**Why Fabric**

* Participants must be identified/identifiable
* Networks need to be *permissioned*
* High transaction throughput performance
* Low latency of transaction confirmation
* Privacy and confidentiality of transactions and data pertaining to business transactions
* Do not require a native cryptocurrency

**Smart Contracts**

* Can be written in go, java , nodejs . This runs as a docker container

**Architecture for Transactions**

Execute 🡪 order 🡪 validate

This design departs radically from the order-execute paradigm in that Fabric executes transactions before reaching final agreement on their order.

**Fabric Components**

* MSP
* Ordering Service
* Endorsing Peer
* Committing Peer
* Client SDK

**What is new in Fabric 1.2**

* 1. [*Private Data*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html)
* This data is stored in a private database on the peer (sometimes called a “side” database, or “SideDB”). The ordering service is not involved here and does not see the private data
  1. *Service Discovery*

Prior to v1.2,  SDK needs a lot of information like below in order to allow applications to connect to the relevant network nodes

* CA
* TLS certificates of the
* orderers and peers on the channel
* IP addresses and port numbers
* relevant endorsement policies
* peers have the chaincode installed
* no way of knowing which peers have updated ledgers and which do not
  1. *Access control for peer functions*
* Configure which client identities can interact with peer functions, per channel.
* Lifecycle System Chain code, System Query Chain code, Configuration System Chain code & Event resources can be ACCESS CONTROLLED

This configuration can be updated in Configtx.yaml

* 1. *Pluggable endorsement and validation*

Utilize pluggable endorsement and validation logic per chaincode.

When a transaction is validated at time of commit, the peer performs various checks

* Validating the identities
* Verifying the signatures
* Ensuring the transaction satisfies the endorsement policies

However, if we require additional check apart from the above then plugin approach can be used ( Ex: Check the identity itself, check for reward points etc. )

**What is new in Fabric 1.3**

1. *Identity Mixer for anonymous transactions*

Idemix credentials can be issued from Fabric CA 1.3 ( This will allows to transaction without revealing the signature/(public key). This will be masked using Zero knowledge proof protocol). For x: Marriott will be transacting using its Public key , but peer will receive only the proof that it is Marriott but will not know the actual signature)

1. *State-based endorsement*

**Previously, ChaincodeEndorsement Policy**

* tied to the thechaincodelifecycle: chaincodeupgrade required to set a new endorsement policy
* one endorsement policy per chaincode: different endorsement policies require multiple chaincodes

**State Based endorsement is based on the KVS ( Key value store)**

|  |  |
| --- | --- |
| ***Key*** | ***Value*** |
| *K1* | *A,B* |

* Setting the endorsement policy in such a way that for the given key k1 , it has to be endorsed by A and B . The value can be changed
* By this way we can have multiple endorsement based on state

*The same can used with private collection as well*

1. *Chaincode pagination of query results*

Supports large result sets when couch DB is used

1. *Java chaincode support*

**What is new in Fabric 1.4**

1. [*Private Data*](https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data-arch.html) *Enhancements* 
   * **Reconciliation**, which allows peers for organizations that are added to private data collections to retrieve the private data for prior transactions to which they now are entitled.
   * **Client access control** to automatically enforce access control within chaincode based on the client organization collection membership without having to write specific chaincode logic. This feature is configured by using the collection configuration property memberOnlyRead:true. If you have a mixed network of v1.4 peers and prior release peers, the prior release peers will not honor this configuration until they are upgraded to v1.4.
2. *Fabric operations service*
   * Health check
   * Dynamic log levels
   * Operational metrics
   * Logging for gRPC interactions

**Ordering Service - Consensus algorithm**

* Crash Fault Tolerance – ***Current Consensus algorithm***
* Byzantine Fault Tolerance *(Planned in future releases)*
* Raft

**Few FAQs on Ordering Service**

|  |  |
| --- | --- |
| **Question:** | **I have an ordering service up and running and want to switch consensus algorithms. How do I do that?** |
| **Answer:** | This is explicitly not supported. |

|  |  |
| --- | --- |
| **Question:** | **What is the orderer system channel?** |
| **Answer:** | The orderer system channel (sometimes called ordering system channel) is the channel the orderer is initially bootstrapped with. It is used to orchestrate channel creation. The orderer system channel defines consortia and the initial configuration for new channels. At channel creation time, the organization definition in the consortium, the /Channelgroup’s values and policies, as well as the /Channel/Orderer group’s values and policies, are all combined to form the new initial channel definition. |

|  |  |
| --- | --- |
| **Question:** | **If I update my application channel, should I update my orderer system channel?** |
| **Answer:** | Once an application channel is created, it is managed independently of any other channel (including the orderer system channel). Depending on the modification, the change may or may not be desirable to port to other channels. In general, MSP changes should be synchronized across all channels, while policy changes are more likely to be specific to a particular channel. |

|  |  |
| --- | --- |
| **Question:** | **Can I have an organization act both in an ordering and application role?** |
| **Answer:** | Although this is possible, it is a highly discouraged configuration. By default the/Channel/Orderer/BlockValidation policy allows any valid certificate of the ordering organizations to sign blocks. If an organization is acting both in an ordering and application role, then this policy should be updated to restrict block signers to the subset |

**Consensus – What if one node is down ??**

One way to circumvent this is to use timeouts. If no progress is being made on deciding the next value, we wait until a timeout, then start the steps all over again. As we’re about to see, this is what consensus algorithms like Paxos and Raft essentially did.

* The transactions log, state data and backed by **Level DB** and **Couch DB**

**Ledger and World State**

There are two place which "store" data in Hyperledger Fabric:

* Ledger
* World State database

**Ledger**

The ledger is the actual "blockchain". It is a file-based ledger which stores serialized blocks. Each block has one or more transactions. Each transaction contains a read-write set which modifies one or more key/value pairs. The ledger is the definitive source of data and is immutable.

This chain is a transaction log, structured as hash-linked blocks, where each block contains a sequence of N transactions. The block header includes a hash of the block’s transactions, as well as a hash of the prior block’s header

**State DB**

The state database holds the last known committed value for any given key. It is populated when each peers validates and commits a transaction. The state database can always be rebuilt from re-processing the ledger. There are currently two options for the state database: an embedded LevelDB or an external CouchDB.

**Event Storage**

[**Hyperledger Caliper**](https://wiki.hyperledger.org/projects/caliper) **– Performance measure Project**