a channel. Some other things that are possible to change with a config update include:

* **Batch Size.** These parameters dictate the number and size of transactions in a block. No block will appear larger than absolute\_max\_bytes large or with more than max\_message\_count transactions inside the block. If it is possible to construct a block under preferred\_max\_bytes, then a block will be cut prematurely, and transactions larger than this size will appear in their own block.
* {
* "absolute\_max\_bytes": 102760448,
* "max\_message\_count": 10,
* "preferred\_max\_bytes": 524288
* }
* **Batch Timeout.** The amount of time to wait after the first transaction arrives for additional transactions before cutting a block. Decreasing this value will improve latency, but decreasing it too much may decrease throughput by not allowing the block to fill to its maximum capacity.
* { "timeout": "2s" }
* **Channel Restrictions.** The total number of channels the orderer is willing to allocate may be specified as max\_count. This is primarily useful in pre-production environments with weak consortium ChannelCreation policies.
* {
* "max\_count":1000
* }
* **Channel Creation Policy.** Defines the policy value which will be set as the mod\_policy for the Application group of new channels for the consortium it is defined in. The signature set attached to the channel creation request will be checked against the instantiation of this policy in the new channel to ensure that the channel creation is authorized. Note that this config value is only set in the orderer system channel.
* {
* "type": 3,
* "value": {
* "rule": "ANY",
* "sub\_policy": "Admins"
* }
* }
* **Kafka brokers.** When ConsensusType is set to kafka, the brokers list enumerates some subset (or preferably all) of the Kafka brokers for the orderer to initially connect to at startup. Note that it is not possible to change your consensus type after it has been established (during the bootstrapping of the genesis block).
* {
* "brokers": [
* "kafka0:9092",
* "kafka1:9092",
* "kafka2:9092",
* "kafka3:9092"
* ]
* }
* **Anchor Peers Definition.** Defines the location of the anchor peers for each Org.
* {
* "host": "peer0.org2.example.com",
* "port": 9051
* }
* **Hashing Structure.** The block data is an array of byte arrays. The hash of the block data is computed as a Merkle tree. This value specifies the width of that Merkle tree. For the time being, this value is fixed to 4294967295 which corresponds to a simple flat hash of the concatenation of the block data bytes.
* { "width": 4294967295 }
* **Hashing Algorithm.** The algorithm used for computing the hash values encoded into the blocks of the blockchain. In particular, this affects the data hash, and the previous block hash fields of the block. Note, this field currently only has one valid value (SHA256) and should not be changed.
* { "name": "SHA256" }
* **Block Validation.** This policy specifies the signature requirements for a block to be considered valid. By default, it requires a signature from some member of the ordering org.
* {
* "type": 3,
* "value": {
* "rule": "ANY",
* "sub\_policy": "Writers"
* }
* }
* **Orderer Address.** A list of addresses where clients may invoke the orderer Broadcast and Deliver functions. The peer randomly chooses among these addresses and fails over between them for retrieving blocks.
* {
* "addresses": [
* "orderer.example.com:7050"
* ]
* }

Just as we add an Org by adding their artifacts and MSP information, you can remove them by reversing the process.

**Note** that once the consensus type has been defined and the network has been bootstrapped, it is not possible to change it through a configuration update.

There is another important channel configuration (especially for v1.1) known as **Capability Requirements**. It has its own doc that can be found [here](https://hyperledger-fabric.readthedocs.io/en/release-1.4/capability_requirements.html).

Let’s say you want to edit the block batch size for the channel (because this is a single numeric field, it’s one of the easiest changes to make). First to make referencing the JSON path easy, we define it as an environment variable.

To establish this, take a look at your config, find what you’re looking for, and back track the path.

If you find batch size, for example, you’ll see that it’s a value of the Orderer. Orderer can be found under groups, which is under channel\_group. The batch size value has a parameter under value of max\_message\_count.

Which would make the path this:

export MAXBATCHSIZEPATH**=**".channel\_group.groups.Orderer.values.BatchSize.value.max\_message\_count"

Next, display the value of that property:

jq "$MAXBATCHSIZEPATH" config**.**json

Which should return a value of 10 (in our sample network at least).

Now, let’s set the new batch size and display the new value:

jq "$MAXBATCHSIZEPATH = 20" config**.**json **>** modified\_config**.**json

jq "$MAXBATCHSIZEPATH" modified\_config**.**json

Once you’ve modified the JSON, it’s ready to be converted and submitted. The scripts and steps in [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) will take you through the process for converting the JSON, so let’s look at the process of submitting it.

## Get the Necessary Signatures

Once you’ve successfully generated the protobuf file, it’s time to get it signed. To do this, you need to know the relevant policy for whatever it is you’re trying to change.

By default, editing the configuration of:

* **A particular org** (for example, changing anchor peers) requires only the admin signature of that org.
* **The application** (like who the member orgs are) requires a majority of the application organizations’ admins to sign.
* **The orderer** requires a majority of the ordering organizations’ admins (of which there are by default only 1).
* **The top level channel group** requires both the agreement of a majority of application organization admins and orderer organization admins.

If you have made changes to the default policies in the channel, you’ll need to compute your signature requirements accordingly.

Note: you may be able to script the signature collection, dependent on your application. In general, you may always collect more signatures than are required.

The actual process of getting these signatures will depend on how you’ve set up your system, but there are two main implementations. Currently, the Fabric command line defaults to a “pass it along” system. That is, the Admin of the Org proposing a config update sends the update to someone else (another Admin, typically) who needs to sign it. This Admin signs it (or doesn’t) and passes it along to the next Admin, and so on, until there are enough signatures for the config to be submitted.

This has the virtue of simplicity – when there are enough signatures, the last Admin can simply submit the config transaction (in Fabric, the peer channel update command includes a signature by default). However, this process will only be practical in smaller channels, since the “pass it along” method can be time consuming.

The other option is to submit the update to every Admin on a channel and wait for enough signatures to come back. These signatures can then be stitched together and submitted. This makes life a bit more difficult for the Admin who created the config update (forcing them to deal with a file per signer) but is the recommended workflow for users which are developing Fabric management applications.

Once the config has been added to the ledger, it will be a best practice to pull it and convert it to JSON to check to make sure everything was added correctly. This will also serve as a useful copy of the latest config.

# Membership Service Providers (MSP)

The [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html) serves to provide [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html) on the setup and best practices for MSPs.

Membership Service Provider (MSP) is a component that aims to offer an abstraction of a membership operation architecture.

In particular, MSP abstracts away all cryptographic mechanisms and protocols behind issuing and validating certificates, and [user](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html) authentication. An MSP may define their own notion of identity, and the rules by which those identities are governed (identity validation) and authenticated (signature generation and verification).

A Hyperledger Fabric blockchain network can be governed by one or more MSPs. This provides modularity of membership operations, and interoperability across different membership standards and architectures.

In the rest of this document we elaborate on the setup of the MSP implementation supported by Hyperledger Fabric, and discuss best practices concerning its use.

## MSP Configuration

To setup an instance of the MSP, its configuration needs to be specified locally at each peer and orderer (to enable peer, and orderer signing), and on the channels to enable peer, orderer, client identity validation, and respective signature verification (authentication) by and for all channel members.

Firstly, for each MSP a name needs to be specified in order to reference that MSP in the network (e.g. msp1, org2, and org3.divA). This is the name under which membership rules of an MSP representing a consortium, organization or organization division is to be referenced in a channel. This is also referred to as the MSP Identifier or MSP ID. MSP Identifiers are required to be unique per MSP instance. For example, shall two MSP instances with the same identifier be detected at the system channel genesis, orderer setup will fail.

In the case of default implementation of MSP, a set of parameters need to be specified to allow for identity (certificate) validation and signature verification. These parameters are deduced by [RFC5280](http://www.ietf.org/rfc/rfc5280.txt), and include:

* A list of self-signed (X.509) certificates to constitute the root of trust
* A list of X.509 certificates to represent intermediate CAs this provider considers for certificate validation; these certificates ought to be certified by exactly one of the certificates in the root of trust; intermediate CAs are optional parameters
* A list of X.509 certificates with a verifiable certificate path to exactly one of the certificates of the root of trust to represent the administrators of this MSP; owners of these certificates are authorized to request changes to this MSP configuration (e.g. root CAs, intermediate CAs)
* A list of Organizational Units that valid members of this MSP should include in their X.509 certificate; this is an optional configuration parameter, used when, e.g., multiple organizations leverage the same root of trust, and intermediate CAs, and have reserved an OU field for their members
* A list of certificate revocation lists (CRLs) each corresponding to exactly one of the listed (intermediate or root) MSP Certificate Authorities; this is an optional parameter
* A list of self-signed (X.509) certificates to constitute the TLS root of trust for TLS certificate.
* A list of X.509 certificates to represent intermediate TLS CAs this provider considers; these certificates ought to be certified by exactly one of the certificates in the TLS root of trust; intermediate CAs are optional parameters.

Valid identities for this MSP instance are required to satisfy the following conditions:

* They are in the form of X.509 certificates with a verifiable certificate path to exactly one of the root of trust certificates;
* They are not included in any CRL;
* And they list one or more of the Organizational Units of the MSP configuration in the OU field of their X.509 certificate structure.

For more information on the validity of identities in the current MSP implementation, we refer the reader to [MSP Identity Validity Rules](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp-identity-validity-rules.html).

In addition to verification related parameters, for the MSP to enable the node on which it is instantiated to [sign](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html) or authenticate, one needs to specify:

* The signing key used for signing by the node (currently only ECDSA keys are supported), and
* The node’s X.509 certificate, that is a valid identity under the verification parameters of this MSP.

It is important to note that MSP identities never expire; they can only be revoked by adding them to the appropriate CRLs. Additionally, there is currently no support for enforcing revocation of TLS certificates.

## How to generate MSP certificates and their signing keys?

To generate X.509 certificates to feed its MSP configuration, the application can use [Openssl](https://www.openssl.org/). We emphasize that in Hyperledger Fabric there is no support for certificates including RSA keys.

Alternatively one can use cryptogen tool, whose operation is explained in [Getting Started](https://hyperledger-fabric.readthedocs.io/en/release-1.4/getting_started.html).

[Hyperledger Fabric CA](http://hyperledger-fabric-ca.readthedocs.io/en/latest/) can also be used to generate the keys and certificates needed to configure an MSP.

## MSP setup on the peer & orderer side

To set up a local MSP (for either a peer or an orderer), the administrator should [create](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html) a folder (e.g. $MY\_PATH/mspconfig) that contains six subfolders and a file:

1. a folder admincerts to include PEM files each corresponding to an administrator certificate
2. a folder cacerts to include PEM files each corresponding to a root CA’s certificate
3. (optional) a folder intermediatecerts to include PEM files each corresponding to an intermediate CA’s certificate
4. (optional) a file config.yaml to configure the supported Organizational Units and identity classifications (see respective sections below).
5. (optional) a folder crls to include the considered CRLs
6. a folder keystore to include a PEM file with the node’s signing key; we emphasise that currently RSA keys are not supported
7. a folder signcerts to include a PEM file with the node’s X.509 certificate
8. (optional) a folder tlscacerts to include PEM files each corresponding to a TLS root CA’s certificate
9. (optional) a folder tlsintermediatecerts to include PEM files each corresponding to an intermediate TLS CA’s certificate

In the configuration file of the node (core.yaml file for the peer, and orderer.yaml for the orderer), one needs to specify the path to the mspconfig folder, and the MSP Identifier of the node’s MSP. The path to the mspconfig folder is expected to be relative to FABRIC\_CFG\_PATH and is provided as the value of parameter mspConfigPath for the peer, and LocalMSPDir for the orderer. The identifier of the node’s MSP is provided as a value of parameter localMspId for the peer and LocalMSPID for the orderer. These variables can be overridden via the environment using the CORE prefix for peer (e.g. CORE\_PEER\_LOCALMSPID) and the ORDERER prefix for the orderer (e.g. ORDERER\_GENERAL\_LOCALMSPID). Notice that for the orderer setup, one needs to generate, and provide to the orderer the genesis block of the system channel. The MSP configuration needs of this block are detailed in the next section.

Reconfiguration of a “local” MSP is only possible manually, and requires that the peer or orderer process is restarted. In subsequent releases we aim to offer online/dynamic reconfiguration (i.e. without requiring to stop the node by using a node managed system chaincode).

## Organizational Units

In order to configure the list of Organizational Units that valid members of this MSP should include in their X.509 certificate, the config.yaml file needs to specify the organizational unit (OU, for short) identifiers. You can find an example below:

OrganizationalUnitIdentifiers:

**-** Certificate: "cacerts/cacert1.pem"

OrganizationalUnitIdentifier: "commercial"

**-** Certificate: "cacerts/cacert2.pem"

OrganizationalUnitIdentifier: "administrators"

The above example declares two organizational unit identifiers: **commercial** and **administrators**. An MSP identity is valid if it carries at least one of these organizational unit identifiers. The Certificate field refers to the CA or intermediate CA certificate path under which identities, having that specific OU, should be validated. The path is relative to the MSP root folder and cannot be empty.

## Identity Classification

The default MSP implementation allows organizations to further classify identities into clients, admins, peers, and orderers based on the OUs of their x509 certificates.

* An identity should be classified as a **client** if it transacts on the network.
* An identity should be classified as an **admin** if it handles administrative tasks such as joining a peer to a channel or signing a channel configuration update transaction.
* An identity should be classified as a **peer** if it endorses or commits transactions.
* An identity should be classified as an **orderer** if belongs to an ordering node.

In order to define the clients, admins, peers, and orderers of a given MSP, the config.yaml file needs to be set appropriately. You can find an example NodeOU section of the config.yaml file below:

NodeOUs:

Enable: true

ClientOUIdentifier:

Certificate: "cacerts/cacert.pem"

OrganizationalUnitIdentifier: "client"

AdminOUIdentifier:

Certificate: "cacerts/cacert.pem"

OrganizationalUnitIdentifier: "admin"

PeerOUIdentifier:

Certificate: "cacerts/cacert.pem"

OrganizationalUnitIdentifier: "peer"

OrdererOUIdentifier:

Certificate: "cacerts/cacert.pem"

OrganizationalUnitIdentifier: "orderer"

Identity classification is enabled when NodeOUs.Enable is set to true. Then the client (admin, peer, orderer) organizational unit identifier is defined by setting the properties of the NodeOUs.ClientOUIdentifier (NodeOUs.AdminOUIdentifier, NodeOUs.PeerOUIdentifier, NodeOUs.OrdererOUIdentifier) key:

1. OrganizationalUnitIdentifier: Is the OU value that the x509 certificate needs to contain to be considered a client (admin, peer, orderer respectively). If this field is empty, then the classification is not applied.
2. Certificate: Set this to the path of the CA or intermediate CA certificate under which client (peer, admin or orderer) identities should be validated. The field is relative to the MSP root folder. This field is optional. You can leave this field blank and allow the certificate to be validated under any CA defined in the MSP configuration.

Notice that if the NodeOUs.ClientOUIdentifier section (NodeOUs.AdminOUIdentifier, NodeOUs.PeerOUIdentifier, NodeOUs.OrdererOUIdentifier) is missing, then the classification is not applied. If NodeOUs.Enable is set to true and no classification keys are defined, then identity classification is assumed to be disabled.

Identities can use organizational units to be classified as either a client, an admin, a peer, or an orderer. The four classifications are mutually exclusive. The 1.1 channel capability needs to be enabled before identities can be classified as clients or peers. The 1.4.3 channel capability needs to be enabled for identities to be classified as an admin or orderer.

Classification allows identities to be classified as admins (and conduct administrator actions) without the certificate being stored in the admincerts folder of the MSP. Instead, the admincerts folder can remain empty and administrators can be created by enrolling identities with the admin OU. Certificates in the admincerts folder will still grant the role of administrator to their bearer, provided that they possess the client or admin OU.

## Channel MSP setup

At the genesis of the system, verification parameters of all the MSPs that appear in the network need to be specified, and included in the system channel’s genesis block. Recall that MSP verification parameters consist of the MSP identifier, the root of trust certificates, intermediate CA and admin certificates, as well as OU specifications and CRLs. The system genesis block is provided to the orderers at their setup phase, and allows them to authenticate channel creation requests. Orderers would reject the system genesis block, if the latter includes two MSPs with the same identifier, and consequently the bootstrapping of the network would fail.

For application channels, the verification components of only the MSPs that govern a channel need to reside in the channel’s genesis block. We emphasize that it is **the responsibility of the application** to ensure that correct MSP configuration information is included in the genesis blocks (or the most recent configuration block) of a channel prior to instructing one or more of their peers to join the channel.

When bootstrapping a channel with the help of the configtxgen tool, one can configure the channel MSPs by including the verification parameters of MSP in the mspconfig folder, and setting that path in the relevant section in configtx.yaml.

Reconfiguration of an MSP on the channel, including announcements of the certificate revocation lists associated to the CAs of that MSP is achieved through the creation of a config\_update object by the owner of one of the administrator certificates of the MSP. The client application managed by the admin would then announce this update to the channels in which this MSP appears.

## Best Practices

In this section we elaborate on best practices for MSP configuration in commonly met scenarios.

**1) Mapping between organizations/corporations and MSPs**

We recommend that there is a one-to-one mapping between organizations and MSPs. If a different type of mapping is chosen, the following needs to be to considered:

* **One organization employing various MSPs.** This corresponds to the case of an organization including a variety of divisions each represented by its MSP, either for management independence reasons, or for privacy reasons. In this case a peer can only be owned by a single MSP, and will not recognize peers with identities from other MSPs as peers of the same organization. The implication of this is that peers may share through gossip organization-scoped data with a set of peers that are members of the same subdivision, and NOT with the full set of providers constituting the actual organization.
* **Multiple organizations using a single MSP.** This corresponds to a case of a consortium of organizations that are governed by similar membership architecture. One needs to know here that peers would propagate organization-scoped messages to the peers that have an identity under the same MSP regardless of whether they belong to the same actual organization. This is a limitation of the granularity of MSP definition, and/or of the peer’s configuration.

**2) One organization has different divisions (say organizational units), to** **which it wants to grant access to different channels.**

Two ways to handle this:

* **Define one MSP to accommodate membership for all organization’s members**. Configuration of that MSP would consist of a list of root CAs, intermediate CAs and admin certificates; and membership identities would include the organizational unit (OU) a member belongs to. Policies can then be defined to capture members of a specific OU, and these policies may constitute the read/write policies of a channel or endorsement policies of a chaincode. A limitation of this approach is that gossip peers would consider peers with membership identities under their local MSP as members of the same organization, and would consequently gossip with them organization-scoped data (e.g. their status).
* **Defining one MSP to represent each division**. This would involve specifying for each division, a set of certificates for root CAs, intermediate CAs, and admin Certs, such that there is no overlapping certification path across MSPs. This would mean that, for example, a different intermediate CA per subdivision is employed. Here the disadvantage is the management of more than one MSPs instead of one, but this circumvents the issue present in the previous approach. One could also define one MSP for each division by leveraging an OU extension of the MSP configuration.

**3) Separating clients from peers of the same organization.**

In many cases it is required that the “type” of an identity is retrievable from the identity itself (e.g. it may be needed that endorsements are guaranteed to have derived by peers, and not clients or nodes acting solely as orderers).

There is limited support for such requirements.

One way to allow for this separation is to create a separate intermediate CA for each node type - one for clients and one for peers/orderers; and configure two different MSPs - one for clients and one for peers/orderers. Channels this organization should be accessing would need to include both MSPs, while endorsement policies will leverage only the MSP that refers to the peers. This would ultimately result in the organization being mapped to two MSP instances, and would have certain consequences on the way peers and clients interact.

Gossip would not be drastically impacted as all peers of the same organization would still belong to one MSP. Peers can restrict the execution of certain system chaincodes to local MSP based policies. For example, peers would only execute “joinChannel” request if the request is signed by the admin of their local MSP who can only be a client (end-user should be sitting at the origin of that request). We can go around this inconsistency if we accept that the only clients to be members of a peer/orderer MSP would be the administrators of that MSP.

Another point to be considered with this approach is that peers authorize event registration requests based on membership of request originator within their local MSP. Clearly, since the originator of the request is a client, the request originator is always deemed to belong to a different MSP than the requested peer and the peer would reject the request.

**4) Admin and CA certificates.**

It is important to set MSP admin certificates to be different than any of the certificates considered by the MSP for root of trust, or intermediate CAs. This is a common (security) practice to separate the duties of management of membership components from the issuing of new certificates, and/or validation of existing ones.

**5) Blacklisting an intermediate CA.**

As mentioned in previous sections, reconfiguration of an MSP is achieved by reconfiguration mechanisms (manual reconfiguration for the local MSP instances, and via properly constructed config\_update messages for MSP instances of a channel). Clearly, there are two ways to ensure an intermediate CA considered in an MSP is no longer considered for that MSP’s identity validation:

1. Reconfigure the MSP to no longer include the certificate of that intermediate CA in the list of trusted intermediate CA certs. For the locally configured MSP, this would mean that the certificate of this CA is removed from the intermediatecerts folder.
2. Reconfigure the MSP to include a CRL produced by the root of trust which denounces the mentioned intermediate CA’s certificate.

In the current MSP implementation we only support method (1) as it is simpler and does not require blacklisting the no longer considered intermediate CA.

**6) CAs and TLS CAs**

MSP identities’ root CAs and MSP TLS certificates’ root CAs (and relative intermediate CAs) need to be declared in different folders. This is to avoid confusion between different classes of certificates. It is not forbidden to reuse the same CAs for both MSP identities and TLS certificates but best practices suggest to avoid this in production.

# Channel Configuration (configtx)

Shared configuration for a Hyperledger Fabric blockchain network is stored in a collection configuration transactions, one per channel. Each configuration transaction is usually referred to by the shorter name configtx.

Channel configuration has the following important properties:

1. **Versioned**: All elements of the configuration have an associated version which is advanced with every modification. Further, every committed configuration receives a sequence [number](https://hyperledger-fabric.readthedocs.io/en/release-1.4/configtx.html).
2. **Permissioned**: Each element of the configuration has an associated policy which governs whether or not modification to that element is permitted. Anyone with a copy of the previous configtx (and no additional info) may verify the validity of a new config based on these policies.
3. **Hierarchical**: A root configuration group contains sub-groups, and each group of the hierarchy has associated values and policies. These policies can take advantage of the hierarchy to derive policies at one level from policies of lower levels.

## Anatomy of a configuration

Configuration is stored as a transaction of type HeaderType\_CONFIG in a block with no other transactions. These blocks are referred to as Configuration Blocks, the first of which is referred to as the Genesis Block.

The proto structures for configuration are stored in fabric/protos/common/configtx.proto. The Envelope of type HeaderType\_CONFIG encodes a ConfigEnvelope message as the Payload data field. The proto for ConfigEnvelope is defined as follows:

message ConfigEnvelope {

Config config **=** 1;

Envelope last\_update **=** 2;

}

The last\_update field is defined below in the **Updates to configuration** section, but is only necessary when validating the configuration, not reading it. Instead, the currently committed configuration is stored in the config field, containing a Config message.

message Config {

uint64 sequence **=** 1;

ConfigGroup channel\_group **=** 2;

}

The sequence number is incremented by one for each committed configuration. The channel\_group field is the root group which contains the configuration. The ConfigGroup structure is recursively defined, and builds a tree of groups, each of which contains values and policies. It is defined as follows:

message ConfigGroup {

uint64 version **=** 1;

map**<**string,ConfigGroup**>** groups **=** 2;

map**<**string,ConfigValue**>** values **=** 3;

map**<**string,ConfigPolicy**>** policies **=** 4;

string mod\_policy **=** 5;

}

Because ConfigGroup is a recursive structure, it has hierarchical arrangement. The following example is expressed for clarity in golang notation.

**//** Assume the following groups are defined

var root, child1, child2, grandChild1, grandChild2, grandChild3 **\***ConfigGroup

**//** Set the following values

root**.**Groups["child1"] **=** child1

root**.**Groups["child2"] **=** child2

child1**.**Groups["grandChild1"] **=** grandChild1

child2**.**Groups["grandChild2"] **=** grandChild2

child2**.**Groups["grandChild3"] **=** grandChild3

**//** The resulting config structure of groups looks like:

**//** root:

**//** child1:

**//** grandChild1

**//** child2:

**//** grandChild2

**//** grandChild3

Each group defines a level in the config hierarchy, and each group has an associated set of values (indexed by string key) and policies (also indexed by string key).

Values are defined by:

message ConfigValue {

uint64 version **=** 1;

bytes value **=** 2;

string mod\_policy **=** 3;

}

Policies are defined by:

message ConfigPolicy {

uint64 version **=** 1;

Policy policy **=** 2;

string mod\_policy **=** 3;

}

Note that Values, Policies, and Groups all have a version and a mod\_policy. The version of an element is incremented each time that element is modified. The mod\_policy is used to govern the required signatures to modify that element. For Groups, modification is adding or removing elements to the Values, Policies, or Groups maps (or changing the mod\_policy). For Values and Policies, modification is changing the Value and Policy fields respectively (or changing the mod\_policy). Each element’s mod\_policy is evaluated in the context of the current level of the config. Consider the following example mod policies defined at Channel.Groups["Application"] (Here, we use the golang map reference syntax, so Channel.Groups["Application"].Policies["policy1"] refers to the base Channel group’s Application group’s Policies map’s policy1 policy.)

* policy1 maps to Channel.Groups["Application"].Policies["policy1"]
* Org1/policy2 maps to Channel.Groups["Application"].Groups["Org1"].Policies["policy2"]
* /Channel/policy3 maps to Channel.Policies["policy3"]

Note that if a mod\_policy references a policy which does not exist, the item cannot be modified.

## Configuration updates

Configuration updates are submitted as an Envelope message of type HeaderType\_CONFIG\_UPDATE. The Payload data of the transaction is a marshaled ConfigUpdateEnvelope. The ConfigUpdateEnvelope is defined as follows:

message ConfigUpdateEnvelope {

bytes config\_update **=** 1;

repeated ConfigSignature signatures **=** 2;

}

The signatures field contains the set of signatures which authorizes the config update. Its message definition is:

message ConfigSignature {

bytes signature\_header **=** 1;

bytes signature **=** 2;

}

The signature\_header is as defined for standard transactions, while the signature is over the concatenation of the signature\_header bytes and the config\_update bytes from the ConfigUpdateEnvelope message.

The ConfigUpdateEnvelope config\_update bytes are a marshaled ConfigUpdate message which is defined as follows:

message ConfigUpdate {

string channel\_id **=** 1;

ConfigGroup read\_set **=** 2;

ConfigGroup write\_set **=** 3;

}

The channel\_id is the channel ID the update is bound for, this is necessary to scope the signatures which support this reconfiguration.

The read\_set specifies a subset of the existing configuration, specified sparsely where only the version field is set and no other fields must be populated. The particular ConfigValue value or ConfigPolicy policy fields should never be set in the read\_set. The ConfigGroup may have a subset of its map fields populated, so as to reference an element deeper in the config tree. For instance, to include the Application group in the read\_set, its parent (the Channel group) must also be included in the read set, but, the Channel group does not need to populate all of the keys, such as the Orderer group key, or any of the values or policies keys.

The write\_set specifies the pieces of configuration which are modified. Because of the hierarchical nature of the configuration, a write to an element deep in the hierarchy must contain the higher level elements in its write\_set as well. However, for any element in the write\_set which is also specified in the read\_set at the same version, the element should be specified sparsely, just as in the read\_set.

For example, given the configuration:

Channel: (version 0)

Orderer (version 0)

Application (version 3)

Org1 (version 2)

To submit a configuration update which modifies Org1, the read\_set would be:

Channel: (version 0)

Application: (version 3)

and the write\_set would be

Channel: (version 0)

Application: (version 3)

Org1 (version 3)

When the CONFIG\_UPDATE is received, the orderer computes the resulting CONFIG by doing the following:

1. Verifies the channel\_id and read\_set. All elements in the read\_set must exist at the given versions.
2. Computes the update set by collecting all elements in the write\_set which do not appear at the same version in the read\_set.
3. Verifies that each element in the update set increments the version number of the element update by exactly 1.
4. Verifies that the signature set attached to the ConfigUpdateEnvelope satisfies the mod\_policy for each element in the update set.
5. Computes a new complete version of the config by applying the update set to the current config.
6. Writes the new config into a ConfigEnvelope which includes the CONFIG\_UPDATE as the last\_update field and the new config encoded in the config field, along with the incremented sequence value.
7. Writes the new ConfigEnvelope into a Envelope of type CONFIG, and ultimately writes this as the sole transaction in a new configuration block.

When the peer (or any other receiver for Deliver) receives this configuration block, it should verify that the config was appropriately validated by applying the last\_update message to the current config and verifying that the orderer-computed config field contains the correct new configuration.

## Permitted configuration groups and values

Any valid configuration is a subset of the following configuration. Here we use the notation peer.<MSG> to define a ConfigValue whose value field is a marshaled proto message of name <MSG> defined in fabric/protos/peer/configuration.proto. The notations common.<MSG>, msp.<MSG>, and orderer.<MSG> correspond similarly, but with their messages defined in fabric/protos/common/configuration.proto, fabric/protos/msp/mspconfig.proto, and fabric/protos/orderer/configuration.proto respectively.

Note, that the keys {{org\_name}} and {{consortium\_name}} represent arbitrary names, and indicate an element which may be repeated with different names.

**&**ConfigGroup{

Groups: map**<**string, **\***ConfigGroup**>** {

"Application":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

"AnchorPeers":peer**.**AnchorPeers,

},

},

},

},

"Orderer":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

},

},

},

Values:map**<**string, **\***ConfigValue**>** {

"ConsensusType":orderer**.**ConsensusType,

"BatchSize":orderer**.**BatchSize,

"BatchTimeout":orderer**.**BatchTimeout,

"KafkaBrokers":orderer**.**KafkaBrokers,

},

},

"Consortiums":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{consortium\_name}}:**&**ConfigGroup{

Groups:map**<**string, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

},

},

},

Values:map**<**string, **\***ConfigValue**>** {

"ChannelCreationPolicy":common**.**Policy,

}

},

},

},

},

Values: map**<**string, **\***ConfigValue**>** {

"HashingAlgorithm":common**.**HashingAlgorithm,

"BlockHashingDataStructure":common**.**BlockDataHashingStructure,

"Consortium":common**.**Consortium,

"OrdererAddresses":common**.**OrdererAddresses,

},

}

## Orderer system channel configuration

The ordering system channel needs to define ordering parameters, and consortiums for creating channels. There must be exactly one ordering system channel for an ordering service, and it is the first channel to be created (or more accurately bootstrapped). It is recommended never to define an Application section inside of the ordering system channel genesis configuration, but may be done for testing. Note that any member with read access to the ordering system channel may see all channel creations, so this channel’s access should be restricted.

The ordering parameters are defined as the following subset of config:

**&**ConfigGroup{

Groups: map**<**string, **\***ConfigGroup**>** {

"Orderer":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

},

},

},

Values:map**<**string, **\***ConfigValue**>** {

"ConsensusType":orderer**.**ConsensusType,

"BatchSize":orderer**.**BatchSize,

"BatchTimeout":orderer**.**BatchTimeout,

"KafkaBrokers":orderer**.**KafkaBrokers,

},

},

},

Each organization participating in ordering has a group element under the Orderer group. This group defines a single parameter MSP which contains the cryptographic identity information for that organization. The Values of the Orderer group determine how the ordering nodes function. They exist per channel, so orderer.BatchTimeout for instance may be specified differently on one channel than another.

At startup, the orderer is faced with a filesystem which contains information for many channels. The orderer identifies the system channel by identifying the channel with the consortiums group defined. The consortiums group has the following structure.

**&**ConfigGroup{

Groups: map**<**string, **\***ConfigGroup**>** {

"Consortiums":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{consortium\_name}}:**&**ConfigGroup{

Groups:map**<**string, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

},

},

},

Values:map**<**string, **\***ConfigValue**>** {

"ChannelCreationPolicy":common**.**Policy,

}

},

},

},

},

},

Note that each consortium defines a set of members, just like the organizational members for the ordering orgs. Each consortium also defines a ChannelCreationPolicy. This is a policy which is applied to authorize channel creation requests. Typically, this value will be set to an ImplicitMetaPolicy requiring that the new members of the channel sign to authorize the channel creation. More details about channel creation follow later in this document.

## Application channel configuration

Application configuration is for channels which are designed for application type transactions. It is defined as follows:

**&**ConfigGroup{

Groups: map**<**string, **\***ConfigGroup**>** {

"Application":**&**ConfigGroup{

Groups:map**<**String, **\***ConfigGroup**>** {

{{org\_name}}:**&**ConfigGroup{

Values:map**<**string, **\***ConfigValue**>**{

"MSP":msp**.**MSPConfig,

"AnchorPeers":peer**.**AnchorPeers,

},

},

},

},

},

}

Just like with the Orderer section, each organization is encoded as a group. However, instead of only encoding the MSP identity information, each org additionally encodes a list of AnchorPeers. This list allows the peers of different organizations to contact each other for peer gossip networking.

The application channel encodes a copy of the orderer orgs and consensus options to allow for deterministic updating of these parameters, so the same Orderer section from the orderer system channel configuration is included. However from an application perspective this may be largely ignored.

## Channel creation

When the orderer receives a CONFIG\_UPDATE for a channel which does not exist, the orderer assumes that this must be a channel creation request and performs the following.

1. The orderer identifies the consortium which the channel creation request is to be performed for. It does this by looking at the Consortium value of the top level group.
2. The orderer verifies that the organizations included in the Application group are a subset of the organizations included in the corresponding consortium and that the ApplicationGroup is set to version 1.
3. The orderer verifies that if the consortium has members, that the new channel also has application members (creation consortiums and channels with no members is useful for testing only).
4. The orderer creates a template configuration by taking the Orderer group from the ordering system channel, and creating an Application group with the newly specified members and specifying its mod\_policy to be the ChannelCreationPolicy as specified in the consortium config. Note that the policy is evaluated in the context of the new configuration, so a policy requiring ALL members, would require signatures from all the new channel members, not all the members of the consortium.
5. The orderer then applies the CONFIG\_UPDATE as an update to this template configuration. Because the CONFIG\_UPDATE applies modifications to the Application group (its version is 1), the config code validates these updates against the ChannelCreationPolicy. If the channel creation contains any other modifications, such as to an individual org’s anchor peers, the corresponding mod policy for the element will be invoked.
6. The new CONFIG transaction with the new channel config is wrapped and sent for ordering on the ordering system channel. After ordering, the channel is created.

# Defining capability requirements

As discussed in [Channel capabilities](https://hyperledger-fabric.readthedocs.io/en/release-1.4/capabilities_concept.html), capability requirements are defined per channel in the channel configuration (found in the channel’s most recent configuration block). The channel configuration contains three locations, each of which defines a capability of a different type.

| **Capability Type** | **Canonical Path** | **JSON Path** |
| --- | --- | --- |
| Channel | /Channel/Capabilities | .channel\_group.values.Capabilities |
| Orderer | /Channel/Orderer/Capabilities | .channel\_group.groups.Orderer.values.Capabilities |
| Application | /Channel/Application/Capabilities | .channel\_group.groups.Application.values. Capabilities |

## Setting Capabilities

Capabilities are set as part of the channel configuration (either as part of the initial configuration – which we’ll talk about in a moment – or as part of a reconfiguration).

**Note**

For a tutorial that shows how to update a channel configuration, check out [Adding an Org to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html). For an overview of the different kinds of channel updates that are possible, check out [Updating a Channel Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html).

Because new channels copy the configuration of the ordering system channel by default, new channels will automatically be configured to work with the orderer and channel capabilities of the ordering system channel and the application capabilities specified by the channel creation transaction. Channels that already exist, however, must be reconfigured.

The schema for the Capabilities value is defined in the protobuf as:

message Capabilities {

map**<**string, Capability**>** capabilities **=** 1;

}

message Capability { }

As an example, rendered in JSON:

{

"capabilities": {

"V1\_1": {}

}

}

### Capabilities in an Initial Configuration

In the configtx.yaml file distributed in the config directory of the release artifacts, there is a Capabilities section which enumerates the possible capabilities for each capability type (Channel, Orderer, and Application).

The simplest way to enable capabilities is to pick a v1.1 sample profile and customize it for your network. For example:

SampleSingleMSPSoloV1\_1:

Capabilities:

**<<**: **\***GlobalCapabilities

Orderer:

**<<**: **\***OrdererDefaults

Organizations:

**-** **\***SampleOrg

Capabilities:

**<<**: **\***OrdererCapabilities

Consortiums:

SampleConsortium:

Organizations:

**-** **\***SampleOrg

Note that there is a Capabilities section defined at the root level (for the channel capabilities), and at the Orderer level (for orderer capabilities). The sample above uses a YAML reference to include the capabilities as defined at the bottom of the YAML.

When defining the orderer system channel there is no Application section, as those capabilities are defined during the creation of an application channel. To define a new channel’s application capabilities at channel creation time, the application admins should model their channel creation transaction after the SampleSingleMSPChannelV1\_1 profile.

SampleSingleMSPChannelV1\_1:

Consortium: SampleConsortium

Application:

Organizations:

**-** **\***SampleOrg

Capabilities:

**<<**: **\***ApplicationCapabilities

Here, the Application section has a new element Capabilities which references the ApplicationCapabilities section defined at the end of the YAML.

**Note**

The capabilities for the Channel and Orderer sections are inherited from the definition in the ordering system channel and are automatically included by the orderer during the process of channel creation.

# Endorsement policies

Every chaincode has an endorsement policy which specifies the set of peers on a channel that must execute chaincode and endorse the execution results in order for the transaction to be considered valid. These endorsement policies define the organizations (through their peers) who must “endorse” (i.e., approve of) the execution of a proposal.

**Note**

Recall that **state**, represented by key-value pairs, is separate from blockchain data. For more on this, check out our [Ledger](https://hyperledger-fabric.readthedocs.io/en/release-1.4/ledger/ledger.html) documentation.

As part of the transaction validation step performed by the peers, each validating peer checks to make sure that the transaction contains the appropriate **number** of endorsements and that they are from the expected sources (both of these are specified in the endorsement policy). The endorsements are also checked to make sure they’re valid (i.e., that they are valid signatures from valid certificates).

## Two ways to require endorsement

By default, endorsement policies are specified for a channel’s chaincode at instantiation or upgrade time (that is, one endorsement policy covers all of the state associated with a chaincode).

However, there are cases where it may be necessary for a particular state (a particular key-value pair, in other words) to have a different endorsement policy. This **state-based endorsement** allows the default chaincode-level endorsement policies to be overridden by a different policy for the specified keys.

To illustrate the circumstances in which these two types of endorsement policies might be used, consider a channel on which [cars](https://hyperledger-fabric.readthedocs.io/en/release-1.4/endorsement-policies.html) are being exchanged. The “creation” — also known as “issuance” – of a car as an asset that can be traded (putting the key-value pair that represents it into the world state, in other words) would have to satisfy the chaincode-level endorsement policy. To see how to set a chaincode-level endorsement policy, check out the section below.

If the car requires a specific endorsement policy, it can be defined either when the car is created or afterwards. There are a number of reasons why it might be necessary or preferable to set a state-specific endorsement policy. The car might have historical importance or value that makes it necessary to have the endorsement of a licensed appraiser. Also, the owner of the car (if they’re a member of the channel) might also want to ensure that their peer signs off on a transaction. In both cases, **an endorsement policy is required for a particular asset that is different from the default endorsement policies for the other assets associated with that chaincode.**

We’ll show you how to define a state-based endorsement policy in a subsequent section. But first, let’s see how we set a chaincode-level endorsement policy.

## Setting chaincode-level endorsement policies

Chaincode-level endorsement policies can be specified at instantiate time using either the SDK (for some sample code on how to do this, [click](https://hyperledger-fabric.readthedocs.io/en/release-1.4/endorsement-policies.html) [here](https://github.com/hyperledger/fabric-sdk-node/blob/f8ffa90dc1b61a4a60a6fa25de760c647587b788/test/integration/e2e/e2eUtils.js#L178)) or in the peer CLI using the -P switch followed by the policy.

**Note**

Don’t worry about the policy syntax ('Org1.member', et all) right now. We’ll talk more about the syntax in the next section.

For example:

peer chaincode instantiate **-**C **<**channelid**>** **-**n mycc **-**P "AND('Org1.member', 'Org2.member')"

This command deploys chaincode mycc (“my chaincode”) with the policy AND('Org1.member', 'Org2.member') which would require that a member of both Org1 and Org2 [sign](https://hyperledger-fabric.readthedocs.io/en/release-1.4/endorsement-policies.html) the transaction.

Notice that, if the identity classification is enabled (see [Membership Service Providers (MSP)](https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html)), one can use the PEER role to restrict endorsement to only peers.

For example:

peer chaincode instantiate **-**C **<**channelid**>** **-**n mycc **-**P "AND('Org1.peer', 'Org2.peer')"

A new organization added to the channel after instantiation can query a chaincode (provided the query has appropriate authorization as defined by channel policies and any application level checks enforced by the chaincode) but will not be able to execute or endorse the chaincode. The endorsement policy needs to be modified to allow transactions to be committed with endorsements from the new organization.

**Note**

if not specified at instantiation time, the endorsement policy defaults to “any member of the organizations in the channel”. For example, a channel with “Org1” and “Org2” would have a default endorsement policy of “OR(‘Org1.member’, ‘Org2.member’)”.

### Endorsement policy syntax

As you can see above, policies are expressed in terms of principals (“principals” are identities matched to a role). Principals are described as 'MSP.ROLE', where MSP represents the required MSP ID and ROLE represents one of the four accepted roles: member, admin, client, and peer.

Here are a few examples of valid principals:

* 'Org0.admin': any administrator of the Org0 MSP
* 'Org1.member': any member of the Org1 MSP
* 'Org1.client': any client of the Org1 MSP
* 'Org1.peer': any peer of the Org1 MSP

The syntax of the language is:

EXPR(E[, E...])

Where EXPR is either AND, OR, or OutOf, and E is either a principal (with the syntax described above) or another nested call to EXPR.

**For example:**

* AND('Org1.member', 'Org2.member', 'Org3.member') requests one signature from each of the three principals.
* OR('Org1.member', 'Org2.member') requests one signature from either one of the two principals.
* OR('Org1.member', AND('Org2.member', 'Org3.member')) requests either one signature from a member of the Org1 MSP or one signature from a member of the Org2 MSP and one signature from a member of the Org3 MSP.
* OutOf(1, 'Org1.member', 'Org2.member'), which resolves to the same thing as OR('Org1.member', 'Org2.member').
* Similarly, OutOf(2, 'Org1.member', 'Org2.member') is equivalent to AND('Org1.member', 'Org2.member'), and OutOf(2, 'Org1.member', 'Org2.member', 'Org3.member') is equivalent to OR(AND('Org1.member', 'Org2.member'), AND('Org1.member', 'Org3.member'), AND('Org2.member', 'Org3.member')).

## Setting key-level endorsement policies

Setting regular chaincode-level endorsement policies is tied to the lifecycle of the corresponding chaincode. They can only be set or modified when instantiating or upgrading the corresponding chaincode on a channel.

In contrast, key-level endorsement policies can be set and modified in a more granular fashion from within a chaincode. The modification is part of the read-write set of a regular transaction.

The shim API provides the following functions to set and retrieve an endorsement policy for/from a regular key.

**Note**

ep below stands for the “endorsement policy”, which can be expressed either by using the same syntax described above or by using the convenience function described below. Either method will generate a binary version of the endorsement policy that can be consumed by the basic shim API.

SetStateValidationParameter(key **string**, ep []**byte**) **error**

GetStateValidationParameter(key **string**) ([]**byte**, **error**)

For keys that are part of [Private data](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html) in a collection the following functions apply:

SetPrivateDataValidationParameter(collection, key **string**, ep []**byte**) **error**

GetPrivateDataValidationParameter(collection, key **string**) ([]**byte**, **error**)

To help set endorsement policies and marshal them into validation parameter byte arrays, the Go shim provides an extension with convenience functions that allow the chaincode developer to deal with endorsement policies in terms of the MSP identifiers of organizations, see [KeyEndorsementPolicy](https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim/ext/statebased#KeyEndorsementPolicy):

**type** KeyEndorsementPolicy **interface** {

*// Policy returns the endorsement policy as bytes*

Policy() ([]**byte**, **error**)

*// AddOrgs adds the specified orgs to the list of orgs that are required*

*// to endorse*

AddOrgs(roleType RoleType, organizations **...string**) **error**

*// DelOrgs delete the specified channel orgs from the existing key-level endorsement*

*// policy for this KVS key. If any org is not present, an error will be returned.*

DelOrgs(organizations **...string**) **error**

*// ListOrgs returns an array of channel orgs that are required to endorse changes*

ListOrgs() ([]**string**)

}

For example, to set an endorsement policy for a key where two specific orgs are required to endorse the key change, pass both org MSPIDs to AddOrgs(), and then call Policy() to construct the endorsement policy byte array that can be passed to SetStateValidationParameter().

To add the shim extension to your chaincode as a dependency, see [Managing external dependencies for chaincode written in Go](https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html#vendoring).

## Validation

At commit time, setting a value of a key is no different from setting the endorsement policy of a key — both update the state of the key and are validated based on the same rules.

| **Validation** | **no validation parameter set** | **validation parameter set** |
| --- | --- | --- |
| modify value | check chaincode ep | check key-level ep |
| modify key-level ep | check chaincode ep | check key-level ep |

As we discussed above, if a key is modified and no key-level endorsement policy is present, the chaincode-level endorsement policy applies by default. This is also true when a key-level endorsement policy is set for a key for the first time — the new key-level endorsement policy must first be endorsed according to the pre-existing chaincode-level endorsement policy.

If a key is modified and a key-level endorsement policy is present, the key-level endorsement policy overrides the chaincode-level endorsement policy. In practice, this means that the key-level endorsement policy can be either less restrictive or more restrictive than the chaincode-level endorsement policy. Because the chaincode-level endorsement policy must be satisfied in order to set a key-level endorsement policy for the first time, no trust assumptions have been violated.

If a key’s endorsement policy is removed (set to nil), the chaincode-level endorsement policy becomes the default again.

If a transaction modifies multiple keys with different associated key-level endorsement policies, all of these policies need to be satisfied in order for the transaction to be valid.

# Pluggable transaction endorsement and validation

## Motivation

When a transaction is validated at time of commit, the peer performs various [checks](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html) before [applying](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html) the state changes that come with the transaction itself:

* Validating the identities that signed the transaction.
* Verifying the signatures of the endorsers on the transaction.
* Ensuring the transaction satisfies the endorsement policies of the namespaces of the corresponding chaincodes.

There are use cases which demand custom transaction validation rules different from the default Fabric validation rules, such as:

* **UTXO (Unspent Transaction Output):** When the validation takes into [account](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html) whether the transaction doesn’t double spend its inputs.
* **Anonymous transactions:** When the endorsement doesn’t contain the identity of the peer, but a signature and a public key are shared that can’t be linked to the peer’s identity.

## Pluggable endorsement and validation logic

Fabric allows for the implementation and deployment of custom endorsement and validation logic into the peer to be associated with chaincode handling in a pluggable manner. This logic can be either compiled into the peer as built in selectable logic, or compiled and deployed alongside the peer as a [Golang plugin](https://golang.org/pkg/plugin/).

Recall that every chaincode is associated with its own endorsement and validation logic at the time of chaincode instantiation. If the [user](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html) doesn’t select one, the default built-in logic is selected implicitly. A peer administrator may alter the endorsement/validation logic that is selected by extending the peer’s local configuration with the customization of the endorsement/validation logic which is loaded and applied at peer startup.

## Configuration

Each peer has a local configuration (core.yaml) that declares a mapping between the endorsement/validation logic name and the implementation that is to be run.

The default logic are called ESCC (with the “E” standing for endorsement) and VSCC (validation), and they can be found in the peer local configuration in the handlers section:

handlers:

endorsers:

escc:

name: DefaultEndorsement

validators:

vscc:

name: DefaultValidation

When the endorsement or validation implementation is compiled into the peer, the name property represents the initialization function that is to be run in order to obtain the factory that creates instances of the endorsement/validation logic.

The function is an instance method of the HandlerLibrary construct under core/handlers/library/library.go and in order for custom endorsement or validation logic to be added, this construct needs to be extended with any additional methods.

Since this is cumbersome and poses a deployment challenge, one can also deploy custom endorsement and validation as a Golang plugin by adding another property under the name called library.

For example, if we have custom endorsement and validation logic which is implemented as a plugin, we would have the following entries in the configuration in core.yaml:

handlers:

endorsers:

escc:

name: DefaultEndorsement

custom:

name: customEndorsement

library: /etc/hyperledger/fabric/plugins/customEndorsement.so

validators:

vscc:

name: DefaultValidation

custom:

name: customValidation

library: /etc/hyperledger/fabric/plugins/customValidation.so

And we’d have to place the .so plugin files in the peer’s local file system.

**Note**

Hereafter, custom endorsement or validation logic implementation is going to be referred to as “plugins”, even if they are compiled into the peer.

## Endorsement plugin implementation

To implement an endorsement plugin, one must implement the Plugin interface found in core/handlers/endorsement/api/endorsement.go:

*// Plugin endorses a proposal response*

**type** Plugin **interface** {

*// Endorse signs the given payload(ProposalResponsePayload bytes), and optionally mutates it.*

*// Returns:*

*// The Endorsement: A signature over the payload, and an identity that is used to verify the signature*

*// The payload that was given as input (could be modified within this function)*

*// Or error on failure*

Endorse(payload []**byte**, sp **\***peer.SignedProposal) (**\***peer.Endorsement, []**byte**, **error**)

*// Init injects dependencies into the instance of the Plugin*

Init(dependencies **...**Dependency) **error**

}

An endorsement plugin instance of a given plugin type (identified either by the method name as an instance method of the HandlerLibrary or by the plugin .so file path) is created for each channel by having the peer invoke the New method in the PluginFactory interface which is also expected to be implemented by the plugin developer:

*// PluginFactory creates a new instance of a Plugin*

**type** PluginFactory **interface** {

New() Plugin

}

The Init method is expected to receive as input all the dependencies declared under core/handlers/endorsement/api/, identified as embedding the Dependency interface.

After the creation of the Plugin instance, the Init method is invoked on it by the peer with the dependencies passed as parameters.

Currently Fabric comes with the following dependencies for endorsement plugins:

* SigningIdentityFetcher: Returns an instance of SigningIdentity based on a given signed proposal:

*// SigningIdentity signs messages and serializes its public identity to bytes*

**type** SigningIdentity **interface** {

*// Serialize returns a byte representation of this identity which is used to verify*

*// messages signed by this SigningIdentity*

Serialize() ([]**byte**, **error**)

*//* [*Sign*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/pluggable_endorsement_and_validation.html) *signs the given payload and returns a signature*

Sign([]**byte**) ([]**byte**, **error**)

}

* StateFetcher: Fetches a **State** object which interacts with the world state:

*// State defines interaction with the world state*

**type** State **interface** {

*// GetPrivateDataMultipleKeys gets the values for the multiple private data items in a single call*

GetPrivateDataMultipleKeys(namespace, collection **string**, keys []**string**) ([][]**byte**, **error**)

*// GetStateMultipleKeys gets the values for multiple keys in a single call*

GetStateMultipleKeys(namespace **string**, keys []**string**) ([][]**byte**, **error**)

*// GetTransientByTXID gets the values private data associated with the given txID*

GetTransientByTXID(txID **string**) ([]**\***rwset.TxPvtReadWriteSet, **error**)

*// Done releases resources occupied by the State*

Done()

}

## Validation plugin implementation

To implement a validation plugin, one must implement the Plugin interface found in core/handlers/validation/api/validation.go:

*// Plugin validates transactions*

**type** Plugin **interface** {

*// Validate returns nil if the action at the given position inside the transaction*

*// at the given position in the given block is valid, or an error if not.*

Validate(block **\***common.Block, namespace **string**, txPosition **int**, actionPosition **int**, contextData **...**ContextDatum) **error**

*// Init injects dependencies into the instance of the Plugin*

Init(dependencies **...**Dependency) **error**

}

Each ContextDatum is additional runtime-derived metadata that is passed by the peer to the validation plugin. Currently, the only ContextDatum that is passed is one that represents the endorsement policy of the chaincode:

*// SerializedPolicy defines a serialized policy*

**type** SerializedPolicy **interface** {

validation.ContextDatum

*// Bytes returns the bytes of the SerializedPolicy*

Bytes() []**byte**

}

A validation plugin instance of a given plugin type (identified either by the method name as an instance method of the HandlerLibrary or by the plugin .so file path) is created for each channel by having the peer invoke the New method in the PluginFactory interface which is also expected to be implemented by the plugin developer:

*// PluginFactory creates a new instance of a Plugin*

**type** PluginFactory **interface** {

New() Plugin

}

The Init method is expected to receive as input all the dependencies declared under core/handlers/validation/api/, identified as embedding the Dependency interface.

After the creation of the Plugin instance, the **Init** method is invoked on it by the peer with the dependencies passed as parameters.

Currently Fabric comes with the following dependencies for validation plugins:

* IdentityDeserializer: Converts byte representation of identities into Identity objects that can be used to verify signatures signed by them, be validated themselves against their corresponding MSP, and see whether they satisfy a given **MSP Principal**. The full specification can be found in core/handlers/validation/api/identities/identities.go.
* PolicyEvaluator: Evaluates whether a given policy is satisfied:

*// PolicyEvaluator evaluates policies*

**type** PolicyEvaluator **interface** {

validation.Dependency

*// Evaluate takes a set of SignedData and evaluates whether this set of signatures satisfies*

*// the policy with the given bytes*

Evaluate(policyBytes []**byte**, signatureSet []**\***common.SignedData) **error**

}

* StateFetcher: Fetches a State object which interacts with the world state:

*// State defines interaction with the world state*

**type** State **interface** {

*// GetStateMultipleKeys gets the values for multiple keys in a single call*

GetStateMultipleKeys(namespace **string**, keys []**string**) ([][]**byte**, **error**)

*// GetStateRangeScanIterator returns an iterator that contains all the key-values between given key ranges.*

*// startKey is included in the results and endKey is excluded. An empty startKey refers to the first available key*

*// and an empty endKey refers to the last available key. For scanning all the keys, both the startKey and the endKey*

*// can be supplied as empty strings. However, a full scan should be used judiciously for performance reasons.*

*// The returned ResultsIterator contains results of type \*KV which is defined in protos/ledger/queryresult.*

GetStateRangeScanIterator(namespace **string**, startKey **string**, endKey **string**) (ResultsIterator, **error**)

*// GetStateMetadata returns the metadata for given namespace and key*

GetStateMetadata(namespace, key **string**) (**map**[**string**][]**byte**, **error**)

*// GetPrivateDataMetadata gets the metadata of a private data item identified by a tuple <namespace, collection, key>*

GetPrivateDataMetadata(namespace, collection, key **string**) (**map**[**string**][]**byte**, **error**)

*// Done releases resources occupied by the State*

Done()

}

## Important notes

* **Validation plugin consistency across peers:** In future releases, the Fabric channel infrastructure would guarantee that the same validation logic is used for a given chaincode by all peers in the channel at any given blockchain height in order to eliminate the chance of mis-configuration which would might lead to state divergence among peers that accidentally run different implementations. However, for now it is the sole responsibility of the system operators and administrators to ensure this doesn’t happen.
* **Validation plugin error handling:** Whenever a validation plugin can’t determine whether a given transaction is valid or not, because of some transient execution problem like inability to access the database, it should return an error of type **ExecutionFailureError** that is defined in core/handlers/validation/api/validation.go. Any other error that is returned, is treated as an endorsement policy error and marks the transaction as invalidated by the validation logic. However, if an ExecutionFailureError is returned, the chain processing halts instead of marking the transaction as invalid. This is to prevent state divergence between different peers.
* **Error handling for private metadata retrieval**: In case a plugin retrieves metadata for private data by making use of the StateFetcher interface, it is important that errors are handled as follows: CollConfigNotDefinedError'' and ``InvalidCollNameError'', signalling that the specified collection does not exist, should be handled as deterministic errors and should not lead the plugin to return an ``ExecutionFailureError.
* **Importing Fabric code into the plugin**: Importing code that belongs to Fabric other than protobufs as part of the plugin is highly discouraged, and can lead to issues when the Fabric code changes between releases, or can cause inoperability issues when running mixed peer versions. Ideally, the plugin code should only use the dependencies given to it, and should import the bare minimum other than protobufs.

# Access Control Lists (ACL)

## What is an Access Control List?

Note: This topic deals with access control and policies on a channel administration *level*. To learn about access control within a chaincode, check out our [*chaincode for developers tutorial*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html#Chaincode_API).

Fabric uses access control lists (ACLs) to manage access to resources by associating a **policy** — which specifies a rule that evaluates to true or false, given a set of identities — with the resource. Fabric contains a [number](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html) of default ACLs. In this [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html), we’ll talk about how they’re formatted and how the defaults can be overridden.

But before we can do that, it’s necessary to understand a little about resources and policies.

### Resources

Users interact with Fabric by targeting a [user chaincode](https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html), [system chaincode](https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4noah.html), or an [events stream source](https://hyperledger-fabric.readthedocs.io/en/release-1.4/peer_event_services.html). As such, these endpoints are considered “resources” on which access control should be exercised.

Application developers need to be aware of these resources and the default policies associated with them. The complete list of these resources are found in configtx.yaml. You can look at a [sample configtx.yaml file here](http://github.com/hyperledger/fabric/blob/release-1.2/sampleconfig/configtx.yaml).

The resources named in configtx.yaml is an exhaustive list of all internal resources currently defined by Fabric. The loose convention adopted there is <component>/<resource>. So cscc/GetConfigBlock is the resource for the GetConfigBlock call in the CSCC component.

### Policies

Policies are fundamental to the way Fabric works because they allow the identity (or set of identities) associated with a request to be checked against the policy associated with the resource needed to fulfill the request. Endorsement policies are used to determine whether a transaction has been appropriately endorsed. The policies defined in the channel configuration are referenced as modification policies as well as for access control, and are defined in the channel configuration itself.

Policies can be structured in one of two ways: as Signature policies or as an ImplicitMeta policy.

#### Signature policies

These policies identify specific users who must [sign in](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html) order for a policy to be satisfied. For example:

Policies:

MyPolicy:

Type: Signature

Rule: “Org1.Peer OR Org2.Peer”

This policy construct can be interpreted as: the policy named *MyPolicy* can only be satisfied by the signature of an identity with role of “a peer from Org1” or “a peer from Org2”.

Signature policies support arbitrary combinations of AND, OR, and NOutOf, allowing the construction of extremely powerful rules like: “An admin of org A and two other admins, or 11 of 20 org admins”.

#### ImplicitMeta policies

ImplicitMeta policies aggregate the result of policies deeper in the configuration hierarchy that are ultimately defined by Signature policies. They support default rules like “A majority of the organization admins”. These policies use a different but still very simple syntax as compared to Signature policies: <ALL|ANY|MAJORITY> <sub\_policy>.

For example: ANY Readers or MAJORITY Admins.

Note that in the default policy configuration *Admins* have an operational role. Policies that specify that only Admins — or some subset of Admins — have access to a resource will tend to be for sensitive or operational aspects of the network (such as instantiating chaincode on a channel). *Writers* will tend to be able to propose ledger updates, such as a transaction, but will not typically have administrative permissions. *Readers* have a passive role. They can access information but do not have the permission to propose ledger updates nor do can they perform administrative tasks. These default policies can be added to, edited, or supplemented, for example by the new *peer* and *client* roles (if you have *NodeOU* support).

Here’s an example of an ImplicitMeta policy structure:

Policies:

AnotherPolicy:

Type: ImplicitMeta

Rule: "MAJORITY Admins"

Here, the policy AnotherPolicy can be satisfied by the MAJORITY of Admins, where Admins is eventually being specified by lower level Signature policy.

### Where is access control specified?

Access control defaults exist inside configtx.yaml, the file that configtxgen uses to build channel configurations.

Access control can be updated one of two ways, either by editing configtx.yaml itself, which will propagate the ACL change to any new channels, or by updating access control in the channel configuration of a particular channel.

## How ACLs are formatted in configtx.yaml

ACLs are formatted as a key-value pair consisting of a resource function name followed by a string. To see what this looks like, reference this [sample configtx.yaml file](https://github.com/hyperledger/fabric/blob/release-1.2/sampleconfig/configtx.yaml).

Two excerpts from this sample:

*# ACL policy for invoking chaincodes on peer*

peer**/**Propose: **/**Channel**/**Application**/**Writers

*# ACL policy for sending block events*

event**/**Block: **/**Channel**/**Application**/**Readers

These ACLs define that access to peer/Propose and event/Block resources is restricted to identities satisfying the policy defined at the canonical path /Channel/Application/Writers and /Channel/Application/Readers, respectively.

### Updating ACL defaults in configtx.yaml

In cases where it will be necessary to override ACL defaults when bootstrapping a network, or to change the ACLs before a channel has been bootstrapped, the best practice will be to update configtx.yaml.

Let’s say you want to modify the peer/Propose ACL default — which specifies the policy for invoking chaincodes on a peer – from /Channel/Application/Writers to a policy called MyPolicy.

This is done by adding a policy called MyPolicy (it could be called anything, but for this example we’ll call it MyPolicy). The policy is defined in the Application.Policies section inside configtx.yaml and specifies a rule to be checked to grant or deny access to a [user](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html). For this example, we’ll be creating a Signature policy identifying SampleOrg.admin.

Policies: **&**ApplicationDefaultPolicies

Readers:

Type: ImplicitMeta

Rule: "ANY Readers"

Writers:

Type: ImplicitMeta

Rule: "ANY Writers"

Admins:

Type: ImplicitMeta

Rule: "MAJORITY Admins"

MyPolicy:

Type: Signature

Rule: "OR('SampleOrg.admin')"

Then, edit the Application: ACLs section inside configtx.yaml to change peer/Propose from this:

peer/Propose: /Channel/Application/Writers

To this:

peer/Propose: /Channel/Application/MyPolicy

Once these fields have been changed in configtx.yaml, the configtxgen tool will use the policies and ACLs defined when creating a channel creation transaction. When appropriately signed and submitted by one of the admins of the consortium members, a new channel with the defined ACLs and policies is created.

Once MyPolicy has been bootstrapped into the channel configuration, it can also be referenced to override other ACL defaults. For example:

SampleSingleMSPChannel:

Consortium: SampleConsortium

Application:

**<<**: **\***ApplicationDefaults

ACLs:

**<<**: **\***ACLsDefault

event**/**Block: **/**Channel**/**Application**/**MyPolicy

This would restrict the ability to subscribe to block events to SampleOrg.admin.

If channels have already been created that want to use this ACL, they’ll have to update their channel configurations one at a time using the following flow:

### Updating ACL defaults in the channel config

If channels have already been created that want to use MyPolicy to restrict access to peer/Propose — or if they want to [create](https://hyperledger-fabric.readthedocs.io/en/release-1.4/access_control.html) ACLs they don’t want other channels to know about — they’ll have to update their channel configurations one at a time through config update transactions.

Note: Channel configuration transactions are an involved process we won’t delve into here. If you want to read more about them check out our document on [*channel configuration updates*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/config_update.html) and our [*“Adding an Org to a Channel” tutorial*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html).

After pulling, translating, and stripping the configuration block of its metadata, you would edit the configuration by adding MyPolicy under Application: policies, where the Admins, Writers, and Readers policies already live.

"MyPolicy": {

"mod\_policy": "Admins",

"policy": {

"type": 1,

"value": {

"identities": [

{

"principal": {

"msp\_identifier": "SampleOrg",

"role": "ADMIN"

},

"principal\_classification": "ROLE"

}

],

"rule": {

"n\_out\_of": {

"n": 1,

"rules": [

{

"signed\_by": 0

}

]

}

},

"version": 0

}

},

"version": "0"

},

Note in particular the msp\_identifer and role here.

Then, in the ACLs section of the config, change the peer/Propose ACL from this:

"peer/Propose": {

"policy\_ref": "/Channel/Application/Writers"

To this:

"peer/Propose": {

"policy\_ref": "/Channel/Application/MyPolicy"

Note: If you do not have ACLs defined in your channel configuration, you will have to add the entire ACL structure.

Once the configuration has been updated, it will need to be submitted by the usual channel update process.

### Satisfying an ACL that requires access to multiple resources

If a member makes a request that calls multiple system chaincodes, all of the ACLs for those system chaincodes must be satisfied.

For example, peer/Propose refers to any proposal request on a channel. If the particular proposal requires access to two system chaincodes that requires an identity satisfying Writers and one system chaincode that requires an identity satisfying MyPolicy, then the member submitting the proposal must have an identity that evaluates to “true” for both Writers and MyPolicy.

In the default configuration, Writers is a signature policy whose rule is SampleOrg.member. In other words, “any member of my organization”. MyPolicy, listed above, has a rule of SampleOrg.admin, or “any admin of my organization”. To satisfy these ACLs, the member would have to be both an administrator and a member of SampleOrg. By default, all administrators are members (though not all administrators are members), but it is possible to overwrite these policies to whatever you want them to be. As a result, it’s important to keep track of these policies to ensure that the ACLs for peer proposals are not impossible to satisfy (unless that is the intention).

#### Migration considerations for customers using the experimental ACL feature

Previously, the management of access control lists was done in an isolated\_data section of the channel creation transaction and updated via PEER\_RESOURCE\_UPDATE transactions. Originally, it was thought that the resources tree would handle the update of several functions that, ultimately, were handled in other ways, so maintaining a separate parallel peer configuration tree was judged to be unnecessary.

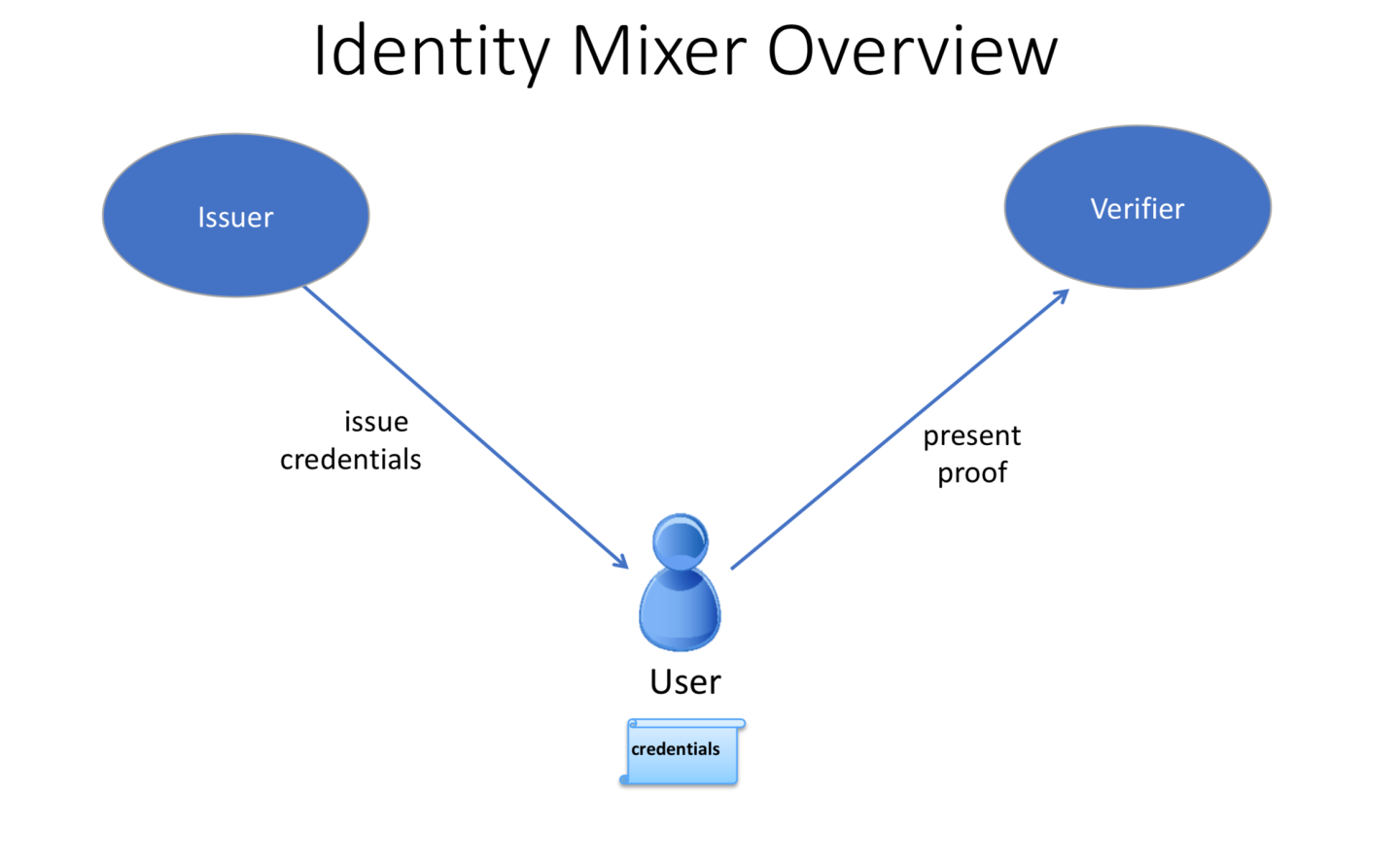
Migration for customers using the experimental resources tree in v1.1 is possible. Because the official v1.2 release does not support the old ACL methods, the network operators should shut down all their peers. Then, they should upgrade them to v1.2, submit a channel reconfiguration transaction which enables the v1.2 capability and sets the desired ACLs, and then finally restart the upgraded peers. The restarted peers will immediately consume the new channel configuration and enforce the ACLs as desired.

# MSP Implementation with Identity Mixer

## What is Idemix?

Idemix is a cryptographic protocol suite, which provides strong authentication as well as privacy-preserving features such as **anonymity**, the ability to transact without revealing the identity of the transactor, and **unlinkability**, the ability of a single identity to send multiple transactions without revealing that the transactions were sent by the same identity.

There are three actors involved in an Idemix flow: [**user**](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html), **issuer**, and **verifier**.



* An issuer certifies a set of user’s attributes are issued in the form of a digital certificate, hereafter called “credential”.
* The user later generates a “[zero-knowledge proof](https://en.wikipedia.org/wiki/Zero-knowledge_proof)” of possession of the credential and also selectively discloses only the attributes the user chooses to reveal. The proof, because it is zero-knowledge, reveals no additional information to the verifier, issuer, or anyone else.

As an example, suppose “Alice” needs to prove to Bob (a store clerk) that she has a driver’s license issued to her by the DMV.

In this scenario, Alice is the user, the DMV is the issuer, and Bob is the verifier. In order to prove to Bob that Alice has a driver’s license, she could show it to him. However, Bob would then be able to see Alice’s name, address, exact age, etc. — much more information than Bob needs to know.

Instead, Alice can use Idemix to generate a “zero-knowledge proof” for Bob, which only reveals that she has a valid driver’s license and nothing else.

So from the proof:

* Bob does not learn any additional information about Alice other than the fact that she has a valid license (anonymity).
* If Alice visits the store multiple times and generates a proof each time for Bob, Bob would not be able to tell from the proof that it was the same person (unlinkability).

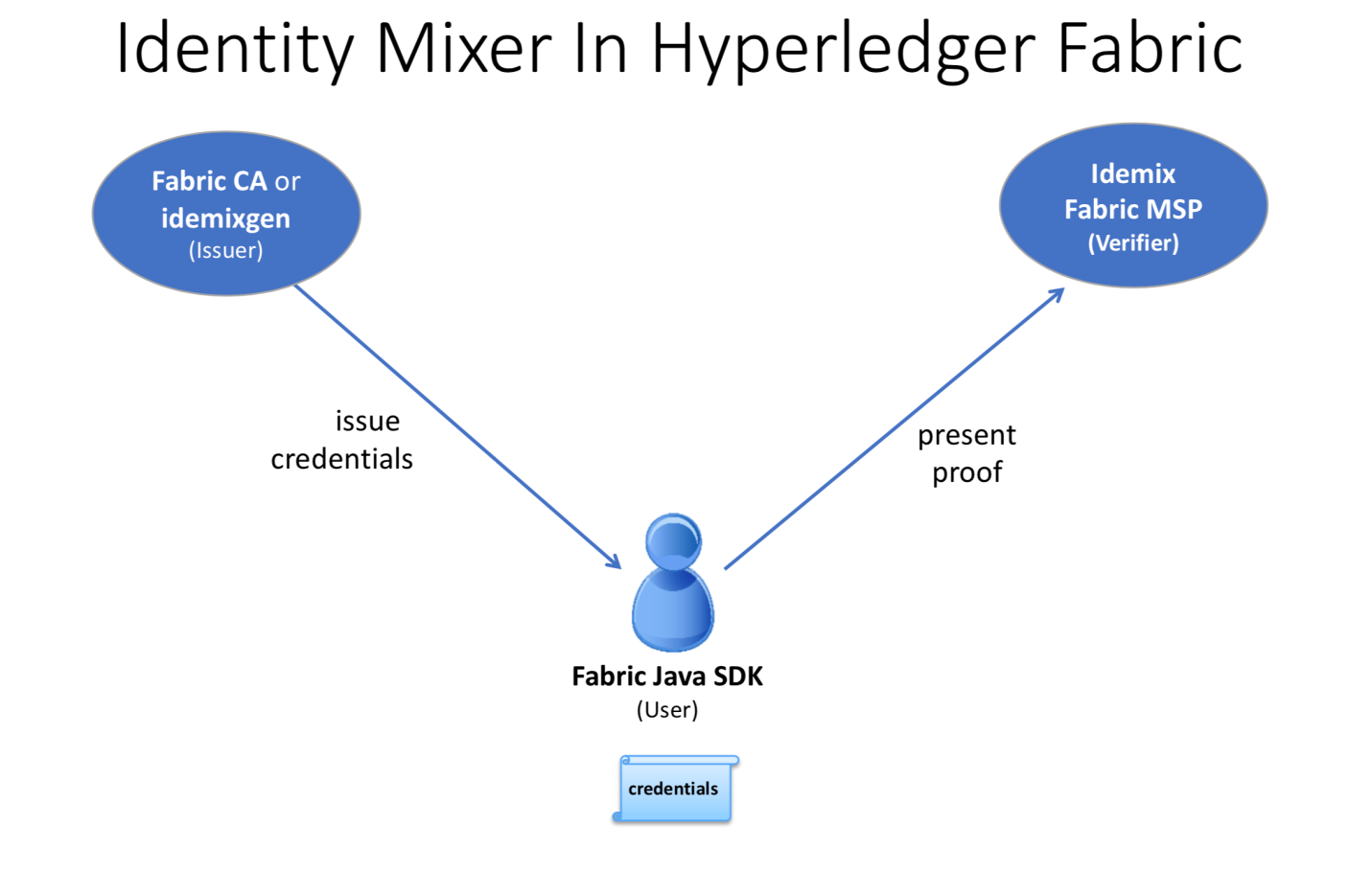
Idemix authentication technology provides the trust model and security guarantees that are similar to what is ensured by standard X.509 certificates but with underlying cryptographic algorithms that efficiently provide advanced privacy features including the ones described above. We’ll compare Idemix and X.509 technologies in detail in the technical section below.

## How to use Idemix

To understand how to use Idemix with Hyperledger Fabric, we need to see which Fabric components correspond to the user, issuer, and verifier in Idemix.

* The Fabric Java SDK is the API for the **user**. In the future, other Fabric SDKs will also support Idemix.
* Fabric provides two possible Idemix **issuers**:
  1. Fabric CA for production environments or development, and
  2. the [idemixgen](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemixgen.html) tool for development environments.
* The **verifier** is an Idemix MSP in Fabric.

In order to use Idemix in Hyperledger Fabric, the following three basic steps are required:



Compare the roles in this image to the ones above.

1. Consider the issuer.

Fabric CA (version 1.3 or later) has been enhanced to automatically function as an Idemix issuer. When fabric-ca-server is started (or initialized via the fabric-ca-server init command), the following two files are automatically created in the home directory of the fabric-ca-server: IssuerPublicKey and IssuerRevocationPublicKey. These files are required in step 2.

For a development environment and if you are not using Fabric CA, you may use [``](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html#id1)idemixgen``to create these files.

1. Consider the verifier.

You need to create an Idemix MSP using the IssuerPublicKey and IssuerRevocationPublicKey from step 1.

For example, consider the following excerpt from [configtx.yaml in the Hyperledger Java SDK sample](https://github.com/hyperledger/fabric-sdk-java/blob/master/src/test/fixture/sdkintegration/e2e-2Orgs/v1.3/configtx.yaml):

**-** **&**Org1Idemix

*# defaultorg defines the organization which is used in the sampleconfig*

*# of the fabric.git development environment*

name: idemixMSP1

*# id to load the msp definition as*

id: idemixMSPID1

msptype: idemix

mspdir: crypto**-**config**/**peerOrganizations**/**org3**.**example**.**com

The msptype is set to idemix and the contents of the mspdir directory (crypto-config/peerOrganizations/org3.example.com/msp in this example) contains the IssuerPublicKey and IssuerRevocationPublicKey files.

Note that in this example, Org1Idemix represents the Idemix MSP for Org1 (not shown), which would also have an X509 MSP.

1. Consider the user. Recall that the Java SDK is the API for the user.

There is only a single additional API call required in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html) to use Idemix with the Java SDK: the idemixEnroll method of the org.hyperledger.fabric\_ca.sdk.HFCAClient class. For example, assume hfcaClient is your HFCAClient object and x509Enrollment is your org.hyperledger.fabric.sdk.Enrollment associated with your X509 certificate.

The following call will return an org.hyperledger.fabric.sdk.Enrollment object associated with your Idemix credential.

IdemixEnrollment idemixEnrollment **=** hfcaClient**.**idemixEnroll(x509enrollment, "idemixMSPID1");

Note also that IdemixEnrollment implements the org.hyperledger.fabric.sdk.Enrollment interface and can, therefore, be used in the same way that one uses the X509 enrollment object, except, of course, that this automatically provides the privacy enhancing features of Idemix.

## Idemix and chaincode

From a verifier perspective, there is one more actor to consider: chaincode. What can chaincode learn about the transactor when an Idemix credential is used?

The [cid (Client Identity) library](https://github.com/hyperledger/fabric/tree/master/core/chaincode/shim/ext/cid) (for golang only) has been extended to support the GetAttributeValue function when an Idemix credential is used. However, as mentioned in the “Current limitations” section below, there are only two attributes which are disclosed in the Idemix case: ou and role.

If Fabric CA is the credential issuer:

* the value of the ou attribute is the identity’s **affiliation** (e.g. “org1.department1”);
* the value of the role attribute will be either ‘member’ or ‘admin’. A value of ‘admin’ means that the identity is an MSP administrator. By default, identities created by Fabric CA will return the ‘member’ role. In order to create an ‘admin’ identity, [register](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html) the identity with the role attribute and a value of 2.

For an example of setting an affiliation in the Java SDK see this [sample](https://github.com/hyperledger/fabric-sdk-java/blob/release-1.4/src/test/java/org/hyperledger/fabric/sdkintegration/End2endIdemixIT.java#L121).

For an example of using the CID [library](https://hyperledger-fabric.readthedocs.io/en/release-1.4/idemix.html) in go chaincode to retrieve attributes, see this [go chaincode](https://github.com/hyperledger/fabric-sdk-java/blob/release-1.4/src/test/fixture/sdkintegration/gocc/sampleIdemix/src/github.com/example_cc/example_cc.go#L88).

## Current limitations

The current version of Idemix does have a few limitations.

* **Fixed set of attributes**

It not yet possible to issue or use an Idemix credential with custom attributes. Custom attributes will be supported in a future release.

The following four attributes are currently supported:

* 1. Organizational Unit attribute (“ou”):
  2. Usage: same as X.509
  3. Type: String
  4. Revealed: always
  5. Role attribute (“role”):
  + Usage: same as X.509
  + Type: integer
  + Revealed: always
  1. Enrollment ID attribute
  + Usage: uniquely identify a user — same in all enrollment credentials that belong to the same user (will be used for auditing in the future releases)
  + Type: BIG
  + Revealed: never in the signature, only when generating an authentication token for Fabric CA
  1. Revocation Handle attribute
  + Usage: uniquely identify a credential (will be used for revocation in future releases)
  + Type: integer
  + Revealed: never
* **Revocation is not yet supported**

Although much of the revocation framework is in place as can be seen by the presence of a revocation handle attribute mentioned above, revocation of an Idemix credential is not yet supported.

* **Peers do not use Idemix for endorsement**

Currently, Idemix MSP is used by the peers only for signature verification. Signing with Idemix is only done via Client SDK. More roles (including a ‘peer’ role) will be supported by Idemix MSP.

## Technical summary

### Comparing Idemix credentials to X.509 certificates

The certificate/credential concept and the issuance process are very similar in Idemix and X.509 certs: a set of attributes is digitally signed with a signature that cannot be forged and there is a secret key to which a credential is cryptographically bound.

The main difference between a standard X.509 certificate and an Identity Mixer credential is the signature scheme that is used to certify the attributes. The signatures underlying the Identity Mixer system allow for efficient proofs of the possession of a signature and the corresponding attributes without revealing the signature and (selected) attribute values themselves. We use zero-knowledge proofs to ensure that such “knowledge” or “information” is not revealed while ensuring that the signature over some attributes is valid and the user is in possession of the corresponding credential secret key.

Such proofs, like X.509 certificates, can be verified with the public key of the authority that originally signed the credential and cannot be successfully forged. Only the user who knows the credential secret key can generate the proofs about the credential and its attributes.

With regard to unlinkability, when an X.509 certificate is presented, all attributes have to be revealed to verify the certificate signature. This implies that all certificate usages for signing transactions are linkable.

To avoid such linkability, fresh X.509 certificates need to be used every time, which results in complex key management and communication and storage overhead. Furthermore, there are cases where it is important that not even the CA issuing the certificates is able to link all the transactions to the user.

Idemix helps to avoid linkability with respect to both the CA and verifiers, since even the CA is not able to link proofs to the original credential. Neither the issuer nor a verifier can tell whether two proofs were derived from the same credential (or from two different ones).

More details on the concepts and features of the Identity Mixer technology are described in the paper [Concepts and Languages for Privacy-Preserving Attribute-Based Authentication](https://link.springer.com/chapter/10.1007%2F978-3-642-37282-7_4).

### Topology Information

Given the above limitations, it is recommended to have only one Idemix-based MSP per channel or, at the extreme, per network. Indeed, for example, having multiple Idemix-based MSPs per channel would allow a party, reading the ledger of that channel, to tell apart transactions signed by parties belonging to different Idemix-based MSPs. This is because, each transaction leak the MSP-ID of the signer. In other words, Idemix currently provides only anonymity of clients among the same organization (MSP).

In the future, Idemix could be extended to support anonymous hierarchies of Idemix-based Certification Authorities whose certified credentials can be verified by using a unique public-key, therefore achieving anonymity across organizations (MSPs). This would allow multiple Idemix-based MSPs to coexist in the same channel.

In principal, a channel can be configured to have a single Idemix-based MSP and multiple X.509-based MSPs. Of course, the interaction between these MSP can potential leak information. An assessment of the leaked information need to be done case by case.wq

### Underlying cryptographic protocols

Idemix technology is built from a blind signature scheme that supports multiple messages and efficient zero-knowledge proofs of signature possession. All of the cryptographic building blocks for Idemix were published at the top conferences and journals and verified by the scientific community.

This particular Idemix implementation for Fabric uses a pairing-based signature scheme that was briefly proposed by [Camenisch and Lysyanskaya](https://link.springer.com/chapter/10.1007/978-3-540-28628-8_4) and described in detail by [Au et al.](https://link.springer.com/chapter/10.1007/11832072_8). The ability to prove knowledge of a signature in a zero-knowledge proof [Camenisch et al.](https://eprint.iacr.org/2016/663.pdf) was used.

# Identity Mixer MSP configuration generator (idemixgen)

This document describes the usage for the idemixgen utility, which can be used to create configuration files for the identity mixer based MSP. Two commands are available, one for creating a fresh CA key pair, and one for creating an MSP config using a previously generated CA key.

## Directory Structure

The idemixgen tool will create directories with the following structure:

**-** **/**ca**/**

IssuerSecretKey

IssuerPublicKey

RevocationKey

**-** **/**msp**/**

IssuerPublicKey

RevocationPublicKey

**-** **/**user**/**

SignerConfig

The ca directory contains the issuer secret key (including the revocation key) and should only be present for a CA. The msp directory contains the information required to set up an MSP verifying idemix signatures. The user directory specifies a default signer.

## CA Key Generation

CA (issuer) keys suitable for identity mixer can be created using command idemixgen ca-keygen. This will create directories ca and msp in the working directory.

## Adding a Default Signer

After generating the ca and msp directories with idemixgen ca-keygen, a default signer specified in the user directory can be added to the config with idemixgen signerconfig.

$ idemixgen signerconfig -h

usage: idemixgen signerconfig [<flags>]

Generate a default signer for this Idemix MSP

Flags:

-h, --help Show context-sensitive help (also try --help-long and --help-man).

-u, --org-unit=ORG-UNIT The Organizational Unit of the default signer

-a, --admin Make the default signer admin

-e, --enrollment-id=ENROLLMENT-ID

The enrollment id of the default signer

-r, --revocation-handle=REVOCATION-HANDLE

The handle used to revoke this signer

For example, we can create a default signer that is a member of organizational unit “OrgUnit1”, with enrollment identity “johndoe”, revocation handle “1234”, and that is an admin, with the following command:

idemixgen signerconfig **-**u OrgUnit1 **--**admin **-**e "johndoe" **-**r 1234

# The Operations Service

The peer and the orderer host an HTTP server that offers a RESTful “operations” API. This API is unrelated to the Fabric network services and is intended to be used by operators, not administrators or “users” of the network.

The API exposes the following capabilities:

* Log level management
* Health checks
* Prometheus target for operational metrics (when configured)
* Version information

## Configuring the Operations Service

The operations service requires two basic pieces of configuration:

* The **address** and **port** to listen on.
* The **TLS certificates** and **keys** to use for authentication and encryption. Note, **these certificates should be generated by a separate and dedicated CA**. Do not use a CA that has generated certificates for any organizations in any channels.

### Peer

For each peer, the operations server can be configured in the operations section of core.yaml:

operations:

*# host and port for the operations server*

listenAddress: 127.0**.**0.1:9443

*# TLS configuration for the operations endpoint*

tls:

*# TLS enabled*

enabled: true

*# path to PEM encoded server certificate for the operations server*

cert:

file: tls**/**server**.**crt

*# path to PEM encoded server key for the operations server*

key:

file: tls**/**server**.**key

*# most operations service endpoints require client authentication when TLS*

*# is enabled. clientAuthRequired requires client certificate authentication*

*# at the TLS layer to access all resources.*

clientAuthRequired: false

*# paths to PEM encoded ca certificates to trust for client authentication*

clientRootCAs:

files: []

The listenAddress key defines the host and port that the operation server will [listen](https://hyperledger-fabric.readthedocs.io/en/release-1.4/operations_service.html) on. If the server should listen on all addresses, the host portion can be omitted.

The tls section is used to indicate whether or not TLS is enabled for the operations service, the location of the service’s certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When enabled is true, most of the operations service endpoints require client authentication, therefore clientRootCAs.files must be set. When clientAuthRequired is true, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more details.

### Orderer

For each orderer, the operations server can be configured in the Operations section of orderer.yaml:

Operations:

*# host and port for the operations server*

ListenAddress: 127.0**.**0.1:8443

*# TLS configuration for the operations endpoint*

TLS:

*# TLS enabled*

Enabled: true

*# PrivateKey: PEM-encoded tls key for the operations endpoint*

PrivateKey: tls**/**server**.**key

*# Certificate governs the file location of the server TLS certificate.*

Certificate: tls**/**server**.**crt

*# Paths to PEM encoded ca certificates to trust for client authentication*

ClientRootCAs: []

*# Most operations service endpoints require client authentication when TLS*

*# is enabled. ClientAuthRequired requires client certificate authentication*

*# at the TLS layer to access all resources.*

ClientAuthRequired: false

The ListenAddress key defines the host and port that the operations server will listen on. If the server should listen on all addresses, the host portion can be omitted.

The TLS section is used to indicate whether or not TLS is enabled for the operations service, the location of the service’s certificate and private key, and the locations of certificate authority root certificates that should be trusted for client authentication. When Enabled is true, most of the operations service endpoints require client authentication, therefore RootCAs must be set. When ClientAuthRequired is true, the TLS layer will require clients to provide a certificate for authentication on every request. See Operations Security section below for more [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/operations_service.html).

### Operations Security

As the operations service is focused on operations and intentionally unrelated to the Fabric network, it does not use the Membership Services Provider for access control. Instead, the operations service relies entirely on mutual TLS with client certificate authentication.

When TLS is disabled, authorization is bypassed and any client that can connect to the operations endpoint will be able to use the API.

When TLS is enabled, a valid client certificate must be provided in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/operations_service.html) to access all resources unless explicitly noted otherwise below.

When clientAuthRequired is also enabled, the TLS layer will require a valid client certificate regardless of the resource being accessed.

### Log Level Management

The operations service provides a /logspec resource that operators can use to manage the active logging spec for a peer or orderer. The resource is a conventional REST resource and supports GET and PUT requests.

When a GET /logspec request is received by the operations service, it will respond with a JSON payload that contains the current logging specification:

{"spec":"info"}

When a PUT /logspec request is received by the operations service, it will read the body as a JSON payload. The payload must consist of a single attribute named spec.

{"spec":"chaincode=debug:info"}

If the spec is activated successfully, the service will respond with a 204 "No Content" response. If an error occurs, the service will respond with a 400 "Bad Request" and an error payload:

{"error":"error message"}

## Health Checks

The operations service provides a /healthz resource that operators can use to help determine the liveness and health of peers and orderers. The resource is a conventional REST resource that supports GET requests. The implementation is intended to be compatible with the liveness probe model used by Kubernetes but can be used in other contexts.

When a GET /healthz request is received, the operations service will call all registered health checkers for the process. When all of the health checkers return successfully, the operations service will respond with a 200 "OK" and a JSON body:

{

"status": "OK",

"time": "2009-11-10T23:00:00Z"

}

If one or more of the health checkers returns an error, the operations service will respond with a 503 "Service Unavailable" and a JSON body that includes information about which health checker failed:

{

"status": "Service Unavailable",

"time": "2009-11-10T23:00:00Z",

"failed\_checks": [

{

"component": "docker",

"reason": "failed to connect to Docker daemon: invalid endpoint"

}

]

}

In the current version, the only health check that is registered is for Docker. Future versions will be enhanced to add additional health checks.

When TLS is enabled, a valid client certificate is not required to use this service unless clientAuthRequired is set to true.

## Metrics

Some components of the Fabric peer and orderer expose metrics that can help provide insight into the behavior of the system. Operators and administrators can use this information to better understand how the system is performing over time.

### Configuring Metrics

Fabric provides two ways to expose metrics: a **pull** model based on Prometheus and a **push** model based on StatsD.

### Prometheus

A typical Prometheus deployment scrapes metrics by requesting them from an HTTP endpoint exposed by instrumented targets. As Prometheus is responsible for requesting the metrics, it is considered a pull system.

When configured, a Fabric peer or orderer will present a /metrics resource on the operations service.

#### Peer

A peer can be configured to expose a /metrics endpoint for Prometheus to scrape by setting the metrics provider to prometheus in the metrics section of core.yaml.

metrics:

provider: prometheus

#### Orderer

An orderer can be configured to expose a /metrics endpoint for Prometheus to scrape by setting the metrics provider to prometheus in the Metrics section of orderer.yaml.

Metrics:

Provider: prometheus

### StatsD

StatsD is a simple statistics aggregation daemon. Metrics are sent to a statsd daemon where they are collected, aggregated, and pushed to a backend for visualization and alerting. As this model requires instrumented processes to send metrics data to StatsD, this is considered a push system.

#### Peer

A peer can be configured to send metrics to StatsD by setting the metrics provider to statsd in the metrics section of core.yaml. The statsd subsection must also be configured with the address of the StatsD daemon, the network type to use (tcp or udp), and how often to send the metrics. An optional prefix may be specified to help differentiate the source of the metrics — for example, differentiating metrics coming from separate peers — that would be prepended to all generated metrics.

metrics:

provider: statsd

statsd:

network: udp

address: 127.0**.**0.1:8125

writeInterval: 10s

prefix: peer**-**0

#### Orderer

An orderer can be configured to send metrics to StatsD by setting the metrics provider to statsd in the Metrics section of orderer.yaml. The Statsd subsection must also be configured with the address of the StatsD daemon, the network type to use (tcp or udp), and how often to send the metrics. An optional prefix may be specified to help differentiate the source of the metrics.

Metrics:

Provider: statsd

Statsd:

Network: udp

Address: 127.0**.**0.1:8125

WriteInterval: 30s

Prefix: org**-**orderer

For a look at the different metrics that are generated, check out [Metrics Reference](https://hyperledger-fabric.readthedocs.io/en/release-1.4/metrics_reference.html).

## Version

The orderer and peer both expose a /version endpoint. This endpoint serves a JSON document containing the orderer or peer version and the commit SHA on which the release was cut.

# Metrics Reference

## Prometheus Metrics

The following metrics are currently exported for consumption by Prometheus.

| **Name** | **Type** | **Description** | **Labels** |
| --- | --- | --- | --- |
| blockcutter\_block\_fill\_duration | histogram | The time from first transaction enqueing to the block being cut in seconds. | channel |
| broadcast\_enqueue\_duration | histogram | The time to enqueue a transaction in seconds. | channel type status |
| broadcast\_processed\_count | counter | The [number](https://hyperledger-fabric.readthedocs.io/en/release-1.4/metrics_reference.html) of transactions processed. | channel type status |
| broadcast\_validate\_duration | histogram | The time to validate a transaction in seconds. | channel type status |
| chaincode\_execute\_timeouts | counter | The number of chaincode executions (Init or Invoke) that have timed out. | chaincode |
| chaincode\_launch\_duration | histogram | The time to launch a chaincode. | chaincode success |
| chaincode\_launch\_failures | counter | The number of chaincode launches that have failed. | chaincode |
| chaincode\_launch\_timeouts | counter | The number of chaincode launches that have timed out. | chaincode |
| chaincode\_shim\_request\_duration | histogram | The time to complete chaincode shim requests. | type channel chaincode success |
| chaincode\_shim\_requests\_completed | counter | The number of chaincode shim requests completed. | type channel chaincode success |
| chaincode\_shim\_requests\_received | counter | The number of chaincode shim requests received. | type channel chaincode |
| cluster\_comm\_egress\_queue\_capacity | gauge | Capacity of the egress queue. | host msg\_type channel |
| cluster\_comm\_egress\_queue\_length | gauge | Length of the egress queue. | host msg\_type channel |
| cluster\_comm\_egress\_queue\_workers | gauge | Count of egress queue workers. | channel |
| cluster\_comm\_egress\_stream\_count | gauge | Count of streams to other nodes. | channel |
| cluster\_comm\_egress\_tls\_connection\_count | gauge | Count of TLS connections to other nodes. |  |
| cluster\_comm\_ingress\_stream\_count | gauge | Count of streams from other nodes. |  |
| cluster\_comm\_msg\_dropped\_count | counter | Count of messages dropped. | host channel |
| cluster\_comm\_msg\_send\_time | histogram | The time it takes to send a message in seconds. | host channel |
| consensus\_etcdraft\_cluster\_size | gauge | Number of nodes in this channel. | channel |
| consensus\_etcdraft\_committed\_block\_number | gauge | The block number of the latest block committed. | channel |
| consensus\_etcdraft\_config\_proposals\_received | counter | The total number of proposals received for config type transactions. | channel |
| consensus\_etcdraft\_data\_persist\_duration | histogram | The time taken for etcd/raft data to be persisted in storage (in seconds). | channel |
| consensus\_etcdraft\_is\_leader | gauge | The leadership status of the current node: 1 if it is the leader else 0. | channel |
| consensus\_etcdraft\_leader\_changes | counter | The number of leader changes since process [start](https://hyperledger-fabric.readthedocs.io/en/release-1.4/metrics_reference.html). | channel |
| consensus\_etcdraft\_normal\_proposals\_received | counter | The total number of proposals received for normal type transactions. | channel |
| consensus\_etcdraft\_proposal\_failures | counter | The number of proposal failures. | channel |
| consensus\_etcdraft\_snapshot\_block\_number | gauge | The block number of the latest snapshot. | channel |
| consensus\_kafka\_batch\_size | gauge | The mean batch size in bytes sent to topics. | topic |
| consensus\_kafka\_compression\_ratio | gauge | The mean compression ratio (as percentage) for topics. | topic |
| consensus\_kafka\_incoming\_byte\_rate | gauge | Bytes/second read off brokers. | broker\_id |
| consensus\_kafka\_last\_offset\_persisted | gauge | The offset specified in the block metadata of the most recently committed block. | channel |
| consensus\_kafka\_outgoing\_byte\_rate | gauge | Bytes/second written to brokers. | broker\_id |
| consensus\_kafka\_record\_send\_rate | gauge | The number of records per second sent to topics. | topic |
| consensus\_kafka\_records\_per\_request | gauge | The mean number of records sent per request to topics. | topic |
| consensus\_kafka\_request\_latency | gauge | The mean request latency in ms to brokers. | broker\_id |
| consensus\_kafka\_request\_rate | gauge | Requests/second sent to brokers. | broker\_id |
| consensus\_kafka\_request\_size | gauge | The mean request size in bytes to brokers. | broker\_id |
| consensus\_kafka\_response\_rate | gauge | Requests/second sent to brokers. | broker\_id |
| consensus\_kafka\_response\_size | gauge | The mean response size in bytes from brokers. | broker\_id |
| couchdb\_processing\_time | histogram | Time taken in seconds for the function to complete request to CouchDB | database function\_name result |
| deliver\_blocks\_sent | counter | The number of blocks sent by the deliver service. | channel filtered |
| deliver\_requests\_completed | counter | The number of deliver requests that have been completed. | channel filtered success |
| deliver\_requests\_received | counter | The number of deliver requests that have been received. | channel filtered |
| deliver\_streams\_closed | counter | The number of GRPC streams that have been closed for the deliver service. |  |
| deliver\_streams\_opened | counter | The number of GRPC streams that have been opened for the deliver service. |  |
| dockercontroller\_chaincode\_container\_build\_duration | histogram | The time to build a chaincode image in seconds. | chaincode success |
| endorser\_chaincode\_instantiation\_failures | counter | The number of chaincode instantiations or upgrade that have failed. | channel chaincode |
| endorser\_duplicate\_transaction\_failures | counter | The number of failed proposals due to duplicate transaction ID. | channel chaincode |
| endorser\_endorsement\_failures | counter | The number of failed endorsements. | channel chaincode chaincodeerror |
| endorser\_proposal\_acl\_failures | counter | The number of proposals that failed ACL checks. | channel chaincode |
| endorser\_proposal\_validation\_failures | counter | The number of proposals that have failed initial validation. |  |
| endorser\_proposals\_received | counter | The number of proposals received. |  |
| endorser\_propsal\_duration | histogram | The time to complete a proposal. | channel chaincode success |
| endorser\_successful\_proposals | counter | The number of successful proposals. |  |
| fabric\_version | gauge | The active version of Fabric. | version |
| gossip\_comm\_messages\_received | counter | Number of messages received |  |
| gossip\_comm\_messages\_sent | counter | Number of messages sent |  |
| gossip\_comm\_overflow\_count | counter | Number of outgoing queue buffer overflows |  |
| gossip\_leader\_election\_leader | gauge | Peer is leader (1) or follower (0) | channel |
| gossip\_membership\_total\_peers\_known | gauge | Total known peers | channel |
| gossip\_payload\_buffer\_size | gauge | Size of the payload buffer | channel |
| gossip\_privdata\_commit\_block\_duration | histogram | Time it takes to commit private data and the corresponding block (in seconds) | channel |
| gossip\_privdata\_fetch\_duration | histogram | Time it takes to fetch missing private data from peers (in seconds) | channel |
| gossip\_privdata\_list\_missing\_duration | histogram | Time it takes to list the missing private data (in seconds) | channel |
| gossip\_privdata\_pull\_duration | histogram | Time it takes to pull a missing private data element (in seconds) | channel |
| gossip\_privdata\_purge\_duration | histogram | Time it takes to purge private data (in seconds) | channel |
| gossip\_privdata\_reconciliation\_duration | histogram | Time it takes for reconciliation to complete (in seconds) | channel |
| gossip\_privdata\_retrieve\_duration | histogram | Time it takes to retrieve missing private data elements from the ledger (in seconds) | channel |
| gossip\_privdata\_send\_duration | histogram | Time it takes to send a missing private data element (in seconds) | channel |
| gossip\_privdata\_validation\_duration | histogram | Time it takes to validate a block (in seconds) | channel |
| gossip\_state\_commit\_duration | histogram | Time it takes to commit a block in seconds | channel |
| gossip\_state\_height | gauge | Current ledger height | channel |
| grpc\_comm\_conn\_closed | counter | gRPC connections closed. Open minus closed is the active number of connections. |  |
| grpc\_comm\_conn\_opened | counter | gRPC connections opened. Open minus closed is the active number of connections. |  |
| grpc\_server\_stream\_messages\_received | counter | The number of stream messages received. | service method |
| grpc\_server\_stream\_messages\_sent | counter | The number of stream messages sent. | service method |
| grpc\_server\_stream\_request\_duration | histogram | The time to complete a stream request. | service method code |
| grpc\_server\_stream\_requests\_completed | counter | The number of stream requests completed. | service method code |
| grpc\_server\_stream\_requests\_received | counter | The number of stream requests received. | service method |
| grpc\_server\_unary\_request\_duration | histogram | The time to complete a unary request. | service method code |
| grpc\_server\_unary\_requests\_completed | counter | The number of unary requests completed. | service method code |
| grpc\_server\_unary\_requests\_received | counter | The number of unary requests received. | service method |
| ledger\_block\_processing\_time | histogram | Time taken in seconds for ledger block processing. | channel |
| ledger\_blockchain\_height | gauge | Height of the chain in blocks. | channel |
| ledger\_blockstorage\_and\_pvtdata\_commit\_time | histogram | Time taken in seconds for committing the block and private data to storage. | channel |
| ledger\_blockstorage\_commit\_time | histogram | Time taken in seconds for committing the block to storage. | channel |
| ledger\_statedb\_commit\_time | histogram | Time taken in seconds for committing block changes to state db. | channel |
| ledger\_transaction\_count | counter | Number of transactions processed. | channel transaction\_type chaincode validation\_code |
| logging\_entries\_checked | counter | Number of log entries checked against the active logging level | level |
| logging\_entries\_written | counter | Number of log entries that are written | level |

## StatsD Metrics

The following metrics are currently emitted for consumption by StatsD. The %{variable\_name} nomenclature represents segments that vary based on context.

For example, %{channel} will be replaced with the name of the channel associated with the metric.

| **Bucket** | **Type** | **Description** |
| --- | --- | --- |
| blockcutter.block\_fill\_duration.%{channel} | histogram | The time from first transaction enqueing to the block being cut in seconds. |
| broadcast.enqueue\_duration.%{channel}.%{type}.%{status} | histogram | The time to enqueue a transaction in seconds. |
| broadcast.processed\_count.%{channel}.%{type}.%{status} | counter | The number of transactions processed. |
| broadcast.validate\_duration.%{channel}.%{type}.%{status} | histogram | The time to validate a transaction in seconds. |
| chaincode.execute\_timeouts.%{chaincode} | counter | The number of chaincode executions (Init or Invoke) that have timed out. |
| chaincode.launch\_duration.%{chaincode}.%{success} | histogram | The time to launch a chaincode. |
| chaincode.launch\_failures.%{chaincode} | counter | The number of chaincode launches that have failed. |
| chaincode.launch\_timeouts.%{chaincode} | counter | The number of chaincode launches that have timed out. |
| chaincode.shim\_request\_duration.%{type}.%{channel}.%{chaincode}.%{success} | histogram | The time to complete chaincode shim requests. |
| chaincode.shim\_requests\_completed.%{type}.%{channel}.%{chaincode}.%{success} | counter | The number of chaincode shim requests completed. |
| chaincode.shim\_requests\_received.%{type}.%{channel}.%{chaincode} | counter | The number of chaincode shim requests received. |
| cluster.comm.egress\_queue\_capacity.%{host}.%{msg\_type}.%{channel} | gauge | Capacity of the egress queue. |
| cluster.comm.egress\_queue\_length.%{host}.%{msg\_type}.%{channel} | gauge | Length of the egress queue. |
| cluster.comm.egress\_queue\_workers.%{channel} | gauge | Count of egress queue workers. |
| cluster.comm.egress\_stream\_count.%{channel} | gauge | Count of streams to other nodes. |
| cluster.comm.egress\_tls\_connection\_count | gauge | Count of TLS connections to other nodes. |
| cluster.comm.ingress\_stream\_count | gauge | Count of streams from other nodes. |
| cluster.comm.msg\_dropped\_count.%{host}.%{channel} | counter | Count of messages dropped. |
| cluster.comm.msg\_send\_time.%{host}.%{channel} | histogram | The time it takes to send a message in seconds. |
| consensus.etcdraft.cluster\_size.%{channel} | gauge | Number of nodes in this channel. |
| consensus.etcdraft.committed\_block\_number.%{channel} | gauge | The block number of the latest block committed. |
| consensus.etcdraft.config\_proposals\_received.%{channel} | counter | The total number of proposals received for config type transactions. |
| consensus.etcdraft.data\_persist\_duration.%{channel} | histogram | The time taken for etcd/raft data to be persisted in storage (in seconds). |
| consensus.etcdraft.is\_leader.%{channel} | gauge | The leadership status of the current node: 1 if it is the leader else 0. |
| consensus.etcdraft.leader\_changes.%{channel} | counter | The number of leader changes since process start. |
| consensus.etcdraft.normal\_proposals\_received.%{channel} | counter | The total number of proposals received for normal type transactions. |
| consensus.etcdraft.proposal\_failures.%{channel} | counter | The number of proposal failures. |
| consensus.etcdraft.snapshot\_block\_number.%{channel} | gauge | The block number of the latest snapshot. |
| consensus.kafka.batch\_size.%{topic} | gauge | The mean batch size in bytes sent to topics. |
| consensus.kafka.compression\_ratio.%{topic} | gauge | The mean compression ratio (as percentage) for topics. |
| consensus.kafka.incoming\_byte\_rate.%{broker\_id} | gauge | Bytes/second read off brokers. |
| consensus.kafka.last\_offset\_persisted.%{channel} | gauge | The offset specified in the block metadata of the most recently committed block. |
| consensus.kafka.outgoing\_byte\_rate.%{broker\_id} | gauge | Bytes/second written to brokers. |
| consensus.kafka.record\_send\_rate.%{topic} | gauge | The number of records per second sent to topics. |
| consensus.kafka.records\_per\_request.%{topic} | gauge | The mean number of records sent per request to topics. |
| consensus.kafka.request\_latency.%{broker\_id} | gauge | The mean request latency in ms to brokers. |
| consensus.kafka.request\_rate.%{broker\_id} | gauge | Requests/second sent to brokers. |
| consensus.kafka.request\_size.%{broker\_id} | gauge | The mean request size in bytes to brokers. |
| consensus.kafka.response\_rate.%{broker\_id} | gauge | Requests/second sent to brokers. |
| consensus.kafka.response\_size.%{broker\_id} | gauge | The mean response size in bytes from brokers. |
| couchdb.processing\_time.%{database}.%{function\_name}.%{result} | histogram | Time taken in seconds for the function to complete request to CouchDB |
| deliver.blocks\_sent.%{channel}.%{filtered} | counter | The number of blocks sent by the deliver service. |
| deliver.requests\_completed.%{channel}.%{filtered}.%{success} | counter | The number of deliver requests that have been completed. |
| deliver.requests\_received.%{channel}.%{filtered} | counter | The number of deliver requests that have been received. |
| deliver.streams\_closed | counter | The number of GRPC streams that have been closed for the deliver service. |
| deliver.streams\_opened | counter | The number of GRPC streams that have been opened for the deliver service. |
| dockercontroller.chaincode\_container\_build\_duration.%{chaincode}.%{success} | histogram | The time to build a chaincode image in seconds. |
| endorser.chaincode\_instantiation\_failures.%{channel}.%{chaincode} | counter | The number of chaincode instantiations or upgrade that have failed. |
| endorser.duplicate\_transaction\_failures.%{channel}.%{chaincode} | counter | The number of failed proposals due to duplicate transaction ID. |
| endorser.endorsement\_failures.%{channel}.%{chaincode}.%{chaincodeerror} | counter | The number of failed endorsements. |
| endorser.proposal\_acl\_failures.%{channel}.%{chaincode} | counter | The number of proposals that failed ACL checks. |
| endorser.proposal\_validation\_failures | counter | The number of proposals that have failed initial validation. |
| endorser.proposals\_received | counter | The number of proposals received. |
| endorser.propsal\_duration.%{channel}.%{chaincode}.%{success} | histogram | The time to complete a proposal. |
| endorser.successful\_proposals | counter | The number of successful proposals. |
| fabric\_version.%{version} | gauge | The active version of Fabric. |
| gossip.comm.messages\_received | counter | Number of messages received |
| gossip.comm.messages\_sent | counter | Number of messages sent |
| gossip.comm.overflow\_count | counter | Number of outgoing queue buffer overflows |
| gossip.leader\_election.leader.%{channel} | gauge | Peer is leader (1) or follower (0) |
| gossip.membership.total\_peers\_known.%{channel} | gauge | Total known peers |
| gossip.payload\_buffer.size.%{channel} | gauge | Size of the payload buffer |
| gossip.privdata.commit\_block\_duration.%{channel} | histogram | Time it takes to commit private data and the corresponding block (in seconds) |
| gossip.privdata.fetch\_duration.%{channel} | histogram | Time it takes to fetch missing private data from peers (in seconds) |
| gossip.privdata.list\_missing\_duration.%{channel} | histogram | Time it takes to list the missing private data (in seconds) |
| gossip.privdata.pull\_duration.%{channel} | histogram | Time it takes to pull a missing private data element (in seconds) |
| gossip.privdata.purge\_duration.%{channel} | histogram | Time it takes to purge private data (in seconds) |
| gossip.privdata.reconciliation\_duration.%{channel} | histogram | Time it takes for reconciliation to complete (in seconds) |
| gossip.privdata.retrieve\_duration.%{channel} | histogram | Time it takes to retrieve missing private data elements from the ledger (in seconds) |
| gossip.privdata.send\_duration.%{channel} | histogram | Time it takes to send a missing private data element (in seconds) |
| gossip.privdata.validation\_duration.%{channel} | histogram | Time it takes to validate a block (in seconds) |
| gossip.state.commit\_duration.%{channel} | histogram | Time it takes to commit a block in seconds |
| gossip.state.height.%{channel} | gauge | Current ledger height |
| grpc.comm.conn\_closed | counter | gRPC connections closed. Open minus closed is the active number of connections. |
| grpc.comm.conn\_opened | counter | gRPC connections opened. Open minus closed is the active number of connections. |
| grpc.server.stream\_messages\_received.%{service}.%{method} | counter | The number of stream messages received. |
| grpc.server.stream\_messages\_sent.%{service}.%{method} | counter | The number of stream messages sent. |
| grpc.server.stream\_request\_duration.%{service}.%{method}.%{code} | histogram | The time to complete a stream request. |
| grpc.server.stream\_requests\_completed.%{service}.%{method}.%{code} | counter | The number of stream requests completed. |
| grpc.server.stream\_requests\_received.%{service}.%{method} | counter | The number of stream requests received. |
| grpc.server.unary\_request\_duration.%{service}.%{method}.%{code} | histogram | The time to complete a unary request. |
| grpc.server.unary\_requests\_completed.%{service}.%{method}.%{code} | counter | The number of unary requests completed. |
| grpc.server.unary\_requests\_received.%{service}.%{method} | counter | The number of unary requests received. |
| ledger.block\_processing\_time.%{channel} | histogram | Time taken in seconds for ledger block processing. |
| ledger.blockchain\_height.%{channel} | gauge | Height of the chain in blocks. |
| ledger.blockstorage\_and\_pvtdata\_commit\_time.%{channel} | histogram | Time taken in seconds for committing the block and private data to storage. |
| ledger.blockstorage\_commit\_time.%{channel} | histogram | Time taken in seconds for committing the block to storage. |
| ledger.statedb\_commit\_time.%{channel} | histogram | Time taken in seconds for committing block changes to state db. |
| ledger.transaction\_count.%{channel}.%{transaction\_type}.%{chaincode}.%{validation\_code} | counter | Number of transactions processed. |
| logging.entries\_checked.%{level} | counter | Number of log entries checked against the active logging level |
| logging.entries\_written.%{level} | counter | Number of log entries that are written |

# Error handling

## General Overview

Hyperledger Fabric code should use the vendored package **github.com/pkg/errors** in place of the standard error type provided by Go. This package allows easy generation and display of stack traces with error messages.

## Usage Instructions

**github.com/pkg/errors** should be used in place of all calls to fmt.Errorf() or errors.New(). Using this package will generate a call stack that will be appended to the error message.

Using this package is simple and will only require easy tweaks to your code.

First, you’ll need to import **github.com/pkg/errors**.

Next, update all errors that are generated by your code to use one of the error creation functions (errors.New(), errors.Errorf(), errors.WithMessage(), errors.Wrap(), errors.Wrapf().

**Note**

See <https://godoc.org/github.com/pkg/errors> for complete documentation of the available error creation function. Also, refer to the General guidelines section below for more specific guidelines for using the package for Fabric code.

Finally, change the formatting directive for any logger or fmt.Printf() calls from %s to %+v to print the call stack along with the error message.

## General guidelines for error handling in Hyperledger Fabric

* If you are servicing a [user](https://hyperledger-fabric.readthedocs.io/en/release-1.4/error-handling.html) request, you should log the error and return it.
* If the error comes from an external source, such as a Go library or vendored package, wrap the error using errors.Wrap() to generate a call stack for the error.
* If the error comes from another Fabric function, add further context, if desired, to the error message using errors.WithMessage() while leaving the call stack unaffected.
* A panic should not be allowed to propagate to other packages.

## Example program

The following example program provides a clear demonstration of using the package:

package main

**import** (

"fmt"

"github.com/pkg/errors"

)

func wrapWithStack() error {

err :**=** createError()

**//** do this when error comes **from** external source (go lib **or** vendor)

**return** errors**.**Wrap(err, "wrapping an error with stack")

}

func wrapWithoutStack() error {

err :**=** createError()

**//** do this when error comes **from** internal Fabric since it already has stack trace

**return** errors**.**WithMessage(err, "wrapping an error without stack")

}

func createError() error {

**return** errors**.**New("original error")

}

func main() {

err :**=** createError()

fmt**.**Printf("print error without stack: %s\n\n", err)

fmt**.**Printf("print error with stack: %+v\n\n", err)

err **=** wrapWithoutStack()

fmt**.**Printf("%+v\n\n", err)

err **=** wrapWithStack()

fmt**.**Printf("%+v\n\n", err)

}

# Logging Control

## Overview

Logging in the peer and orderer is provided by the common/flogging package. Chaincodes written in Go also use this package if they use the logging methods provided by the shim. This package supports

* Logging control based on the severity of the message
* Logging control based on the software logger generating the message
* Different pretty-printing options based on the severity of the message

All logs are currently directed to stderr. Global and logger-level control of logging by severity is provided for both users and developers. There are currently no formalized rules for the types of information provided at each severity level. When submitting bug reports, developers may want to see [full](https://hyperledger-fabric.readthedocs.io/en/release-1.4/logging-control.html) logs down to the DEBUG level.

In pretty-printed logs the logging level is indicated both by color and by a four-character code, e.g, “ERRO” for ERROR, “DEBU” for DEBUG, etc. In the logging context a logger is an arbitrary name (string) given by developers to groups of related messages. In the pretty-printed example below, the loggers ledgermgmt, kvledger, and peer are generating logs.

2018**-**11**-**01 15:32:38.268 UTC [ledgermgmt] initialize **->** INFO 002 Initializing ledger mgmt

2018**-**11**-**01 15:32:38.268 UTC [kvledger] NewProvider **->** INFO 003 Initializing ledger provider

2018**-**11**-**01 15:32:38.342 UTC [kvledger] NewProvider **->** INFO 004 ledger provider Initialized

2018**-**11**-**01 15:32:38.357 UTC [ledgermgmt] initialize **->** INFO 005 ledger mgmt initialized

2018**-**11**-**01 15:32:38.357 UTC [peer] func1 **->** INFO 006 Auto**-**detected peer address: 172.24**.**0.3:7051

2018**-**11**-**01 15:32:38.357 UTC [peer] func1 **->** INFO 007 Returning peer0**.**org1**.**example**.**com:7051

An arbitrary number of loggers can be created at runtime, therefore there is no “master list” of loggers, and logging control constructs can not check whether logging loggers actually do or will exist.

## Logging specification

The logging levels of the peer and orderer commands are controlled by a logging specification, which is set via the FABRIC\_LOGGING\_SPEC environment variable.

The full logging level specification is of the form

[**<**logger**>**[,**<**logger**>...**]**=**]**<**level**>**[:[**<**logger**>**[,**<**logger**>...**]**=**]**<**level**>...**]

Logging severity levels are specified using case-insensitive strings chosen from

FATAL **|** PANIC **|** ERROR **|** WARNING **|** INFO **|** DEBUG

A logging level by itself is taken as the overall default. Otherwise, overrides for individual or groups of loggers can be specified using the

**<**logger**>**[,**<**logger**>...**]**=<**level**>**

syntax. Examples of specifications:

info **-** Set default to INFO

warning:msp,gossip**=**warning:chaincode**=**info **-** Default WARNING; Override **for** msp, gossip, **and** chaincode

chaincode**=**info:msp,gossip**=**warning:warning **-** Same **as** above

## Logging format

The logging format of the peer and orderer commands is controlled via the FABRIC\_LOGGING\_FORMAT environment variable. This can be set to a format string, such as the default

"%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{shortfunc} -> %{level:.4s} %{id:03x}%{color:reset} %{message}"

to print the logs in a human-readable console format. It can be also set to json to output logs in JSON format.

## Go chaincodes

The standard mechanism to log within a chaincode application is to integrate with the logging transport exposed to each chaincode instance via the peer. The chaincode shim package provides APIs that allow a chaincode to create and manage logging objects whose logs will be formatted and interleaved consistently with the shim logs.

As independently executed programs, user-provided chaincodes may technically also produce output on stdout/stderr. While naturally useful for “devmode”, these channels are normally disabled on a production network to mitigate abuse from broken or malicious code. However, it is possible to enable this output even for peer-managed containers (e.g. “netmode”) on a per-peer basis via the CORE\_VM\_DOCKER\_ATTACHSTDOUT=true configuration option.

Once enabled, each chaincode will receive its own logging channel keyed by its container-id. Any output written to either stdout or stderr will be integrated with the peer’s log on a per-line basis. It is not recommended to enable this for production.

### API

NewLogger(name string) \*ChaincodeLogger - Create a logging object for use by a chaincode

(c \*ChaincodeLogger) SetLevel(level LoggingLevel) - Set the logging level of the logger

(c \*ChaincodeLogger) IsEnabledFor(level LoggingLevel) bool - Return true if logs will be generated at the given level

LogLevel(levelString string) (LoggingLevel, error) - Convert a string to a LoggingLevel

A LoggingLevel is a member of the enumeration

LogDebug, LogInfo, LogNotice, LogWarning, LogError, LogCritical

which can be used directly, or generated by passing a case-insensitive version of the strings

DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL

to the LogLevel API.

Formatted logging at various severity levels is provided by the functions

(c **\***ChaincodeLogger) Debug(args **...**interface{})

(c **\***ChaincodeLogger) Info(args **...**interface{})

(c **\***ChaincodeLogger) Notice(args **...**interface{})

(c **\***ChaincodeLogger) **Warning**(args **...**interface{})

(c **\***ChaincodeLogger) Error(args **...**interface{})

(c **\***ChaincodeLogger) Critical(args **...**interface{})

(c **\***ChaincodeLogger) Debugf(format string, args **...**interface{})

(c **\***ChaincodeLogger) Infof(format string, args **...**interface{})

(c **\***ChaincodeLogger) Noticef(format string, args **...**interface{})

(c **\***ChaincodeLogger) Warningf(format string, args **...**interface{})

(c **\***ChaincodeLogger) Errorf(format string, args **...**interface{})

(c **\***ChaincodeLogger) Criticalf(format string, args **...**interface{})

The f forms of the logging APIs provide for precise control over the formatting of the logs. The non-f forms of the APIs currently insert a space between the printed representations of the arguments, and arbitrarily choose the formats to use.

In the current implementation, the logs produced by the shim and a ChaincodeLogger are timestamped, marked with the logger name and severity level, and written to stderr. Note that logging level control is currently based on the name provided when the ChaincodeLogger is created. To avoid ambiguities, all ChaincodeLogger should be given unique names other than “shim”. The logger name will appear in all log messages created by the logger. The shim logs as “shim”.

The default logging level for loggers within the Chaincode container can be set in the [core.yaml](https://github.com/hyperledger/fabric/blob/master/sampleconfig/core.yaml) file. The key chaincode.logging.level sets the default level for all loggers within the Chaincode container. The key chaincode.logging.shim overrides the default level for the shim logger.

*# Logging section for the chaincode container*

logging:

*# Default level for all loggers within the chaincode container*

level: info

*# Override default level for the 'shim' logger*

shim: warning

The default logging level can be overridden by using environment variables. CORE\_CHAINCODE\_LOGGING\_LEVEL sets the default logging level for all loggers. CORE\_CHAINCODE\_LOGGING\_SHIM overrides the level for the shim logger.

Go language chaincodes can also control the logging level of the chaincode shim interface through the SetLoggingLevel API.

SetLoggingLevel(LoggingLevel level) - Control the logging level of the shim

Below is a simple example of how a chaincode might create a private logging object logging at the LogInfo level.

var logger **=** shim**.**NewLogger("myChaincode")

func main() {

logger**.**SetLevel(shim**.**LogInfo)

**...**

}

# Securing Communication With Transport Layer Security (TLS)

Fabric supports for secure communication between nodes using TLS. TLS communication can use both one-way (server only) and two-way (server and client) authentication.

## Configuring TLS for peers nodes

A peer node is both a TLS server and a TLS client. It is the former when another peer node, application, or the CLI makes a connection to it and the latter when it makes a connection to another peer node or orderer.

To enable TLS on a peer node set the following peer configuration properties:

* peer.tls.enabled = true
* peer.tls.cert.file = fully qualified path of the file that contains the TLS server certificate
* peer.tls.key.file = fully qualified path of the file that contains the TLS server private key
* peer.tls.rootcert.file = fully qualified path of the file that contains the certificate chain of the certificate authority(CA) that issued TLS server certificate

By default, TLS client authentication is turned off when TLS is enabled on a peer node. This means that the peer node will not verify the certificate of a client (another peer node, application, or the CLI) during a TLS handshake. To enable TLS client authentication on a peer node, set the peer configuration property peer.tls.clientAuthRequired to true and set the peer.tls.clientRootCAs.files property to the CA chain file(s) that contain(s) the CA certificate chain(s) that issued TLS certificates for your organization’s clients.

By default, a peer node will use the same certificate and private key pair when acting as a TLS server and client. To use a different certificate and private key pair for the client side, set the peer.tls.clientCert.file and peer.tls.clientKey.file configuration properties to the fully qualified path of the client certificate and key file, respectively.

TLS with client authentication can also be enabled by setting the following environment variables:

* CORE\_PEER\_TLS\_ENABLED = true
* CORE\_PEER\_TLS\_CERT\_FILE = fully qualified path of the server certificate
* CORE\_PEER\_TLS\_KEY\_FILE = fully qualified path of the server private key
* CORE\_PEER\_TLS\_ROOTCERT\_FILE = fully qualified path of the CA chain file
* CORE\_PEER\_TLS\_CLIENTAUTHREQUIRED = true
* CORE\_PEER\_TLS\_CLIENTROOTCAS\_FILES = fully qualified path of the CA chain file
* CORE\_PEER\_TLS\_CLIENTCERT\_FILE = fully qualified path of the client certificate
* CORE\_PEER\_TLS\_CLIENTKEY\_FILE = fully qualified path of the client key

When client authentication is enabled on a peer node, a client is required to send its certificate during a TLS handshake. If the client does not send its certificate, the handshake will fail and the peer will close the connection.

When a peer joins a channel, root CA certificate chains of the channel members are read from the config block of the channel and are added to the TLS client and server root CAs data structure. So, peer to peer communication, peer to orderer communication should work seamlessly.

## Configuring TLS for orderer nodes

To enable TLS on an orderer node, set the following orderer configuration properties:

* General.TLS.Enabled = true
* General.TLS.PrivateKey = fully qualified path of the file that contains the server private key
* General.TLS.Certificate = fully qualified path of the file that contains the server certificate
* General.TLS.RootCAs = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate

By default, TLS client authentication is turned off on orderer, as is the case with peer. To enable TLS client authentication, set the following config properties:

* General.TLS.ClientAuthRequired = true
* General.TLS.ClientRootCAs = fully qualified path of the file that contains the certificate chain of the CA that issued the TLS server certificate

TLS with client authentication can also be enabled by setting the following environment variables:

* ORDERER\_GENERAL\_TLS\_ENABLED = true
* ORDERER\_GENERAL\_TLS\_PRIVATEKEY = fully qualified path of the file that contains the server private key
* ORDERER\_GENERAL\_TLS\_CERTIFICATE = fully qualified path of the file that contains the server certificate
* ORDERER\_GENERAL\_TLS\_ROOTCAS = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate
* ORDERER\_GENERAL\_TLS\_CLIENTAUTHREQUIRED = true
* ORDERER\_GENERAL\_TLS\_CLIENTROOTCAS = fully qualified path of the file that contains the certificate chain of the CA that issued TLS server certificate

## Configuring TLS for the peer CLI

The following environment variables must be set when running peer CLI commands against a TLS enabled peer node:

* CORE\_PEER\_TLS\_ENABLED = true
* CORE\_PEER\_TLS\_ROOTCERT\_FILE = fully qualified path of the file that contains cert chain of the CA that issued the TLS server cert

If TLS client authentication is also enabled on the remote server, the following variables must to be set in addition to those above:

* CORE\_PEER\_TLS\_CLIENTAUTHREQUIRED = true
* CORE\_PEER\_TLS\_CLIENTCERT\_FILE = fully qualified path of the client certificate
* CORE\_PEER\_TLS\_CLIENTKEY\_FILE = fully qualified path of the client private key

When running a command that connects to orderer service, like peer channel <create|update|fetch> or peer chaincode <invoke|instantiate>, following command line arguments must also be specified if TLS is enabled on the orderer:

* –tls
* –cafile <fully qualified path of the file that contains cert chain of the orderer CA>

If TLS client authentication is enabled on the orderer, the following arguments must be specified as well:

* –clientauth
* –keyfile <fully qualified path of the file that contains the client private key>
* –certfile <fully qualified path of the file that contains the client certificate>

## Debugging TLS issues

Before debugging TLS issues, it is advisable to enable GRPC debug on both the TLS client and the server side to get additional information. To enable GRPC debug, set the environment variable FABRIC\_LOGGING\_SPEC to include grpc=debug. For example, to set the default logging level to INFO and the GRPC logging level to DEBUG, set the logging specification to grpc=debug:info.

If you see the error message remote error: tls: bad certificate on the client side, it usually means that the TLS server has enabled client authentication and the server either did not receive the correct client certificate or it received a client certificate that it does not trust. Make sure the client is sending its certificate and that it has been signed by one of the CA certificates trusted by the peer or orderer node.

If you see the error message remote error: tls: bad certificate in your chaincode logs, ensure that your chaincode has been built using the chaincode shim provided with Fabric v1.1 or newer. If your chaincode does not contain a vendored copy of the shim, deleting the chaincode container and restarting its peer will rebuild the chaincode container using the current shim version.

# Configuring and operating a Raft ordering service

**Audience**: Raft ordering node admins

## Conceptual overview

For a high level overview of the concept of ordering and how the supported ordering service implementations (including Raft) work at a high level, check out our conceptual documentation on the [Ordering Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html).

To learn about the process of setting up an ordering node — including the creation of a local MSP and the creation of a genesis block — check out our documentation on [Setting up an ordering node](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer_deploy.html).

## Configuration

While every Raft node must be added to the system channel, a node does not need to be added to every application channel. Additionally, you can remove and add a node from a channel dynamically without affecting the other nodes, a process described in the Reconfiguration section below.

Raft nodes identify each other using TLS pinning, so in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html) to impersonate a Raft node, an attacker needs to obtain the **private key** of its TLS certificate. As a result, it is not possible to run a Raft node without a valid TLS configuration.

A Raft cluster is configured in two planes:

* **Local configuration**: Governs node specific aspects, such as TLS communication, replication behavior, and file storage.
* **Channel configuration**: Defines the membership of the Raft cluster for the corresponding channel, as well as protocol specific parameters such as heartbeat frequency, leader timeouts, and more.

Recall, each channel has its own instance of a Raft protocol running. Thus, a Raft node must be referenced in the configuration of each channel it belongs to by adding its server and client TLS certificates (in PEM format) to the channel config. This ensures that when other nodes receive a message from it, they can securely confirm the identity of the node that sent the message.

The following section from configtx.yaml shows three Raft nodes (also called “consenters”) in the channel:

Consenters:

**-** Host: raft0**.**example**.**com

Port: 7050

ClientTLSCert: path**/**to**/**ClientTLSCert0

ServerTLSCert: path**/**to**/**ServerTLSCert0

**-** Host: raft1**.**example**.**com

Port: 7050

ClientTLSCert: path**/**to**/**ClientTLSCert1

ServerTLSCert: path**/**to**/**ServerTLSCert1

**-** Host: raft2**.**example**.**com

Port: 7050

ClientTLSCert: path**/**to**/**ClientTLSCert2

ServerTLSCert: path**/**to**/**ServerTLSCert2

Note: an orderer will be listed as a consenter in the system channel as well as any application channels they’re joined to.

When the channel config block is created, the configtxgen tool reads the paths to the TLS certificates, and replaces the paths with the corresponding bytes of the certificates.

### Local configuration

The orderer.yaml has two configuration sections that are relevant for Raft orderers:

**Cluster**, which determines the TLS communication configuration. And **consensus**, which determines where Write Ahead Logs and Snapshots are stored.

**Cluster parameters:**

By default, the Raft service is running on the same gRPC server as the client facing server (which is used to send transactions or pull blocks), but it can be configured to have a separate gRPC server with a separate port.

This is useful for cases where you want TLS certificates issued by the organizational CAs, but used only by the cluster nodes to communicate among each other, and TLS certificates issued by a public TLS CA for the client facing API.

* ClientCertificate, ClientPrivateKey: The file path of the client TLS certificate and corresponding private key.
* ListenPort: The port the cluster listens on. If blank, the port is the same port as the orderer general port (general.listenPort)
* ListenAddress: The address the cluster service is listening on.
* ServerCertificate, ServerPrivateKey: The TLS server certificate key pair which is used when the cluster service is running on a separate gRPC server (different port).
* SendBufferSize: Regulates the number of messages in the egress buffer.

Note: ListenPort, ListenAddress, ServerCertificate, ServerPrivateKey must be either set together or unset together. If they are unset, they are inherited from the general TLS section, for example general.tls.{privateKey, certificate}.

There are also hidden configuration parameters for general.cluster which can be used to further fine tune the cluster communication or replication mechanisms:

* DialTimeout, RPCTimeout: Specify the timeouts of creating connections and establishing streams.
* ReplicationBufferSize: the maximum [number](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html) of bytes that can be allocated for each in-memory buffer used for block replication from other cluster nodes. Each channel has its own memory buffer. Defaults to 20971520 which is 20MB.
* PullTimeout: the maximum duration the ordering node will wait for a block to be received before it aborts. Defaults to five seconds.
* ReplicationRetryTimeout: The maximum duration the ordering node will wait between two consecutive attempts. Defaults to five seconds.
* ReplicationBackgroundRefreshInterval: the time between two consecutive attempts to replicate existing channels that this node was added to, or channels that this node failed to replicate in the past. Defaults to five minutes.
* TLSHandshakeTimeShift: If the TLS certificates of the ordering nodes expire and are not replaced in time (see TLS certificate rotation below), communication between them cannot be established, and it will be impossible to send new transactions to the ordering service. To recover from such a scenario, it is possible to make TLS handshakes between ordering nodes consider the time to be shifted backwards a given amount that is configured to TLSHandshakeTimeShift. It only effects ordering nodes that use a separate gRPC server for their intra-cluster communication (via general.cluster.ListenPort and general.cluster.ListenAddress).

**Consensus parameters:**

* WALDir: the location at which Write Ahead Logs for etcd/raft are stored. Each channel will have its own subdirectory named after the channel ID.
* SnapDir: specifies the location at which snapshots for etcd/raft are stored. Each channel will have its own subdirectory named after the channel ID.

There is also a hidden configuration parameter that can be set by adding it to the consensus section in the orderer.yaml:

* EvictionSuspicion: The cumulative period of time of channel eviction suspicion that triggers the node to pull blocks from other nodes and see if it has been evicted from the channel in order to confirm its suspicion. If the suspicion is confirmed (the inspected block doesn’t contain the node’s TLS certificate), the node halts its operation for that channel. A node suspects its channel eviction when it doesn’t know about any elected leader nor can be elected as leader in the channel. Defaults to 10 minutes.

### Channel configuration

Apart from the (already discussed) consenters, the Raft channel configuration has an Options section which relates to protocol specific knobs. It is currently not possible to change these values dynamically while a node is running. The node have to be reconfigured and restarted.

The only exceptions is SnapshotIntervalSize, which can be adjusted at runtime.

Note: It is recommended to avoid changing the following values, as a misconfiguration might lead to a state where a leader cannot be elected at all (i.e, if the TickInterval and ElectionTick are extremely low). Situations where a leader cannot be elected are impossible to resolve, as leaders are required to make changes. Because of such dangers, we suggest not tuning these parameters for most use cases.

* TickInterval: The time interval between two Node.Tick invocations.
* ElectionTick: The number of Node.Tick invocations that must pass between elections. That is, if a follower does not receive any message from the leader of current term before ElectionTick has elapsed, it will become candidate and start an election.
* ElectionTick must be greater than HeartbeatTick.
* HeartbeatTick: The number of Node.Tick invocations that must pass between heartbeats. That is, a leader sends heartbeat messages to maintain its leadership every HeartbeatTick ticks.
* MaxInflightBlocks: Limits the max number of in-flight append blocks during optimistic replication phase.
* SnapshotIntervalSize: Defines number of bytes per which a snapshot is taken.

## Reconfiguration

The Raft orderer supports dynamic (meaning, while the channel is being serviced) addition and removal of nodes as long as only one node is added or removed at a time. Note that your cluster must be operational and able to achieve consensus before you attempt to reconfigure it. For instance, if you have three nodes, and two nodes fail, you will not be able to reconfigure your cluster to remove those nodes. Similarly, if you have one failed node in a channel with three nodes, you should not attempt to rotate a certificate, as this would induce a second fault. As a rule, you should never attempt any configuration changes to the Raft consenters, such as adding or removing a consenter, or rotating a consenter’s certificate unless all consenters are online and healthy.

If you do decide to change these parameters, it is recommended to only attempt such a change during a maintenance cycle. Problems are most likely to occur when a configuration is attempted in clusters with only a few nodes while a node is down. For example, if you have three nodes in your consenter set and one of them is down, it means you have two out of three nodes alive. If you extend the cluster to four nodes while in this state, you will have only two out of four nodes alive, which is not a quorum. The fourth node won’t be able to onboard because nodes can only onboard to functioning clusters (unless the total size of the cluster is one or two).

So by extending a cluster of three nodes to four nodes (while only two are alive) you are effectively stuck until the original offline node is resurrected.

Adding a new node to a Raft cluster is done by:

1. **Adding the TLS certificates** of the new node to the channel through a channel configuration update transaction. Note: the new node must be added to the system channel before being added to one or more application channels.
2. **Fetching the latest config block** of the system channel from an orderer node that’s part of the system channel.
3. **Ensuring that the node that will be added is part of the system channel** by checking that the config block that was fetched includes the certificate of (soon to be) added node.
4. **Starting the new Raft node** with the path to the config block in the General.GenesisFile configuration parameter.
5. **Waiting for the Raft node to replicate the blocks** from existing nodes for all channels its certificates have been added to. After this step has been completed, the node begins servicing the channel.
6. **Adding the endpoint** of the newly added Raft node to the channel configuration of all channels.

It is possible to add a node that is already running (and participates in some channels already) to a channel while the node itself is running. To do this, simply add the node’s certificate to the channel config of the channel. The node will autonomously detect its addition to the new channel (the default value here is five minutes, but if you want the node to detect the new channel more quickly, reboot the node) and will pull the channel blocks from an orderer in the channel, and then start the Raft instance for that chain.

After it has successfully done so, the channel configuration can be updated to include the endpoint of the new Raft orderer.

Removing a node from a Raft cluster is done by:

1. Removing its endpoint from the channel config for all channels, including the system channel controlled by the orderer admins.
2. Removing its entry (identified by its certificates) from the channel configuration for all channels. Again, this includes the system channel.
3. Shut down the node.

Removing a node from a specific channel, but keeping it servicing other channels is done by:

1. Removing its endpoint from the channel config for the channel.
2. Removing its entry (identified by its certificates) from the channel configuration.
3. The second phase causes:
   * The remaining orderer nodes in the channel to cease communicating with the removed orderer node in the context of the removed channel. They might still be communicating on other channels.
   * The node that is removed from the channel would autonomously detect its removal either immediately or after EvictionSuspicion time has passed (10 minutes by default) and will shut down its Raft instance.

### TLS certificate rotation for an orderer node

All TLS certificates have an expiration date that is determined by the issuer. These expiration dates can range from 10 years from the date of issuance to as little as a few months, so check with your issuer. Before the expiration date, you will need to rotate these certificates on the node itself and every channel the node is joined to, including the system channel.

For each channel the node participates in:

1. Update the channel configuration with the new certificates.
2. Replace its certificates in the file system of the node.
3. Restart the node.

Because a node can only have a single TLS certificate key pair, the node will be unable to service channels its new certificates have not been added to during the update process, degrading the capacity of fault tolerance. Because of this, **once the certificate rotation process has been started, it should be completed as quickly as possible.**

If for some reason the rotation of the TLS certificates has started but cannot complete in all channels, it is advised to rotate TLS certificates back to what they were and attempt the rotation later.

## Metrics

For a description of the Operations Service and how to set it up, check out [our documentation on the Operations Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/operations_service.html).

For a list at the metrics that are gathered by the Operations Service, check out our [reference material on metrics](https://hyperledger-fabric.readthedocs.io/en/release-1.4/metrics_reference.html).

While the metrics you prioritize will have a lot to do with your particular use case and configuration, there are two metrics in particular you might want to monitor:

* consensus\_etcdraft\_is\_leader: identifies which node in the cluster is currently leader. If no nodes have this set, you have lost quorum.
* consensus\_etcdraft\_data\_persist\_duration: indicates how long write operations to the Raft cluster’s persistent write ahead log take. For protocol safety, messages must be persisted durably, calling fsync where appropriate, before they can be shared with the consenter set. If this value begins to climb, this node may not be able to participate in consensus (which could lead to a service interruption for this node and possibly the network).

## Troubleshooting

* The more stress you put on your nodes, the more you might have to change certain parameters. As with any system, computer or mechanical, stress can lead to a drag in performance. As we noted in the conceptual documentation, leader elections in Raft are triggered when follower nodes do not receive either a “heartbeat” messages or an “append” message that carries data from the leader for a certain amount of time. Because Raft nodes share the same communication layer across channels (this does not mean they share data — they do not!), if a Raft node is part of the consenter set in many channels, you might want to lengthen the amount of time it takes to trigger an election to avoid inadvertent leader elections.

# Migrating from Kafka to Raft

**Note: this**[**document**](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html)**presumes a high degree of expertise with channel configuration update transactions. As the process for migration involves several channel configuration update transactions, do not attempt to migrate from Kafka to Raft without first familiarizing yourself with the**[**Add an Organization to a Channel**](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html)**tutorial, which describes the channel update process in detail.**

For users who want to transition channels from using Kafka-based ordering services to [Raft-based](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html#Raft) ordering services, v1.4.2 allows this to be accomplished through a series of configuration update transactions on each channel in the network.

This tutorial will describe this process at a high level, calling out specific [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html) where necessary, rather than show each command in detail.

## Assumptions and considerations

Before attempting migration, take the following into account:

1. This process is solely for migration from Kafka to Raft. Migrating between any other orderer consensus types is not currently supported.
2. Migration is one way. Once the ordering service is migrated to Raft, and starts committing transactions, it is not possible to go back to Kafka.
3. Because the ordering nodes must go down and be brought back up, downtime must be allowed during the migration.
4. Recovering from a botched migration is possible only if a backup is taken at the point in migration prescribed later in this document. If you do not take a backup, and migration fails, you will not be able to recover your previous state.
5. All channels must be migrated during the same maintenance window. It is not possible to migrate only some channels before resuming operations.
6. At the end of the migration process, every channel will have the same consenter set of Raft nodes. This is the same consenter set that will exist in the ordering system channel. This makes it possible to diagnose a successful migration.
7. Migration is done in place, utilizing the existing ledgers for the deployed ordering nodes. Addition or removal of orderers should be performed after the migration.

## High level migration flow

Migration is carried out in five phases.

1. The system is placed into a maintenance mode where application transactions are rejected and only ordering service admins can make changes to the channel configuration.
2. The system is stopped, and a backup is taken in case an error occurs during migration.
3. The system is started, and each channel has its consensus type and metadata modified.
4. The system is restarted and is now operating on Raft consensus; each channel is checked to confirm that it has successfully achieved a quorum.
5. The system is moved out of maintenance mode and normal function resumes.

## Preparing to migrate

There are several steps you should take before attempting to migrate.

* Design the Raft deployment, deciding which ordering service nodes are going to remain as Raft consenters. You should deploy at least three ordering nodes in your cluster, but note that deploying a consenter set of at least five nodes will maintain high availability should a node goes down, whereas a three node configuration will lose high availability once a single node goes down for any reason (for example, as during a maintenance cycle).
* Prepare the material for building the Raft Metadata configuration. **Note: all the channels should receive the same Raft Metadata configuration**. Refer to the [Raft configuration guide](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html) for more information on these fields. Note: you may find it easiest to bootstrap a new ordering network with the Raft consensus protocol, then copy and modify the consensus metadata section from its config. In any case, you will need (for each ordering node):
  + hostname
  + port
  + server certificate
  + client certificate
* Compile a list of all channels (system and application) in the system. Make sure you have the correct credentials to [sign](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html) the configuration updates. For example, the relevant ordering service admin identities.
* Ensure all ordering service nodes are running the same version of Fabric, and that this version is v1.4.2 or greater.
* Ensure all peers are running at least v1.4.2 of Fabric. Make sure all channels are configured with the channel capability that enables migration.
  + Orderer capability V1\_4\_2 (or above).
  + Channel capability V1\_4\_2 (or above).

### Entry to maintenance mode

Prior to setting the ordering service into maintenance mode, it is recommended that the peers and clients of the network be stopped. Leaving peers or clients up and running is safe, however, because the ordering service will reject all of their requests, their logs will fill with benign but misleading failures.

Follow the process in the [Add an Organization to a Channel](https://hyperledger-fabric.readthedocs.io/en/release-1.4/channel_update_tutorial.html) tutorial to pull, translate, and scope the configuration of **each channel, starting with the system channel**. The only field you should change during this step is in the channel configuration at /Channel/Orderer/ConsensusType. In a JSON representation of the channel configuration, this would be .channel\_group.groups.Orderer.values.ConsensusType​.

The ConsensusType is represented by three values: Type, Metadata, and State, where:

* Type is either kafka or etcdraft (Raft). This value can only be changed while in maintenance mode.
* Metadata will be empty if the Type is kafka, but must carry valid Raft metadata if the ConsensusType is etcdraft. More on this below.
* State is either STATE\_NORMAL, when the channel is processing transactions, or STATE\_MAINTENANCE, during the migration process.

In the first step of the channel configuration update, only change the State from STATE\_NORMAL to STATE\_MAINTENANCE. Do not change the Type or the Metadata field yet. Note that the Type should currently be kafka.

While in maintenance mode, normal transactions, config updates unrelated to migration, and Deliver requests from the peers used to retrieve new blocks are rejected. This is done in order to prevent the need to both backup, and if necessary restore, peers during migration, as they only receive updates once migration has successfully completed. In other words, we want to keep the ordering service backup point, which is the next step, ahead of the peer’s ledger, in [order](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka_raft_migration.html) to be able to perform rollback if needed. However, ordering node admins can issue Deliver requests (which they need to be able to do in order to continue the migration process).

**Verify** that each ordering service node has entered maintenance mode on each of the channels. This can be done by fetching the last config block and making sure that the Type, Metadata, State on each channel is kafka, empty (recall that there is no metadata for Kafka), and STATE\_MAINTENANCE, respectively.

If the channels have been updated successfully, the ordering service is now ready for backup.

#### Backup files and shut down servers

Shut down all ordering nodes, Kafka servers, and Zookeeper servers. It is important to **shutdown the ordering service nodes first**. Then, after allowing the Kafka service to flush its logs to disk (this typically takes about 30 seconds, but might take longer depending on your system), the Kafka servers should be shut down. Shutting down the Kafka brokers at the same time as the orderers can result in the filesystem state of the orderers being more recent than the Kafka brokers which could prevent your network from starting.

Create a backup of the file system of these servers. Then restart the Kafka service and then the ordering service nodes.

### Switch to Raft in maintenance mode

The next step in the migration process is another channel configuration update for each channel. In this configuration update, switch the Type to etcdraft (for Raft) while keeping the State in STATE\_MAINTENANCE, and fill in the Metadata configuration​. It is highly recommended that the Metadata configuration​ be identical​ on all channels. If you want to establish different consenter sets with different nodes, you will be able to reconfigure the Metadata configuration​after the system is restarted into etcdraft mode. Supplying an identical metadata object, and hence, an identical consenter set, means that when the nodes are restarted, if the system channel forms a quorum and can exit maintenance mode, other channels will likely be able do the same. Supplying different consenter sets to each channel can cause one channel to succeed in forming a cluster while another channel will fail.

Then, validate that each ordering service node has committed the ConsensusType change configuration update by pulling and inspecting the configuration of each channel.

Note: For each channel, the transaction that changes the ConsensusType must be the last configuration transaction before restarting the nodes (in the next step). If some other configuration transaction happens after this step, the nodes will most likely crash on restart, or result in undefined behavior.

#### Restart and validate leader

Note: exit of maintenance mode **must** be done **after** restart.

After the ConsensusType update has been completed on each channel, stop all ordering service nodes, stop all Kafka brokers and Zookeepers, and then restart only the ordering service nodes. They should restart as Raft nodes, form a cluster per channel, and elect a leader on each channel.

**Note**: Since Raft-based ordering service requires mutual TLS between orderer nodes, **additional configurations** are required before you start them again, see [Section: Local Configuration](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.md#local-configuration) for more details.

After restart process finished, make sure to **validate** that a leader has been elected on each channel by inspecting the node logs (you can see what to look for below). This will confirm that the process has been completed successfully.

When a leader is elected, the log will show, for each channel:

"Raft leader changed: 0 -> ​node-number​ ​channel=​channel-name​

node=​node-number​ ​"

For example:

2019**-**05**-**26 10:07:44.075 UTC [orderer**.**consensus**.**etcdraft] serveRequest **->**

INFO 047 Raft leader changed: 0 **->** 1 channel**=**testchannel1 node**=**2

In this example ​node 2​ reports that a leader was elected (the leader is​node 1​) by the cluster of channel ​testchannel1​.

### Switch out of maintenance mode

Perform another channel configuration update on each channel (sending the config update to the same ordering node you have been sending configuration updates to until now), switching the State from STATE\_MAINTENANCE to STATE\_NORMAL. Start with the system channel, as usual. If it succeeds on the ordering system channel, migration is likely to succeed on all channels. To verify, fetch the last config block of the system channel from the ordering node, verifying that the State is now STATE\_NORMAL. For completeness, verify this on each ordering node.

When this process is completed, the ordering service is now ready to accept all transactions on all channels. If you stopped your peers and application as recommended, you may now restart them.

## Abort and rollback

If a problem emerges during the migration process **before exiting maintenance mode**, simply perform the rollback procedure below.

1. Shut down the ordering nodes and the Kafka service (servers and Zookeeper ensemble).
2. Rollback the file system of these servers to the backup taken at maintenance mode before changing the ConsensusType.
3. Restart said servers, the ordering nodes will bootstrap to Kafka in maintenance mode.
4. Send a configuration update exiting maintenance mode to continue using Kafka as your consensus mechanism, or resume the instructions after the point of backup and fix the error which prevented a Raft quorum from forming and retry migration with corrected Raft configuration Metadata.

There are a few states which might indicate migration has failed:

1. Some nodes crash or shutdown.
2. There is no record of a successful leader election per channel in the logs.
3. The attempt to flip to STATE\_NORMAL mode on the system channel fails.

# Bringing up a Kafka-based Ordering Service

## Caveat emptor

This [document](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html) assumes that the reader knows how to set up a Kafka cluster and a ZooKeeper ensemble, and keep them secure for general usage by preventing unauthorized access. The sole purpose of this guide is to identify the steps you need to take so as to have a set of Hyperledger Fabric ordering service nodes (OSNs) use your Kafka cluster and provide an ordering service to your blockchain network.

For information about the role orderers play in a network and in a transaction flow, [checkout](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html) our [The Ordering Service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer/ordering_service.html) documentation.

For information on how to set up an ordering node, check out our [Setting up an ordering node](https://hyperledger-fabric.readthedocs.io/en/release-1.4/orderer_deploy.html) documentation.

For information about configuring Raft ordering services, check out [Configuring and operating a Raft ordering service](https://hyperledger-fabric.readthedocs.io/en/release-1.4/raft_configuration.html).

## Big picture

Each channel maps to a separate single-partition topic in Kafka. When an OSN receives transactions via the Broadcast RPC, it checks to make sure that the broadcasting client has permissions to write on the channel, then relays (i.e. produces) those transactions to the appropriate partition in Kafka. This partition is also consumed by the OSN which groups the received transactions into blocks locally, persists them in its local ledger, and serves them to receiving clients via the Deliver RPC. For low-level [details](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html), refer to [the document that describes how we came to this design](https://docs.google.com/document/d/19JihmW-8blTzN99lAubOfseLUZqdrB6sBR0HsRgCAnY/edit). **Figure 8** is a schematic representation of the process described above.

## Steps

Let K and Z be the number of nodes in the Kafka cluster and the ZooKeeper ensemble respectively:

1. At a minimum, K should be set to 4. (As we will explain in Step 4 below, this is the minimum number of nodes necessary in order to exhibit crash fault tolerance, i.e. with 4 brokers, you can have 1 broker go down, all channels will continue to be writeable and readable, and new channels can be created.)
2. Z will either be 3, 5, or 7. It has to be an odd number to avoid split-brain scenarios, and larger than 1 in order to avoid single point of failures. Anything beyond 7 ZooKeeper servers is considered overkill.

Then proceed as follows:

1. Orderers: **Encode the Kafka-related information in the network’s genesis block.** If you are using configtxgen, edit configtx.yaml. Alternatively, pick a preset profile for the system channel’s genesis block— so that:
   * Orderer.OrdererType is set to kafka.
   * Orderer.Kafka.Brokers contains the address of at least two of the Kafka brokers in your cluster in IP:port notation. The list does not need to be exhaustive. (These are your bootstrap brokers.)
2. Orderers: **Set the maximum block size.** Each block will have at most Orderer.AbsoluteMaxBytes bytes (not including headers), a value that you can set in configtx.yaml. Let the value you pick here be A and make note of it —– it will affect how you configure your Kafka brokers in Step 6.
3. Orderers: [**Create**](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html)**the genesis block.** Use configtxgen. The settings you picked in Steps 3 and 4 above are system-wide settings, i.e. they apply across the network for all the OSNs. Make note of the genesis block’s location.
4. Kafka cluster: **Configure your Kafka brokers appropriately.** Ensure that every Kafka broker has these keys configured:
   * unclean.leader.election.enable = false — Data consistency is key in a blockchain environment. We cannot have a channel leader chosen outside of the in-sync replica set, or we run the risk of overwriting the offsets that the previous leader produced, and —as a result— rewrite the blockchain that the orderers produce.
   * min.insync.replicas = M — Where you pick a value M such that 1 < M < N (see default.replication.factor below). Data is considered committed when it is written to at least M replicas (which are then considered in-sync and belong to the in-sync replica set, or ISR). In any other case, the write operation returns an error. Then:
     + If up to N-M replicas —out of the N that the channel data is written to become unavailable, operations proceed normally.
     + If more replicas become unavailable, Kafka cannot maintain an ISR set of M, so it stops accepting writes. Reads work without issues. The channel becomes writeable again when M replicas get in-sync.
   * default.replication.factor = N — Where you pick a value N such that N < K. A replication factor of N means that each channel will have its data replicated to N brokers. These are the candidates for the ISR set of a channel. As we noted in the min.insync.replicas section above, not all of these brokers have to be available all the time. N should be set strictly smaller to K because channel creations cannot go forward if less than N brokers are up. So if you set N = K, a single broker going down means that no new channels can be created on the blockchain network — the crash fault tolerance of the ordering service is non-existent.

Based on what we’ve described above, the minimum allowed values for M and N are 2 and 3 respectively. This configuration allows for the creation of new channels to go forward, and for all channels to continue to be writeable.

* + message.max.bytes and replica.fetch.max.bytes should be set to a value larger than A, the value you picked in Orderer.AbsoluteMaxBytes in Step 4 above. Add some buffer to [account](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html) for headers —– 1 MiB is more than enough. The following condition applies:
  + Orderer**.**AbsoluteMaxBytes **<** replica**.**fetch**.**max**.**bytes **<=** message**.**max**.**bytes

(For completeness, we note that message.max.bytes should be strictly smaller to socket.request.max.bytes which is set by default to 100 MiB. If you wish to have blocks larger than 100 MiB you will need to edit the hard-coded value in brokerConfig.Producer.MaxMessageBytes in fabric/orderer/kafka/config.go and rebuild the binary from source. This is not advisable.)

* + log.retention.ms = -1. Until the ordering service adds support for pruning of the Kafka logs, you should disable time-based retention and prevent segments from expiring. (Size-based retention — see log.retention.bytes — is disabled by default in Kafka at the time of this writing, so there’s no need to set it explicitly.)

1. Orderers: **Point each OSN to the genesis block.**

Edit General.GenesisFile in orderer.yaml so that it points to the genesis block created in Step 5 above. While at it, ensure all other keys in that YAML file are set appropriately.

1. Orderers: **Adjust polling intervals and timeouts.** (Optional step.)
   * The Kafka.Retry section in the orderer.yaml file allows you to adjust the frequency of the metadata/producer/consumer requests, as well as the socket timeouts. (These are all settings you would expect to see in a Kafka producer or consumer.)
   * Additionally, when a new channel is created, or when an existing channel is reloaded (in case of a just-restarted orderer), the orderer interacts with the Kafka cluster in the following ways:
     + It creates a Kafka producer (writer) for the Kafka partition that corresponds to the channel. . It uses that producer to post a no-op CONNECT message to that partition. . It creates a Kafka consumer (reader) for that partition.
     + If any of these steps fail, you can adjust the frequency with which they are repeated. Specifically they will be re-attempted every Kafka.Retry.ShortInterval for a total of Kafka.Retry.ShortTotal, and then every Kafka.Retry.LongInterval for a total of Kafka.Retry.LongTotal until they succeed. Note that the orderer will be unable to write to or read from a channel until all of the steps above have been completed successfully.
2. **Set up the OSNs and Kafka cluster so that they communicate over SSL.** (Optional step, but highly recommended.) Refer to [the Confluent guide](https://docs.confluent.io/2.0.0/kafka/ssl.html) for the Kafka cluster side of the equation, and set the keys under Kafka.TLS in orderer.yaml on every OSN accordingly.
3. **Bring up the nodes in the following order: ZooKeeper ensemble, Kafka cluster, ordering service nodes.**

## Additional considerations

1. **Preferred message size.** In Step 4 above (see [Steps](https://hyperledger-fabric.readthedocs.io/en/release-1.4/kafka.html#steps) section) you can also set the preferred size of blocks by setting the Orderer.Batchsize.PreferredMaxBytes key. Kafka offers higher throughput when dealing with relatively small messages; aim for a value no bigger than 1 MiB.
2. **Using environment variables to override settings.** When using the sample Kafka and Zookeeper Docker images provided with Fabric (see images/kafka and images/zookeeper respectively), you can override a Kafka broker or a ZooKeeper server’s settings by using environment variables. Replace the dots of the configuration key with underscores. For example, KAFKA\_UNCLEAN\_LEADER\_ELECTION\_ENABLE=false will allow you to override the default value of unclean.leader.election.enable. The same applies to the OSNs for their local configuration, i.e. what can be set in orderer.yaml. For example ORDERER\_KAFKA\_RETRY\_SHORTINTERVAL=1s allows you to override the default value for Orderer.Kafka.Retry.ShortInterval.

## Kafka Protocol Version Compatibility

Fabric uses the [sarama client library](https://github.com/Shopify/sarama) and vendors a version of it that supports Kafka 0.10 to 1.0, yet is still known to work with older versions.

Using the Kafka.Version key in orderer.yaml, you can configure which version of the Kafka protocol is used to communicate with the Kafka cluster’s brokers. Kafka brokers are backward compatible with older protocol versions. Because of a Kafka broker’s backward compatibility with older protocol versions, upgrading your Kafka brokers to a new version does not require an update of the Kafka.Version key value, but the Kafka cluster might suffer a [performance penalty](https://kafka.apache.org/documentation/#upgrade_11_message_format) while using an older protocol version.

## Debugging

Set environment variable FABRIC\_LOGGING\_SPEC to DEBUG and set Kafka.Verbose to true in orderer.yaml .

<https://creativecommons.org/licenses/by/4.0/>