**ENDORSEMENT POLICIES**

**Hyperledger Composer supports the existing Hyperledger Fabric blockchain infrastructure and runtime, which supports pluggable blockchain consensus protocols to ensure that transactions are validated according to policy by the designated business network participants.**

**Consensus**

The process of keeping the ledger transactions synchronized across the network — to ensure that ledgers update only when transactions are approved by the appropriate participants, and that when ledgers do update, they update with the same transactions in the same order — is called consensus.

**Transaction lifecycle**

Hyperledger Fabric separating the process of executing smart contracts from the process of updating the ledger makes it possible to improve transaction throughput, support more granular privacy controls, and implement more flexible and powerful smart contracts. One of the key ingredients for making this possible is a system for explicitly endorsing transactions before they are added to the ledger.

|  |  |
| --- | --- |
| Other blockchain platforms:  **Order:** Transactions are added to the ledger in some order and disseminated to all peers.  **Execute:** Transactions are sequentially executed (e.g. using smart contract code) on all peers.  In order for all peers to end up with the same state, transactions must always execute deterministically — i.e. the same transaction must always create the same result, no matter where or when it is executed.  This requirement places strong limitations on what a smart contract can do, which is one reason why smart contracts often use a special domain-specific language. It’s generally not possible to enforce determinism in a general-purpose language (e.g. Java or Go). | In Hyperledger Fabric, the lifecycle of a transaction is different:  *Execute: Transactions are executed (using chaincode) in any order, possibly even in parallel.*  *Order: When enough peers agree on the results of a transaction, it’s added to the ledger and disseminated to all peers.* This step is where the transactions are first given an ordering — until transactions are added to the ledger, there’s no concept of one transaction happening before or after another.  Validate: Each peer validates and applies the ledger’s transactions in sequence. Now that the transactions have an ordering, the peers can check whether a later transaction was invalidated by an earlier transaction. For example, this prevents one item from being sold two times (called double-spending).  The first note to make here is that there’s a separation between the execution of a transaction (Execute step) and actually updating the ledger (Validate step). This separation has useful effects:  All peers need to update the ledger, so all peers do the Validate step. But not every peer needs to Execute the smart contract. Hyperledger Fabric uses endorsement policies to define which peers need to execute which transactions. This means that a given chaincode (smart contract) can be kept private from peers that aren’t part of the endorsement policy.  Transactions can be executed before they are put in order. This allows peers to execute transactions in parallel, which can improve throughput.  In Fabric’s three-step execute-order-validate model, the results of executing chaincode for a transaction are explicitly agreed upon (according to the endorsement policy) before the transaction is added to the ledger. The two-step order-execute model uses deterministic “chaincode”, which implies that peers will agree on the result of executing a smart contract. Making agreement explicit allows Fabric to use non-deterministic chaincode, which is why you can write Fabric chaincode in Go, Java, and Node.js. |

**How Endorsement Works**

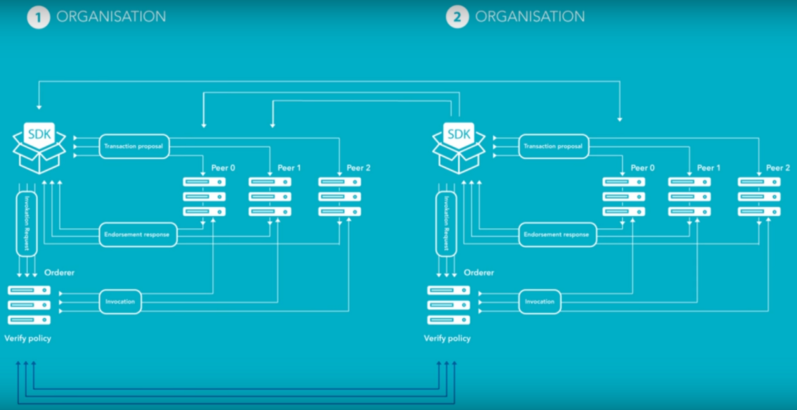
In the order-execute model, the concepts of executing chaincode and updating the ledger are combined into one idea — transaction.

In Fabric, these two concepts are separated, so the idea of transaction is split as well.

Fabric starts with a transaction proposal. It’s a bundle of information used to trigger a specific chaincode. The transaction proposal is sent to some peers for endorsement. An endorsing peer executes the chaincode, which (if it succeeds) yields an actual transaction for the ledger. The endorsing peer then signs the transaction and returns it to the proposer. This is the Execute step in execute-order-validate.

Once the creator of the proposal receives enough signatures to satisfy the endorsement policy, it can submit the transaction (and the signatures) to be added to the ledger. This is the Order step.

**Transaction Endorsement Workflow**



In the above block diagram, we have Two Organizations that are actually part of the same network and they are part of the same Channel(channels can be referred as a data partitioning mechanism that allow transaction visibility for stakeholders only, each channel is an independent chain of transaction blocks containing only transactions for that particular channel). Both Organizations i.e. Organization 1 and Organization 2 have 3 Peers each (i.e. Peer 0, Peer 1, and Peer 2) and are having their own individual Ordering Service respectively. Let us assume that they are configured, they are in the same channel, everything is working and they can see each other on the network level and they can exchange data and are in synchronization.

Now, If you see, we have an SDK i.e. it can be Nodejs, Java, Go, Python which ever you can select that doesn’t matter. Now, let’s start executing some operations, say, for example, we have to change the ownership of some assets. So, for that we need to create the proper transaction, we need to take the certificate from the Certificate Authority(CA), we need to embed proper attributes inside the certificate and also provide the proper arguments for the transaction (if required).

So, to summarize (Considering a use-case of single Organization in the network):-

→ SDK creates a Transaction Proposal and forwards it to the all the Peers in the network.

→ Now, the Peers accept the Transaction Proposal request and do Simulation i.e. takes the current state of every single Peer because the state must be the same and also the result must be the same.

→The output of the Simulation will be a read/write set.

→Peer actually take the this (say some updated key) result, cryptographically sign it and do a lot of other stuff and send back the Endorsement response to the SDK.

→ SDK then generates an Invocation Request and forwards it to the Orderer for verification.

→The Ordering Service takes the Invocation Request from the SDK, do the required verification’s, order them and then send this order with all the read/write set to the Peers and they will append the Ledger in the same order.

So, let us, try to understand, **Double Spending problem** with a simple example. Let say, I have 10 bucks in my account and I want to send them to Alice. So, I am executing an transaction operation to send 10 bucks to Alice, but, at the same moment, that’s exactly the same moment, I am executing the same transaction operation, but to send 10 bucks to Bob. So, it is possible because of the concurrency of the transaction operation, i.e. 10 bucks will go the Alice and 10 bucks will go to Bob. This happens because, while the money is transferred to Alice, they are not removed from my account. And when the verification is done to check, Do I have that money to send to Bob? The answer will be, “Yes”, but later, when the transaction operation to Alice is executed, 10 bucks will be deducted from my account and later when Bob’s transaction operation is executed, again, 10 bucks will be removed from my account resulting -10 (i.e. minus 10) bucks in my account balance. Well, This is a really simplified version of the Double Spending problem, but it is a really serious problem in the concurrent and parallel system.

Important Points to Noted —

→ Here, **Orderer** plays a very important role and Orderer is responsible for deciding which Transaction Request is to be executed first. As you can see in the block diagram, both the Organizations i.e. Org 1 and Org 2 have their own individual Ordering Services. This is the proper way, how we have to do it in production, because if the Orderer in Org 2 is compromised, then their will be a mismatch between these two Orderers and the Invocation Request will not be executed. If both of the Orderers are compromised then it is impossible to bring the same result in output. So, every single Organization has its own Ordering Service and its taking care that its own Ordering Service in not compromised.

→ It is okay, to have a single Orderer, when we are developing locally or in testing environment, but never proceed with a single Orderer in production environment, because this will be a single point of failure. If the Ordering Service disappears for some reason entire network will stop working and probably we will loose some information. And if there is any information that is not yet committed to the blockchain, then, this information will be lost(it is kind of similar to shutting down of our computer without saving our file).

**Endorsement Policy**

- define which peers need to agree/endorse on the results of a transaction before it can be added to the ledger. Fabric includes a small domain-specific language for specifying endorsement policies. Example endorsement policies might be:

- The Hyperledger Fabric Endorsement policy is configured when the chaincode is instantiated. The Endorsement policies are configured as logical gate predicates. The rule for this logic gate is as follows:

**EXPR(E[, E...]) where EXPR is AND or OR and E is either a principal (Org1.Admin, Org1.Member) or a nested expression EXPR(..)**

Example: "AND('Org1.member', 'Org2.member')" - means that both Org1 and Org2 members are requested to sign the transaction.

#### setEndorsementPolicyDefinition(policy)

# Hyperledger Fabric SDK for node.js **-** Class: Chaincode

#### This class allows an application to contain all chaincode attributes and artifacts in one place during runtime. This will assist the administration of the chaincode's lifecycle. From your [Client](https://fabric-sdk-node.github.io/master/Client.html) instance use the [Client#newChaincode](https://fabric-sdk-node.github.io/master/Client.html" \l "newChaincode) method. This will return a Chaincode object instance that has been associated with that client. This will provide access to user credentials used for signing requests, access to peer, orderer, and channel information.

Provide the endorsement policy definition for this chaincode. The input is a JSON object.

##### Parameters:

| **Name** | **Type** | **Description** |
| --- | --- | --- |
| policy | string|object | When the policy is a string it will be the canonical path to a policy in the Channel configuration. When an object, it will be the fabric-client's JSON representation of an fabric endorsement policy. |

##### Examples

Object Endorsement policy: "Signed by any member from one of the organizations"

*{*

*identities: [*

*{ role: {name: "member", mspId: "org1"}},*

*{ role: {name: "member", mspId: "org2"}}*

*],*

*policy: {*

*"1-of": [{"signed-by": 0}, {"signed-by": 1}]*

*}*

*}*

Object Endorsement policy: "Signed by admin of the ordererOrg and any member from one of the peer organizations"

*{*

*identities: [*

*{role: {name: "member", mspId: "peerOrg1"}},*

*{role: {name: "member", mspId: "peerOrg2"}},*

*{role: {name: "admin", mspId: "ordererOrg"}}*

*],*

*policy: {*

*"2-of": [*

*{"signed-by": 2},*

*{"1-of": [{"signed-by": 0}, {"signed-by": 1}]}*

*]*

*}*

*}*

String Endorsement policy: "Policy reference of an existing policy in your channel configuration"

/Channel/Application/Endorsement

**Materials**

1. <https://medium.com/kokster/hyperledger-fabric-endorsing-transactions-3c1b7251a709>
2. <https://medium.com/@u.avinash070990/hyperledger-fabrics-transaction-endorsement-workflow-e10302b3174>

<https://www.youtube.com/watch?v=2_RgCfjunEU&t=540s>

1. <https://fabric-sdk-node.github.io/master/Chaincode.html#getEndorsementPolicyDefinition__anchor>
2. <https://buildmedia.readthedocs.org/media/pdf/hyperledger-fabric/latest/hyperledger-fabric.pdf>