




ARAF KARSH HAMID

Co-Founder / CTO

MetaMagic Global Inc., NJ, USA

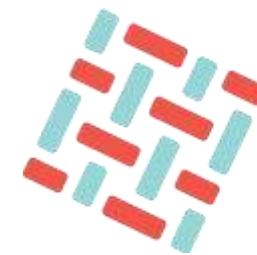
 @arafkarsh

 arafkarsh

$$\sum_{b=1}^n f(b) = \sqrt[3]{desi^3 r^2 e} \ 3D$$



BlockChain



HYPERLEDGER FABRIC

Blockchain Technology Conference

Bangkok, March 29, 2018

Hilton, Sukhumvit, Bangkok

<https://1point21gws.com/BlockchainSummit/bangkok/>

1

Why do we need Blockchain?

- Problem Statement
- How Blockchain solves this problem
- Business Requirement
- Distributed Ledger
- HyperLedger Fabric

2

HyperLedger Fabric Architecture

- Fabric Key Components
- Microservices Architecture
- 3 Components of Fabric
- Fabric Models (Assets, Chaincode..)
- Fabric Channels
- Gossip Protocol

3

Fabric Getting Started

- Setting up Fabric Network
- Fabric Example (Node.JS)
- Fabric Transaction Flow
- v1.1 Release

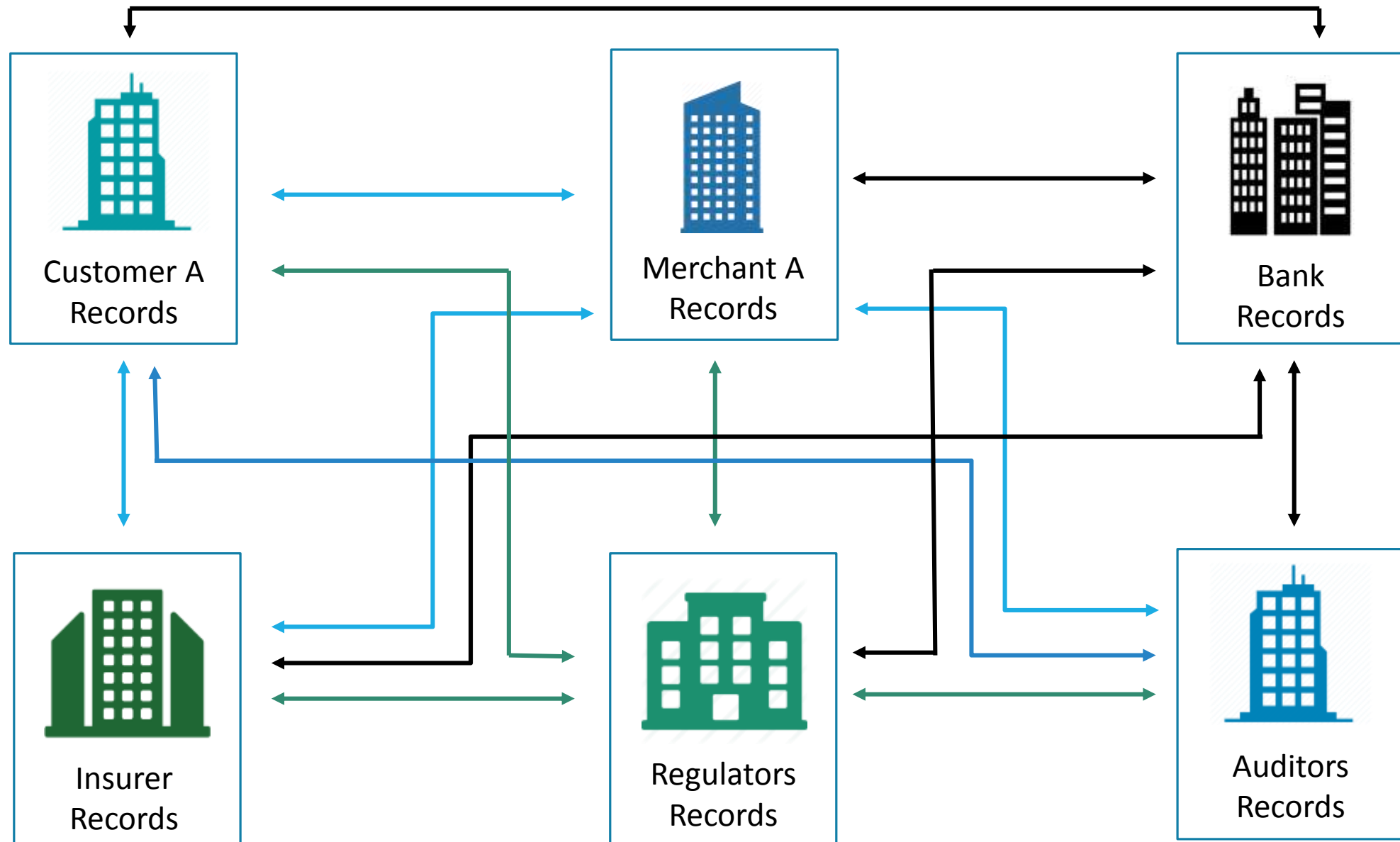
4

Conclusion

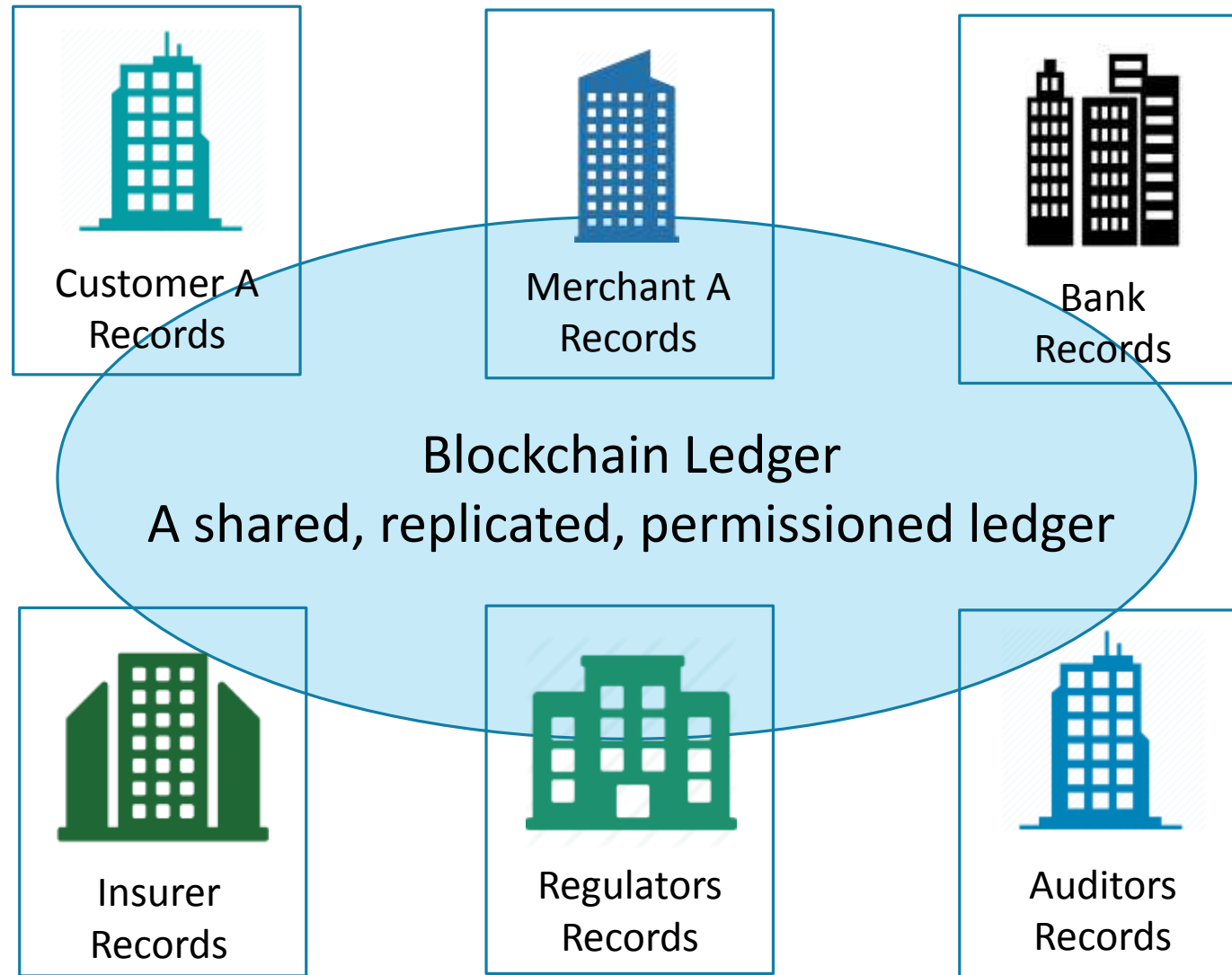
- Ethereum Vs. Fabric Vs. Corda
- Blockchain Benefits
- Industry Scenarios – Media
- References

Problem Statement

Expensive, Vulnerable & Inefficient



How Blockchain Solves this Problem...



- ✓ Consensus
- ✓ Immutability
- ✓ Provenance
- ✓ Finality

Business Requirements



Privacy

Transactions are secure, authenticated and access controls are enabled for granular access.



Trust

All the transactions are verified and endorsed by the trusted relevant parties



Shared Ledger

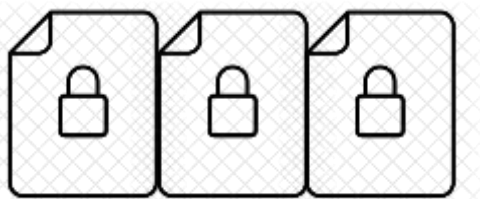
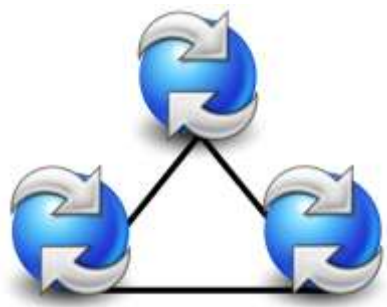
Append only immutable database shared across the business network.



Smart Contract

Business logic is embedded within the database and executed to validate and store the transactions.

Properties of a Distributed Ledger



- Decentralized, replicated across many participants, each of whom collaborate in its maintenance.
- Information recorded is append-only. Immutable.
- Immutability makes it simple to determine the **provenance of the information**.
- These properties makes it called as **“Systems of Proof”**



HyperLedger Fabric	Bitcoin
Provides Identity	Anonymity
Selective Endorsement	Proof of Work
Assets	Cryptocurrency

Bitcoin is a specific implementation of Blockchain technology

HyperLedger Fabric Key Features



Identity Management

- Provides Membership Identity Service that manages User IDs and authenticates all participants on the network.
- Access control lists can be used to Provide additional layers of permission.
- A Specific user ID could be permitted to invoke a Chaincode application but blocked from deploying a Chaincode.

- It is **private and permission based**
- Members of HyperLedger Fabric enrolls thru MSP - Membership Service Provider
- Different MSPs are supported
- Ledger data can be stored in multiple formats.
- Consensus mechanisms can be switched in and out
- **Ability to create channels** - Allowing a group of participants to create a separate ledger of transactions.



HyperLedger Fabric Key Features



Privacy & Confidentiality

- Private channels are restricted messaging paths that can be used to provide transaction privacy and confidentiality for specific subset of network members.
- All data including transaction, member and channel information, on a channel are invisible and inaccessible to any network members not explicitly access to that channel.

Efficient Processing

- Transaction execution is separated from transaction ordering and commitment.
- Division of labor unburdens ordering nodes from the demands of transaction execution and ledger maintenance,
- while peer nodes are freed from ordering (consensus) workloads.

Chaincode Functionality

- Chaincode Applications encode logic that is invoked by specific types of transactions on the channel.
- System Chaincode is distinguish as Chaincode that defines operating params for the entire channel.

Summary – Why do we need Blockchain?

Problem Statement

- Reconciliation Issues in a Multi Party environment.
- Expensive
- Vulnerable
- Inefficient

Solution

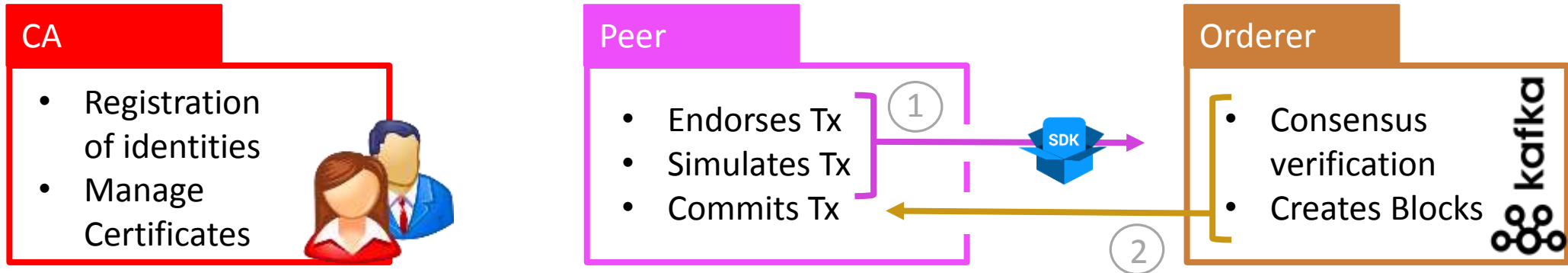
- **Consensus** among the Parties before a transaction is committed.
- **Provenance** – Origin of the asset can be easily Tracked.
- **Immutable** – Transactions are tamper proof
- **Finality** – Single Source of Truth



Fabric Architecture

HyperLedger Fabric Architecture

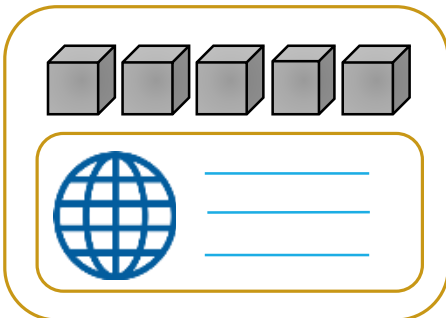
3 Components of Fabric



All these components can be clustered for scalability and to avoid Single Point of Failure

Ledger

Blockchain & World State



Channels

- Private subnet** for a set of parties based on Smart contract
- Ledger / Channel**
- Peers** can have multiple Channels

Smart Contract

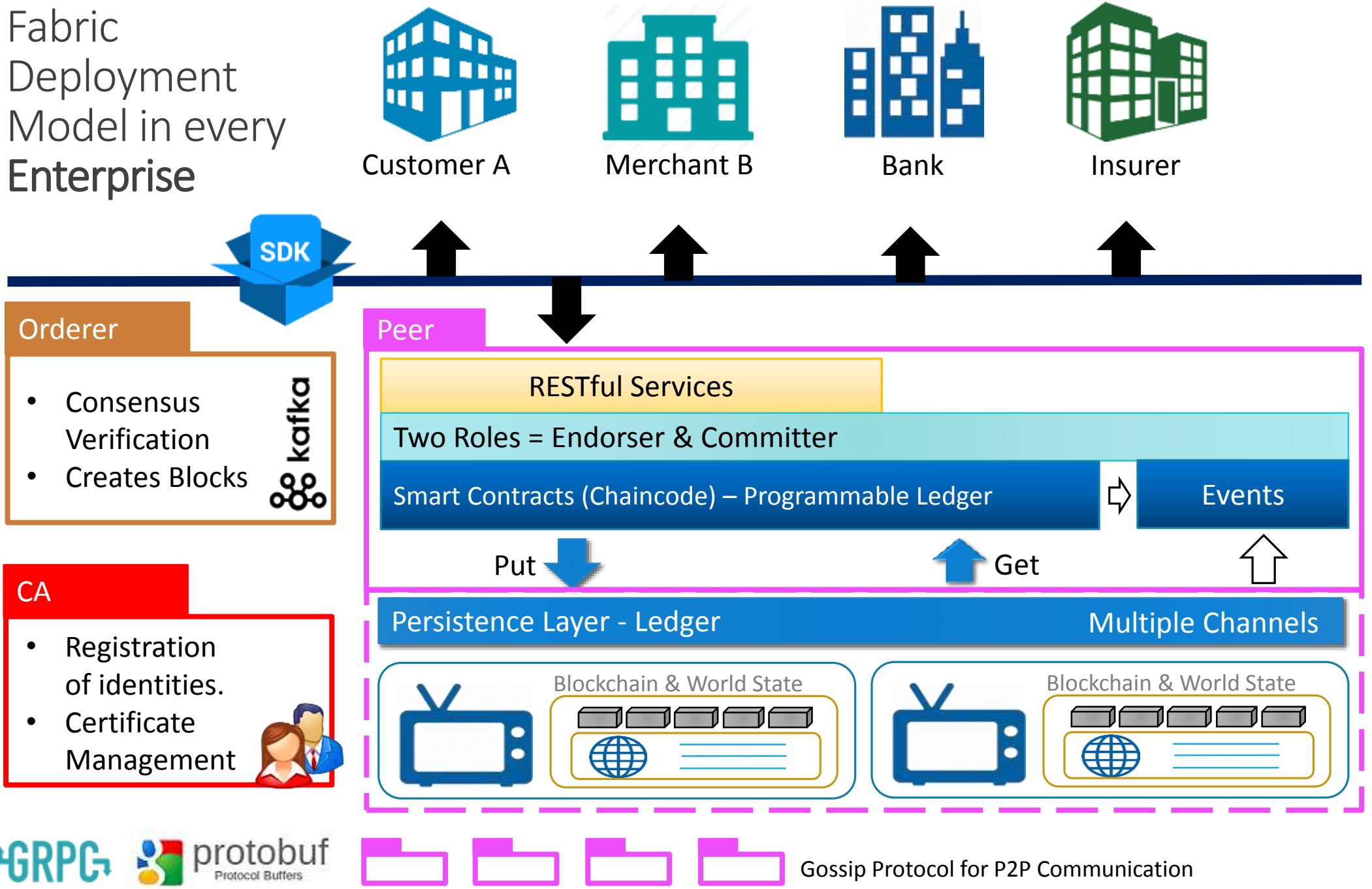
- createCar
- queryAllCars
- queryCarProperties
- changeCarColor
- changeCarOwner

Other Concepts

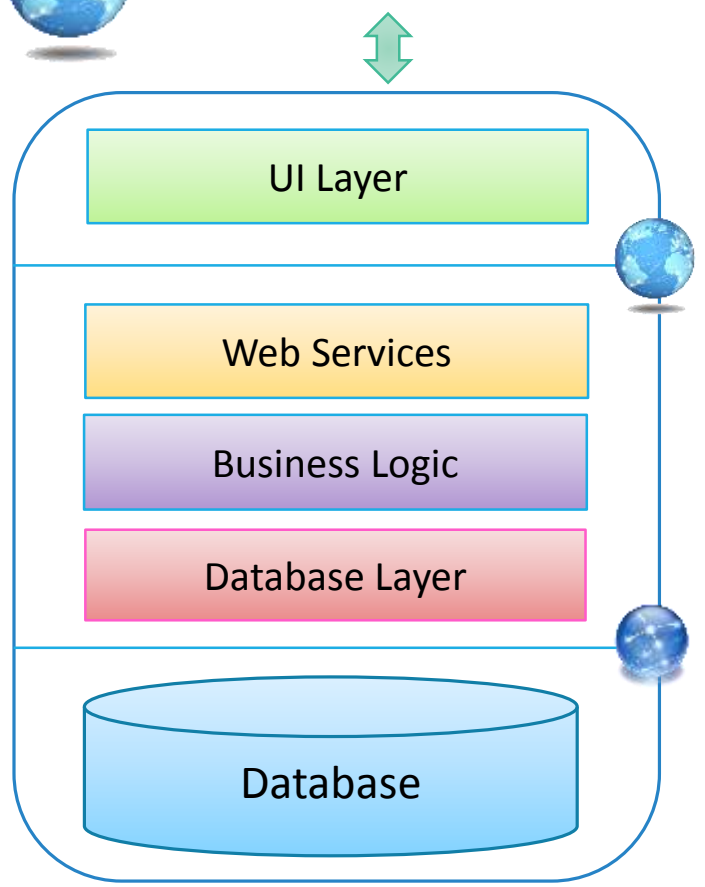
- Assets** : Anything that's valuable for the Organization
- Transactions** (State changes of Assets)
- Endorsement Policies**
- Gossip Protocol** : The glue that keeps the peers in healthy state.



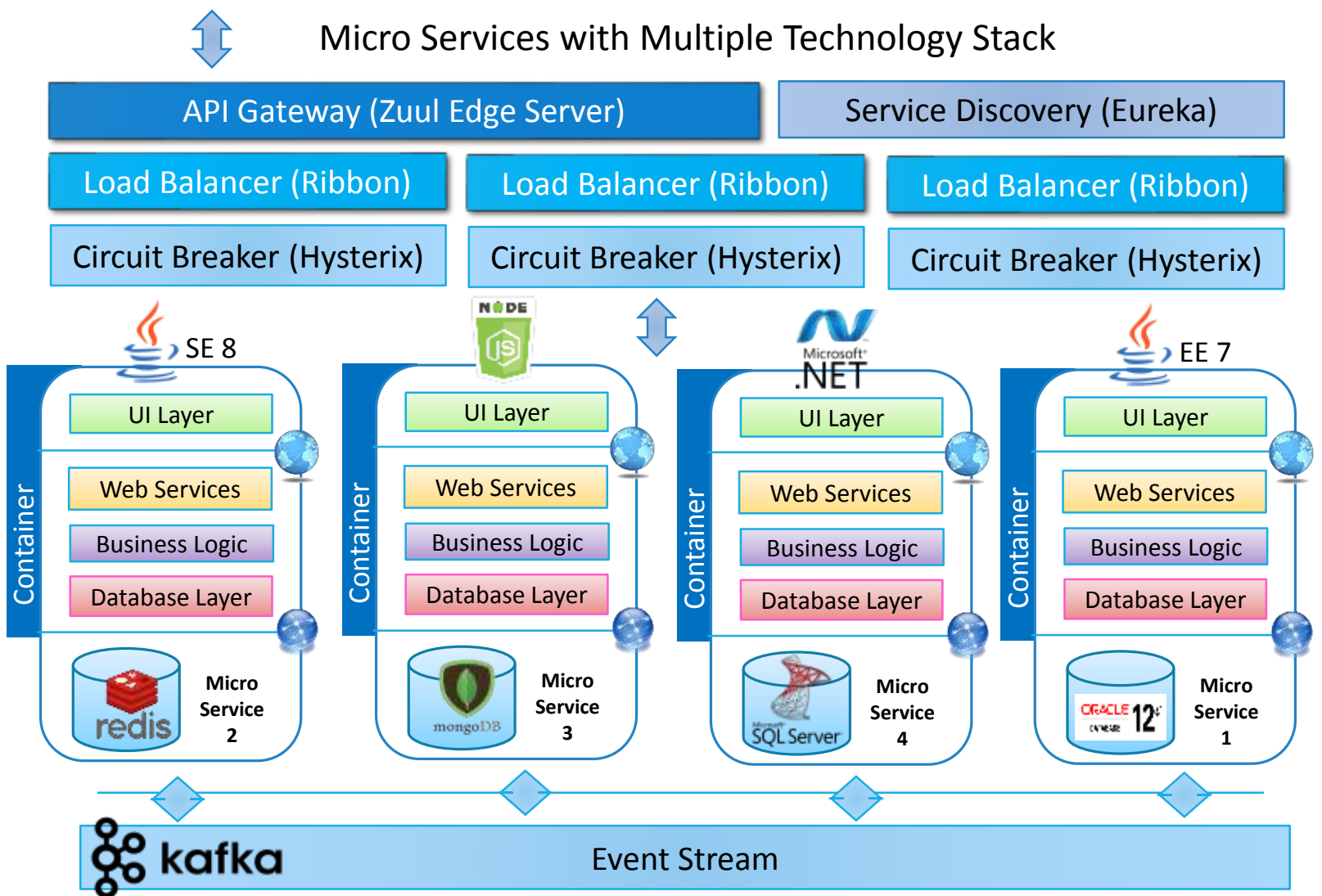
Fabric Deployment Model in every Enterprise



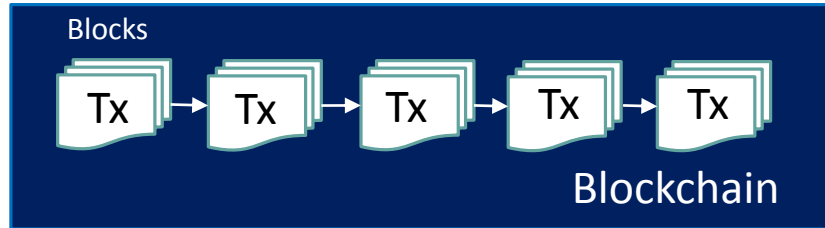
Monolithic / Micro Services Architecture



Traditional Monolithic App using Single Technology Stack



HyperLedger Fabric – Ledger



As well as a **state database to maintain current fabric state.**
This is auto derived by the Peers.



Couch DB

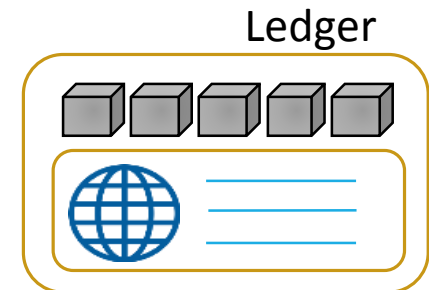
- The ledger is the **sequenced, tamper-resistant record of all state transitions** in the fabric.
- State transitions are a **result of Chaincode invocations** ('transactions') submitted by participating parties.



There is **one ledger per channel.**



Each transaction results in a set of asset key-value pairs that are committed to the ledger as creates, updates, or deletes.



Source: http://hyperledger-fabric.readthedocs.io/en/release-1.0/fabric_model.html

HyperLedger Fabric : 3 Components



Certificate Authority

- Registration of identities, or connects to LDAP as the user registry
- Issuance of Enrollment Certificates (ECerts)
- Certificate renewal and revocation
- **Every Single operation MUST be signed with a Certificate.**
- Certificates are per user.

Peer

- Can **update or Query the ledger.**
- **One Peer can be part of multiple channels.**
- Peer **Endorse** the transaction.
- All the Peers find each other and synchronizes automatically.
- Peer **manages the Ledger** which consists of the Transaction Log (Blockchain) and World State.
- **Peer has two roles Endorser and Committer.**

Orderer

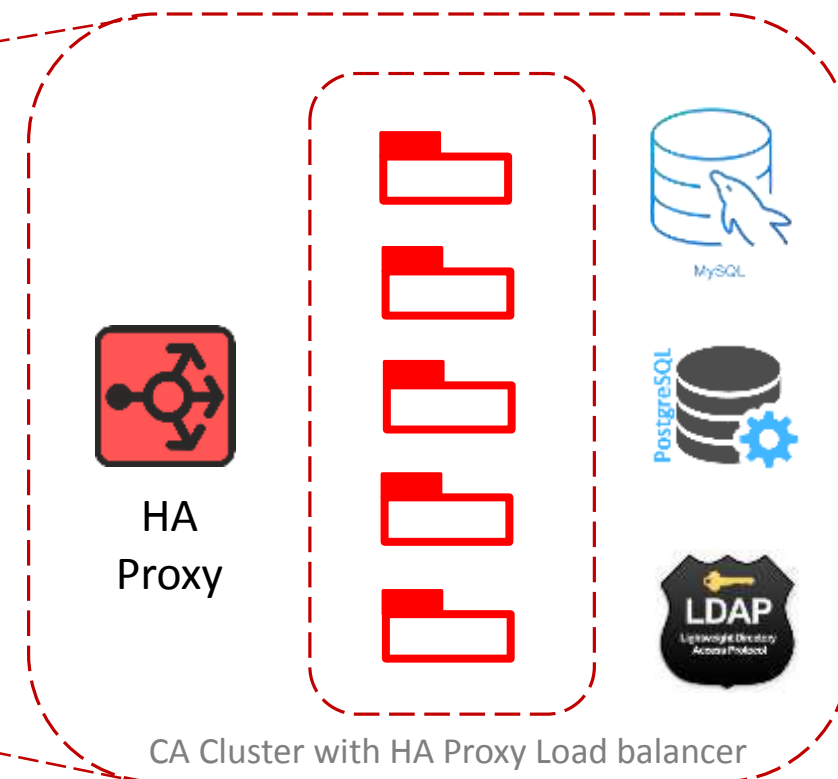
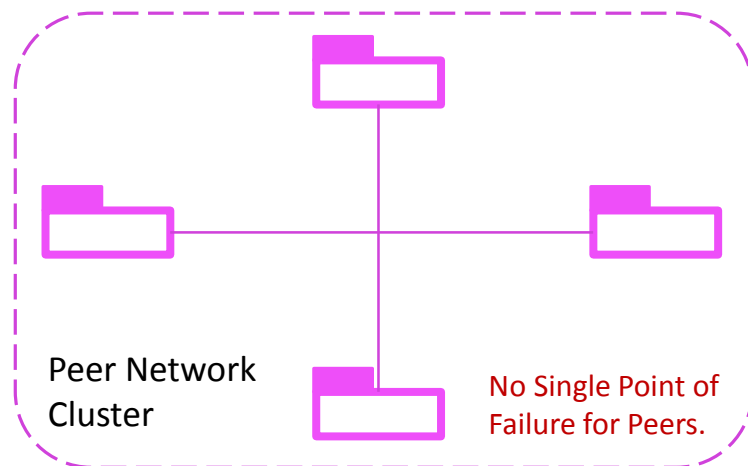
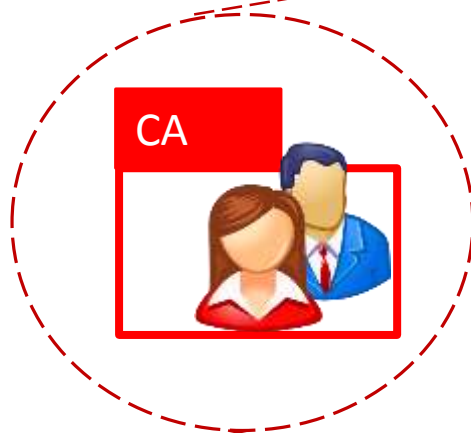
- Before anything is committed to the Ledger it must pass thru Orderer Service.
- **Provides Order of Transactions.**
- **It creates the Blocks for the Blockchain.**
- Block size can be based on no. of transactions and / or timeout value.
- *Orderer in two modes - Solo for the Dev environment and Kafka for the Production environment.*

Source: <http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html>



Fabric CA Server can be connected using

1. Fabric CA Client
2. Fabric CA SDK (REST API Calls)



Fabric CA servers in a cluster share the same database for keeping track of identities and certificates. If LDAP is configured, the identity information is kept in LDAP rather than the database.

This ensures that there is NO Single Point of Failure for CA.

Source: <http://hyperledger-fabric-ca.readthedocs.io/en/latest/users-guide.html#table-of-contents>



HyperLedger Fabric – Peer

Peer

Peers validate transactions against endorsement policies and enforce the policies.



Endorsement Policies

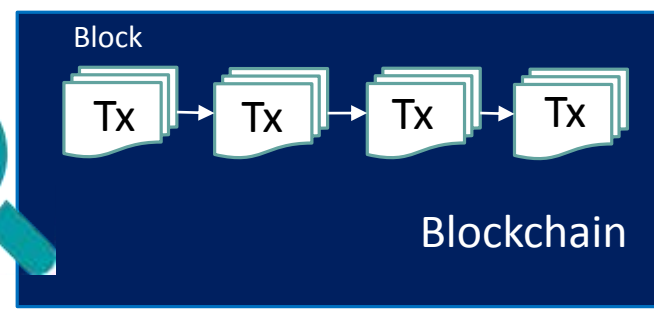
Peer



Peers can have multiple Channels

Peer

Query ledger history for a key, enabling data provenance scenarios.



```
$ docker run -it hyperledger/fabric-peer bash
# peer chaincode --help
```

Peer

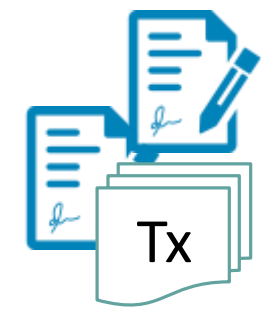
Read-only queries using a rich query language (if using CouchDB as state database)



CouchDB

Peer

Transactions contain signatures of every endorsing peer and are submitted to ordering service.



Peer



Prior to appending a block, a versioning check is performed to ensure that states for assets that were read have not changed since Chaincode execution time.

HyperLedger Fabric – Peer CLI

```
$ docker run -it hyperledger/fabric-peer bash  
# peer chaincode -help
```

Usage:

peer chaincode [command]

Available Commands:

install	Package the specified chaincode into a deployment spec and save it on the peer's path.
instantiate	Deploy the specified chaincode to the network.
invoke	Invoke the specified chaincode.
package	Package the specified chaincode into a deployment spec.
query	Query using the specified chaincode.
signpackage	Sign the specified chaincode package
upgrade	Upgrade chaincode.

Source : <http://hyperledger-fabric.readthedocs.io/en/release-1.0/chaincode4noah.html>



Flags:

	--cafile string	Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint
-C,	--chainID string	The chain on which this command should be executed (default "testchainid")
-c,	--ctor string	Constructor message for the chaincode in JSON format (default "{}")
-E,	--escc string	The name of the endorsement system chaincode to be used for this chaincode
-l,	--lang string	Language the chaincode is written in (default "golang")
-n,	--name string	Name of the chaincode
-o,	--orderer string	Ordering service endpoint
-p,	--path string	Path to chaincode
-P,	--policy string	The endorsement policy associated to this chaincode
-t,	--tid string	Name of a custom ID generation algorithm (hashing and decoding) e.g. sha256base64
	--tls Use	TLS when communicating with the orderer endpoint
-u,	--username string	Username for chaincode operations when security is enabled
-v,	--version string	Version of the chaincode specified in install/instantiate/upgrade commands
-V,	--vscc string	The name of the verification system chaincode to be used for this chaincode

Global Flags:

	--logging-level string	Default logging level and overrides, see core.yaml for full syntax
	--test.coverprofile string	Done (default "coverage.cov")

Use "peer chaincode [command] --help" **for** more information about a command.

Source : <http://hyperledger-fabric.readthedocs.io/en/release-1.0/chaincode4noah.html>

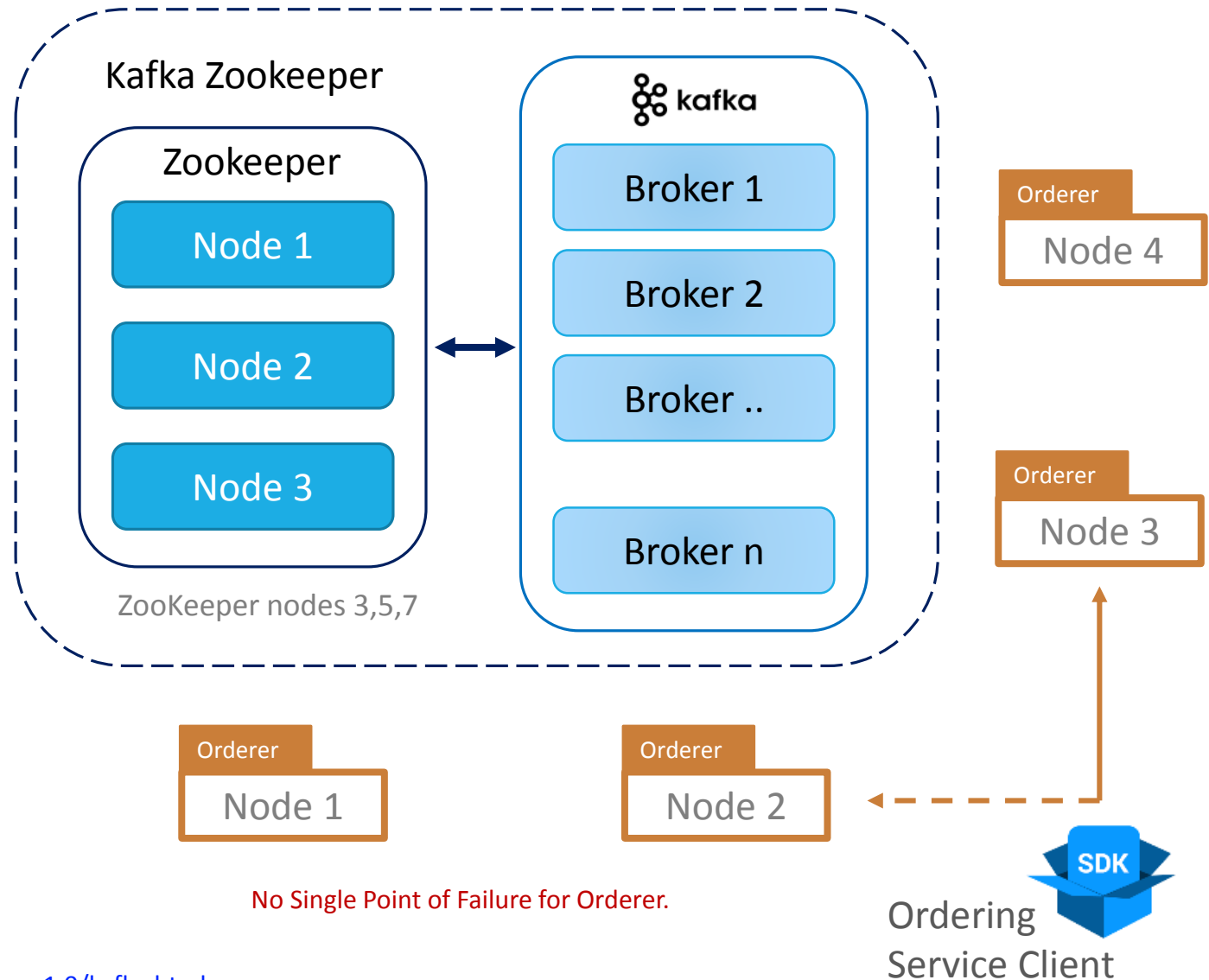
HyperLedger Fabric - Orderer

Defining Consensus is 3 Phase Process

1. Endorsement
2. Ordering
3. Validation

Service Type

1. Solo (Dev)
2. Kafka (Prod)
3. PBFT (Prod)*



HyperLedger Fabric Model – Assets / Chaincode



Assets

- The exchange of almost anything with monetary value over the network, from **whole foods to antique cars to currency futures**.
- Represented as a collection of key-value pairs, with state changes recorded as transactions on a [Channel](#) ledger.
- Assets can be represented in binary and/or JSON form.



Chaincode / Smart Contract

- It **defines an asset or assets**, and the transaction instructions for modifying the asset(s). Its the business logic.
- It enforces the **rules for reading or altering key value pairs or other state database information**.
- It execute against the ledger's current state database & are initiated through a **transaction proposal**.
- It's execution **results in a set of key value writes (write set)** that can be submitted to the network and applied to the ledger on all peers.

Source: http://hyperledger-fabric.readthedocs.io/en/release-1.0/fabric_model.html

Fabric - Endorsement Policy

- Peer uses **Endorsement policies** to decide if the **transaction** is properly **Endorsed**.
- Peer invokes **VSCC** (Validation System Chaincode) associated with transaction's Chaincode
- VSCC is tasked to do the following
 - **All Endorsements are valid** (Valid signatures from Valid Certificates)
 - There is an **appropriate number of Endorsements**.
 - **Endorsement came from expected Source(s)**

4

Roles (of the Signer) are supported

1. Member
2. Admin
3. Client
4. Peer

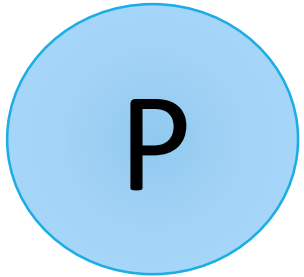
Examples

- Org1.admin
- Org2.member
- Org3.peer



Fabric – Endorsement Policy Design

Endorsement Policy has two Components



A **Principal (P)** identifies the entity whose signature is expected



A **Threshold Gate <T>** takes two inputs:

- **<t>** an integer : threshold
- **[n]** a list of : principals or gates

T(2, 'A', 'B', 'C')

Requests a signature from any 2 Principals out of A,B,C

T(1, 'A', T(1, 'B', 'C'))

Requests 1 signature from A or 1 from Principal B and C each.

This Gate essentially captures the expectation that:

- out of those **[n]** principals or gates,
- **<t>** are requested to be satisfied

Fabric - Endorsement Policy using CLI

Principals are described as **MSP.ROLE**

EXPR(E[, E...])

where **EXPR** is either **AND** or **OR**, representing the two Boolean expressions 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 1 signature from each of the three principals
- **OR ('Org1.member', 'Org2.member')** requests 1 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 1 signature from a member of the Org2 MSP and 1 signature from a member of the Org3 MSP.

The policy can be specified at instantiate time using the **-P** switch, followed by the policy. For example:

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

Source: <http://hyperledger-fabric.readthedocs.io/en/release-1.0/endorsement-policies.html>

4

Roles (of the Signer) are supported

1. Member
2. Admin
3. Client
4. Peer

Examples

- Org1.admin
- Org2.member
- Org3.peer





Why Channel is unique to Fabric?

- A Hyperledger Fabric **channel** is a private “subnet” of communication between two or more specific network members, for the purpose of conducting private and confidential transactions.
- **Channel is a completely separate instance of HyperLedger Fabric.** Every Channel is completely isolated and will never talk to each other.
- To create a new channel, the client SDK calls configuration system Chaincode and references properties such as **anchor peer**s, and members (organizations).** **This request creates a **genesis block** for the channel ledger, which stores configuration information about the channel policies, members and anchor peers.
- The consensus **service orders transactions and delivers them, in a block, to each leading peer**, which then **distributes the block to its member peers**, and across the channel, using the **gossip** protocol.

Gossip Protocol



- **Manages peer discovery and channel membership**, by continually identifying available member peers, and eventually detecting peers that have gone offline.



- Disseminates ledger data across all peers on a channel. Any peer with data that is out of sync with the **rest of the channel identifies the missing blocks and syncs itself by copying the correct data**. A state reconciliation process synchronizes **world state** across peers on each channel. **Each peer continually pulls blocks from other peers on the channel, in order to repair its own state if discrepancies are identified**

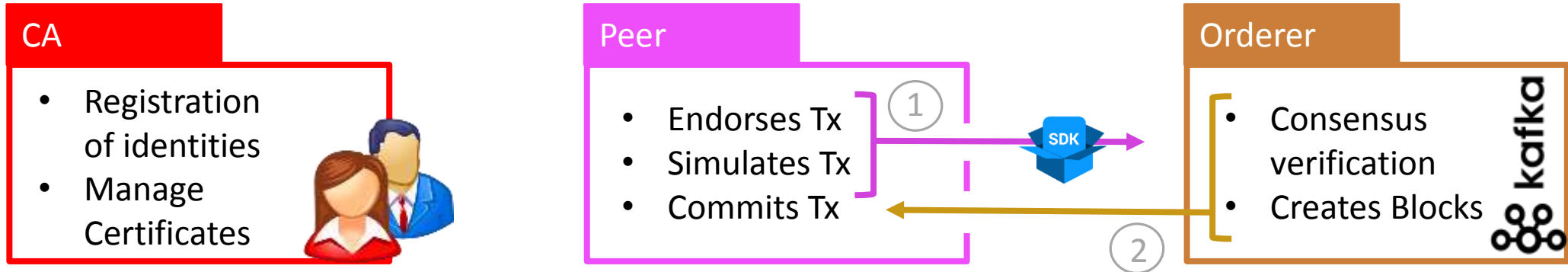


- Bring newly connected peers up to speed by allowing **peer-to-peer state transfer update of ledger data**.

Source: <http://hyperledger-fabric.readthedocs.io/en/release-1.0/gossip.html>

Summary – HyperLedger Fabric Architecture

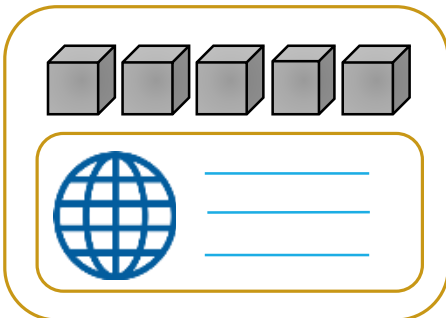
3 Components of Fabric



All these components can be clustered for scalability and to avoid Single Point of Failure

Ledger

Blockchain & World State



Channels

- Private subnet** for a set of parties based on Smart contract
- Ledger / Channel**
- Peers** can have multiple Channels

Smart Contract

- createCar
- queryAllCars
- queryCarProperties
- changeCarColor
- changeCarOwner

Other Concepts

- Endorsement Policies**
- Assets** : Anything that's valuable for the Organization
- Transactions** (State changes of Assets)
- Gossip Protocol** : The glue that keeps the peers in healthy state.



Fabric : Getting Started

Fabric Example : Pre requisites – Fabric Network

```
$ git clone -b master https://github.com/hyperledger/fabric-samples.git  
$ cd fabric-samples  
$ git checkout {TAG}
```

To ensure the samples are compatible with the version of Fabric binaries you download below, checkout the samples {TAG} that matches your Fabric version, for example, v1.1.0. To see a list of all fabric-samples tags, use command “git tag”.

Please execute the following command from within the directory into which you will extract the platform-specific binaries:

```
$ curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0
```

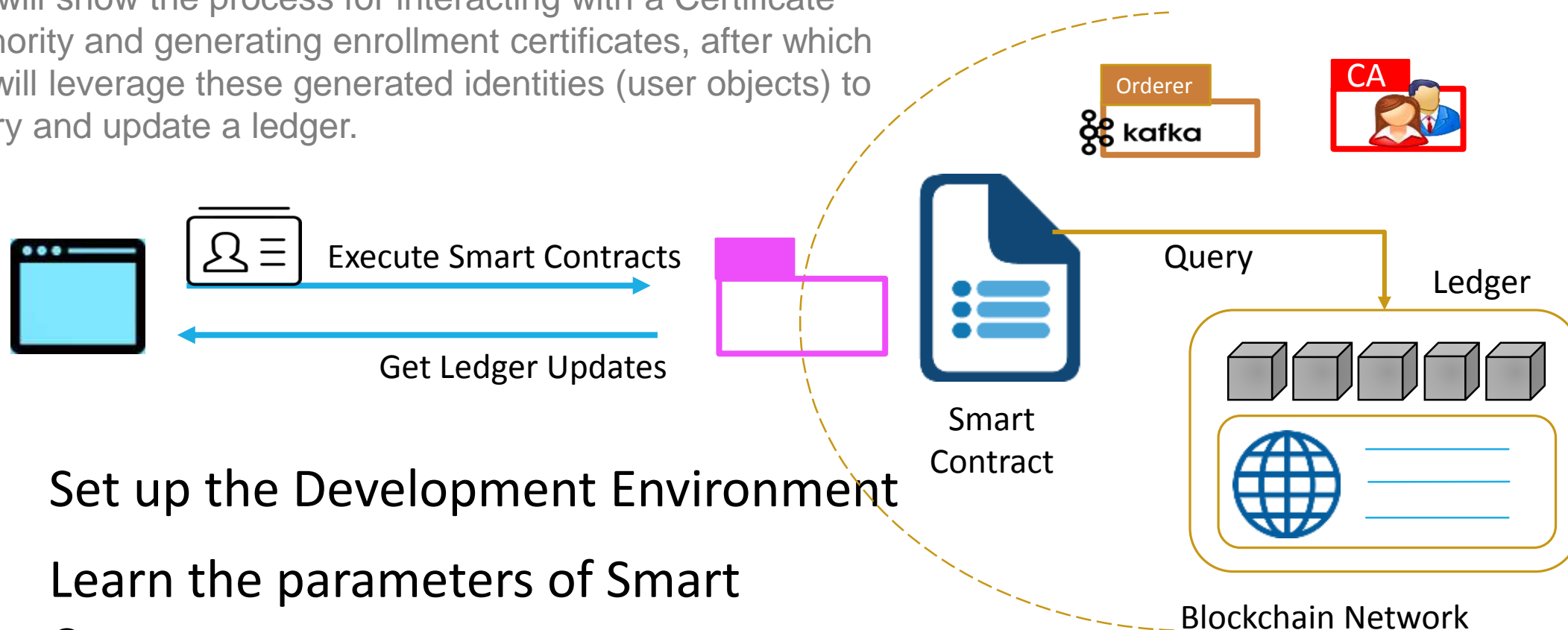
If you get an error running the above curl command, you may have too old a version of curl that does not handle redirects or an unsupported environment.

```
$ export PATH=<path to download location>/bin:$PATH
```

The script lists out the Docker images installed upon conclusion.

Fabric Example : Writing your First App

We will show the process for interacting with a Certificate Authority and generating enrollment certificates, after which we will leverage these generated identities (user objects) to query and update a ledger.



1. Set up the Development Environment
2. Learn the parameters of Smart Contract
3. Develop Apps that can Query and Update the Ledger

Fabric Example : Install the Fabcar Example

```
$ cd fabric-samples/fabcar && ls *.js
```

You will see for Node JS Programs as follows

```
enrollAdmin.js  Invoke.js  query.js  registerUser.js
```

Do a cleanup before u start installing the Sample App in the Ledger

```
$ docker rm -f $(docker ps -aq)
$ docker network prune
$ docker rmi dev-peer0.org1.example.com-fabcar-1.0-5c906e402ed29f20260ae42283216aa75549c571e2e380f3615826365d8269ba
```

```
$ npm install
```

```
$ ./startFabric.sh
```

Source: http://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html

Fabric Example : Enroll User

An admin user - admin - was registered with our Certificate Authority. Now we need to send an enroll call to the CA server and retrieve the enrollment certificate (**eCert**) for this user.

This program will invoke a certificate signing request (**CSR**) and ultimately output an eCert and key material into a newly created folder - **hfc-key-store** - at the root of this project.

```
$ node enrollAdmin.js
```

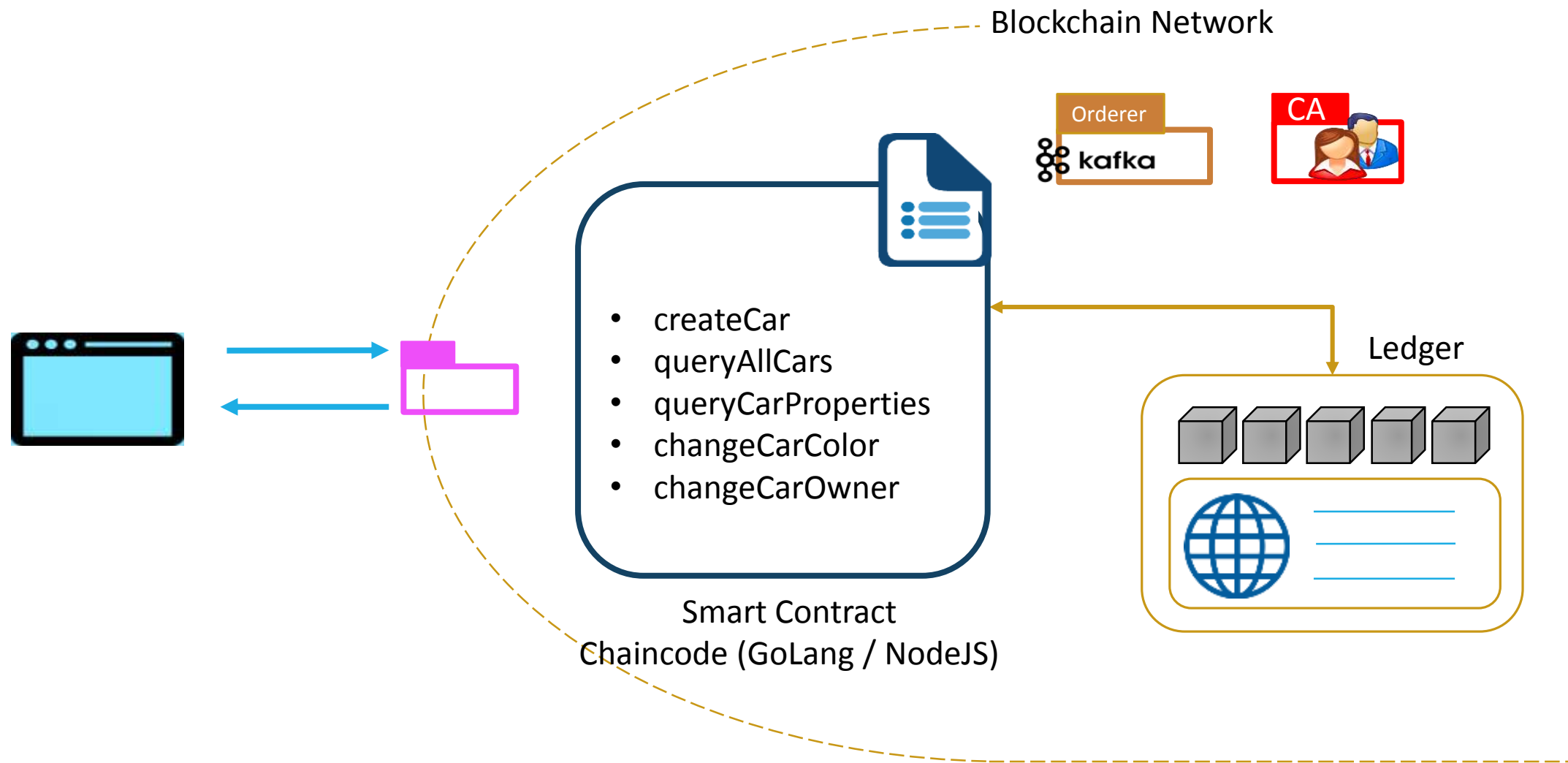
Newly generated admin **eCert**, we will now communicate with the **CA** server once more to register and enroll a new user. This user - **user1** - will be the identity we use when **querying** and **updating** the ledger.

Similar to the admin enrollment, this program invokes a **CSR** and outputs the keys and eCert into the **hfc-key-store** subdirectory. So now we have identity material for two separate users - admin & user1

```
$ node registerUser.js
```

Source: http://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html

Fabric Example : Smart Contract Functions



Chaincode Constructor



```
'use strict';
const shim = require('fabric-shim');
const util = require('util');

let Chaincode = class {

  // The Init method is called when the Smart Contract 'fabcar' is instantiated by the blockchain network
  // Best practice is to have any Ledger initialization in separate function — see initLedger()
  async Init(stub) {
    console.info('===== Instantiated fabcar chaincode =====');
    return shim.success();
  }

  // The Invoke method is called as a result of an application request to run the Smart Contract
  // 'fabcar'. The calling application program has also specified the particular smart contract
  // function to be called, with arguments
  async Invoke(stub) {
    let ret = stub.getFunctionAndParameters();
    console.info(ret);

    let method = this[ret.fcn];
    if (!method) {
      console.error('no function of name: ' + ret.fcn + ' found');
      throw new Error('Received unknown function ' + ret.fcn + ' invocation');
    }
    try {
      let payload = await method(stub, ret.params);
      return shim.success(payload);
    } catch (err) {
      console.log(err);
      return shim.error(err);
    }
  }

  async createCar(stub, args) {
    console.info('===== START : Create Car =====');
    if (args.length !== 5) {
      throw new Error('Incorrect number of arguments. Expecting 5');
    }
    var car = {
      docType: 'car',
      make: args[1],
      model: args[2],
      color: args[3],
      owner: args[4]
    };

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : Create Car =====');
  }

  async changeCarOwner(stub, args) {
    console.info('===== START : changeCarOwner =====');
    if (args.length !== 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let carAsBytes = await stub.getState(args[0]);
    let car = JSON.parse(carAsBytes);
    car.owner = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : changeCarOwner =====');
  }
}
```

Invoke(stub) method is called as result
of an application request to the
Chaincode



```
'use strict';
const shim = require('fabric-shim');
const util = require('util');

let Chaincode = class {

  // The Init method is called when the Smart Contract 'fabcar' is instantiated by the blockchain network
  // Best practice is to have any Ledger initialization in separate function — see initLedger()
  async Init(stub) {
    console.info('===== Instantiated fabcar chaincode =====');
    return shim.success();
  }

  // The Invoke method is called as a result of an application request to run the Smart Contract
  // 'fabcar'. The calling application program has also specified the particular smart contract
  // function to be called, with arguments
  async Invoke(stub) {
    let ret = stub.getFunctionAndParameters();
    console.info(ret);

    let method = this[ret.fcn];
    if (!method) {
      console.error('no function of name: ' + ret.fcn + ' found');
      throw new Error('Received unknown function ' + ret.fcn + ' invocation');
    }
    try {
      let payload = await method(stub, ret.params);
      return shim.success(payload);
    } catch (err) {
      console.log(err);
      return shim.error(err);
    }
  }

  async createCar(stub, args) {
    console.info('===== START : Create Car =====');
    if (args.length !== 5) {
      throw new Error('Incorrect number of arguments. Expecting 5');
    }
    var car = {
      docType: 'car',
      make: args[1],
      model: args[2],
      color: args[3],
      owner: args[4]
    };

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : Create Car =====');
  }

  async changeCarOwner(stub, args) {
    console.info('===== START : changeCarOwner =====');
    if (args.length !== 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let carAsBytes = await stub.getState(args[0]);
    let car = JSON.parse(carAsBytes);
    car.owner = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : changeCarOwner =====');
  }
}
```

Ledger Persistence Calls – Commands

createCar(stub, args)



```
'use strict';
const shim = require('fabric-shim');
const util = require('util');

let Chaincode = class {

  // The Init method is called when the Smart Contract 'fabcar' is instantiated by the blockchain network
  // Best practice is to have any Ledger initialization in separate function — see initLedger()
  async Init(stub) {
    console.info('===== Instantiated fabcar chaincode =====');
    return shim.success();
  }

  // The Invoke method is called as a result of an application request to run the Smart Contract
  // 'fabcar'. The calling application program has also specified the particular smart contract
  // function to be called, with arguments
  async Invoke(stub) {
    let ret = stub.getFunctionAndParameters();
    console.info(ret);

    let method = this[ret.fcn];
    if (!method) {
      console.error('no function of name: ' + ret.fcn + ' found');
      throw new Error('Received unknown function ' + ret.fcn + ' invocation');
    }
    try {
      let payload = await method(stub, ret.params);
      return shim.success(payload);
    } catch (err) {
      console.log(err);
      return shim.error(err);
    }
  }

  async createCar(stub, args) {
    console.info('===== START : Create Car =====');
    if (args.length !== 5) {
      throw new Error('Incorrect number of arguments. Expecting 5');
    }
    var car = {
      docType: 'car',
      make: args[1],
      model: args[2],
      color: args[3],
      owner: args[4]
    };

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : Create Car =====');
  }

  async changeCarOwner(stub, args) {
    console.info('===== START : changeCarOwner =====');
    if (args.length !== 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let carAsBytes = await stub.getState(args[0]);
    let car = JSON.parse(carAsBytes);
    car.owner = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : changeCarOwner =====');
  }
}
```

changeCarOwner(stub, args)



```
'use strict';
const shim = require('fabric-shim');
const util = require('util');

let Chaincode = class {

  // The Init method is called when the Smart Contract 'fabcar' is instantiated by the blockchain network
  // Best practice is to have any Ledger initialization in separate function — see initLedger()
  async Init(stub) {
    console.info('===== Instantiated fabcar chaincode =====');
    return shim.success();
  }

  // The Invoke method is called as a result of an application request to run the Smart Contract
  // 'fabcar'. The calling application program has also specified the particular smart contract
  // function to be called, with arguments
  async Invoke(stub) {
    let ret = stub.getFunctionAndParameters();
    console.info(ret);

    let method = this[ret.fcn];
    if (!method) {
      console.error('no function of name: ' + ret.fcn + ' found');
      throw new Error('Received unknown function ' + ret.fcn + ' invocation');
    }
    try {
      let payload = await method(stub, ret.params);
      return shim.success(payload);
    } catch (err) {
      console.log(err);
      return shim.error(err);
    }
  }

  async createCar(stub, args) {
    console.info('===== START : Create Car =====');
    if (args.length !== 5) {
      throw new Error('Incorrect number of arguments. Expecting 5');
    }
    var car = {
      docType: 'car',
      make: args[1],
      model: args[2],
      color: args[3],
      owner: args[4]
    };

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : Create Car =====');
  }

  async changeCarOwner(stub, args) {
    console.info('===== START : changeCarOwner =====');
    if (args.length !== 2) {
      throw new Error('Incorrect number of arguments. Expecting 2');
    }

    let carAsBytes = await stub.getState(args[0]);
    let car = JSON.parse(carAsBytes);
    car.owner = args[1];

    await stub.putState(args[0], Buffer.from(JSON.stringify(car)));
    console.info('===== END : changeCarOwner =====');
  }
}
```

Ledger Persistence Calls – Queries

queryCar(stub, args) →

```
async queryCar(stub, args) {
  if (args.length !== 1) {
    throw new Error('Incorrect number of arguments. Expecting CarNumber ex: CAR01');
  }
  let carNumber = args[0];

  let carAsBytes = await stub.getState(carNumber); //get the car from chaincode state
  if (!carAsBytes || carAsBytes.toString().length <= 0) {
    throw new Error(carNumber + ' does not exist: ');
  }
  console.log(carAsBytes.toString());
  return carAsBytes;
}
```

queryAllCars(stub, args) →

```
async queryAllCars(stub, args) {
  let startKey = 'CAR0';
  let endKey = 'CAR999';

  let iterator = await stub.getStateByRange(startKey, endKey);

  let allResults = [];
  while (true) {
    let res = await iterator.next();

    if (res.value && res.value.value.toString()) {
      let jsonRes = {};
      console.log(res.value.value.toString('utf8'));

      jsonRes.Key = res.value.key;
      try {
        jsonRes.Record = JSON.parse(res.value.value.toString('utf8'));
      } catch (err) {
        console.log(err);
        jsonRes.Record = res.value.value.toString('utf8');
      }
      allResults.push(jsonRes);
    }
    if (res.done) {
      console.log('end of data');
      await iterator.close();
      console.info(allResults);
      return Buffer.from(JSON.stringify(allResults));
    }
  }
}
```

```
};
```

```
shim.start(new Chaincode());
```

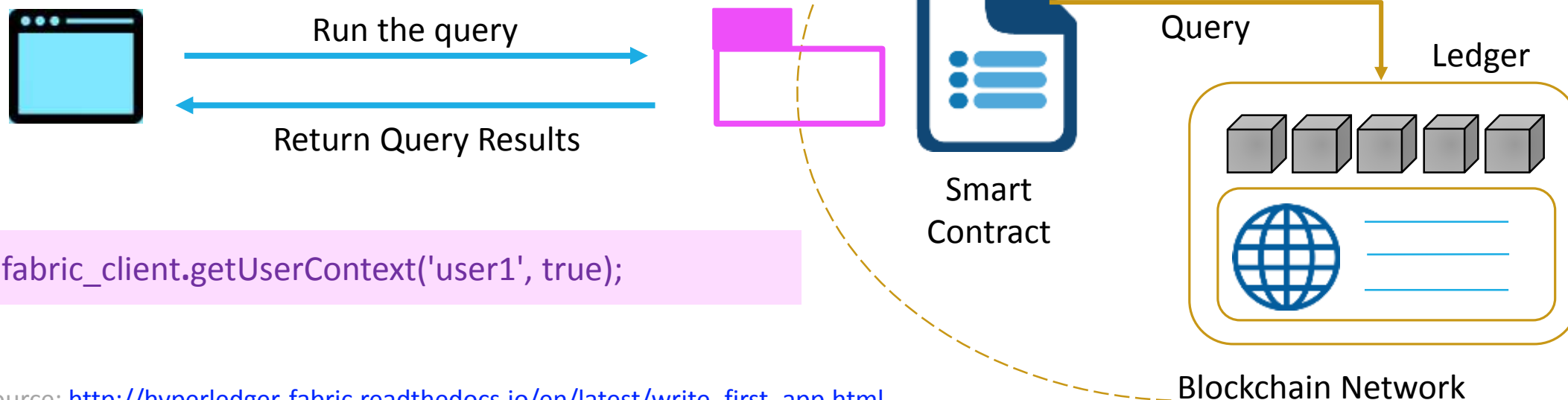


Fabric Example : Query The Ledger

Similar to the admin enrollment, this program invokes a CSR and outputs the keys and eCert into the **hfc-key-store** subdirectory. So now we have identity material for two separate users - admin & user1

```
$ node query.js
```

Queries are how you read data from the ledger. This data is stored as a series of key-value pairs, and you can query for the value of a single key, multiple keys, or – if the ledger is written in a rich data storage format like JSON – perform complex searches against it (looking for all assets that contain certain keywords, for example)



Source: http://hyperledger-fabric.readthedocs.io/en/latest/write_first_app.html

HyperLedger Fabric – Node.JS Client API

1. Initialize the Client API
2. Create a Channel
3. Create a Peer
4. Add Peer to the Channel

GetHistoryForKey()

Query ChainCode: **fabcar**

Query Function: **queryAllCars**

```
$ node query.js
```

GetHistoryForKey()

```
//  
var fabric_client = new Fabric_Client();  
  
// setup the fabric network  
var channel = fabric_client.newChannel('mychannel');  
var peer = fabric_client.newPeer('grpc://localhost:7051');  
channel.addPeer(peer);  
  
//  
var member_user = null;  
var store_path = path.join(__dirname, 'hfc-key-store');  
console.log('Store path: '+store_path);  
var tx_id = null;
```

```
// queryCar chaincode function - requires 1 argument, ex: args: ['CAR4'],  
// queryAllCars chaincode function - requires no arguments , ex: args: [''],  
const request = {  
  //targets : --- letting this default to the peers assigned to the channel  
  chaincodeId: 'fabcar',  
  fcn: 'queryAllCars',  
  args: ['']  
};  
  
// send the query proposal to the peer  
return channel.queryByChaincode(request);
```

HyperLedger Fabric – Node.JS Client API

```
//  
var fabric_client = new Fabric_Client();  
  
// setup the fabric network  
var channel = fabric_client.newChannel('mychannel');  
var peer = fabric_client.newPeer('grpc://localhost:7051');  
channel.addPeer(peer);  
var order = fabric_client.newOrderer('grpc://localhost:7050');  
channel.addOrderer(order);
```

\$ node invoke.js

1. Get a Transaction ID based on the user assigned to the client.
2. Send Transaction Proposal for
3. createCar() function in the Chaincode creates a new transaction
4. changeCarOwner() function in the Chaincode updates the owner.

```
// get a transaction id object based on the current user assigned to fabric client  
tx_id = fabric_client.newTransactionID();  
console.log("Assigning transaction_id: ", tx_id.transaction_id);  
  
// createCar chaincode function - requires 5 args, ex: args: ['CAR12', 'Honda', 'Accord',  
// changeCarOwner chaincode function - requires 2 args , ex: args: ['CAR10', 'Dave'],  
// must send the proposal to endorsing peers  
var request = {  
  //targets: let default to the peer assigned to the client  
  chaincodeId: 'fabcar',  
  fcn: 'createCar',  
  args: ['CAR10', 'Chevy', 'Volt', 'Red', 'Nick'],  
  chainId: 'mychannel',  
  txId: tx_id  
};  
// send the transaction proposal to the peers  
return channel.sendTransactionProposal(request);
```

```
// get a transaction id object based on the current user assigned to fabric client  
tx_id = fabric_client.newTransactionID();  
console.log("Assigning transaction_id: ", tx_id.transaction_id);  
  
var request = {  
  //targets: let default to the peer assigned to the client  
  chaincodeId: 'fabcar',  
  fcn: 'changeCarOwner',  
  args: ['CAR10', 'Dave'],  
  chainId: 'mychannel',  
  txId: tx_id  
};  
  
// send the transaction proposal to the peers  
return channel.sendTransactionProposal(request);
```



Fabric : Transaction Flow



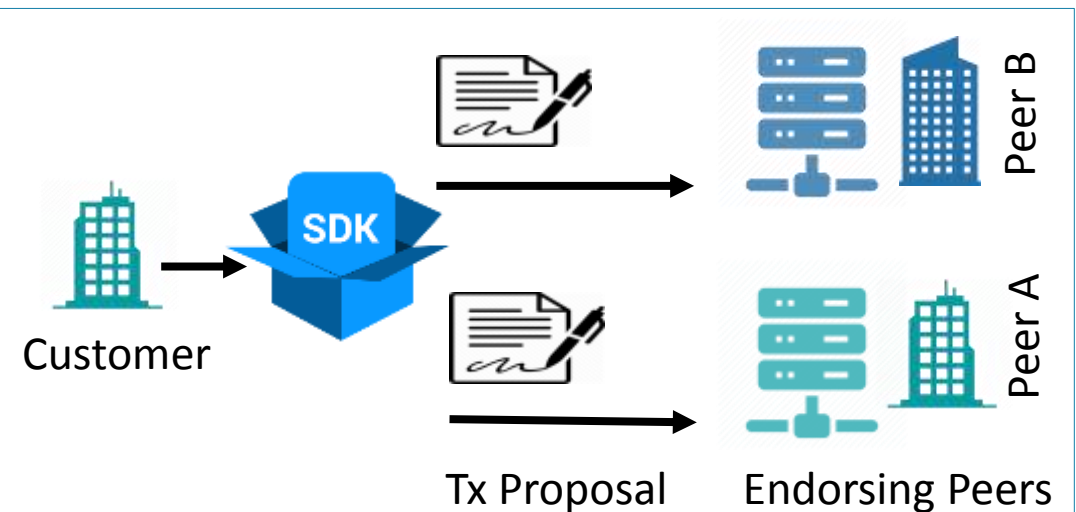
Scenario

Customer wants to order Red Roses from Merchant for a new price.



1

- Endorsement Policy in the Chaincode states proposal must be accepted by all parties.
- Proposal includes the function of the Chaincode that needs to be triggered and other params.
- The SDK packages the transaction proposal into the properly architected format (**protocol buffer over gRPC**) and takes the user's cryptographic credentials to produce a unique signature for this transaction proposal.



2

2.1 The Endorsing Peers : Verification

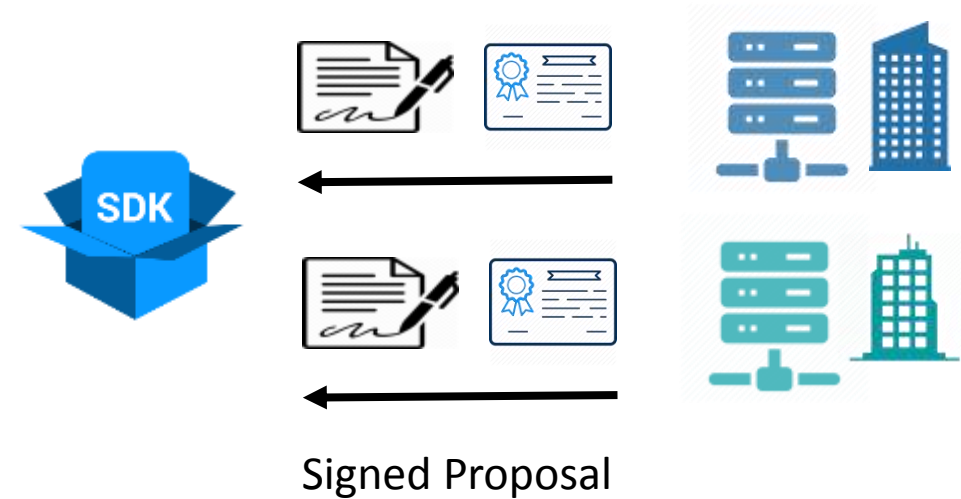
1. The transaction Proposal is well formed
2. It has not been submitted already in the past (replay attack protection)
3. The signature is valid using (MSP) and the
4. Submitter (Customer) is properly authorized to perform the proposed operation on the Channel.



2

2.2 The Endorsing Peers : Execute

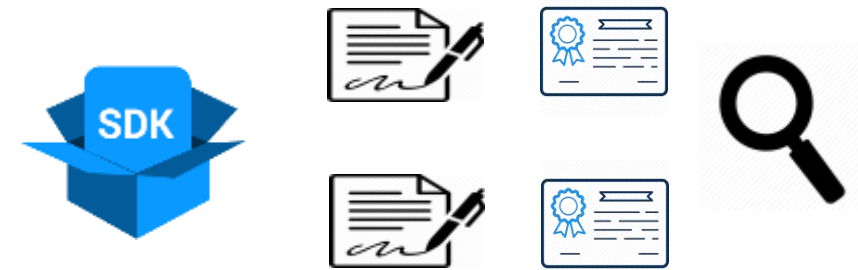
1. The endorsing peers take the transaction proposal inputs as arguments to the invoked chaincode's function. It has not been submitted already in the past (replay attack protection)
2. The chaincode is then executed against the current state database to produce transaction results including a **response value, read set, and write set**.
 - No updates are made to the ledger at this point.



3

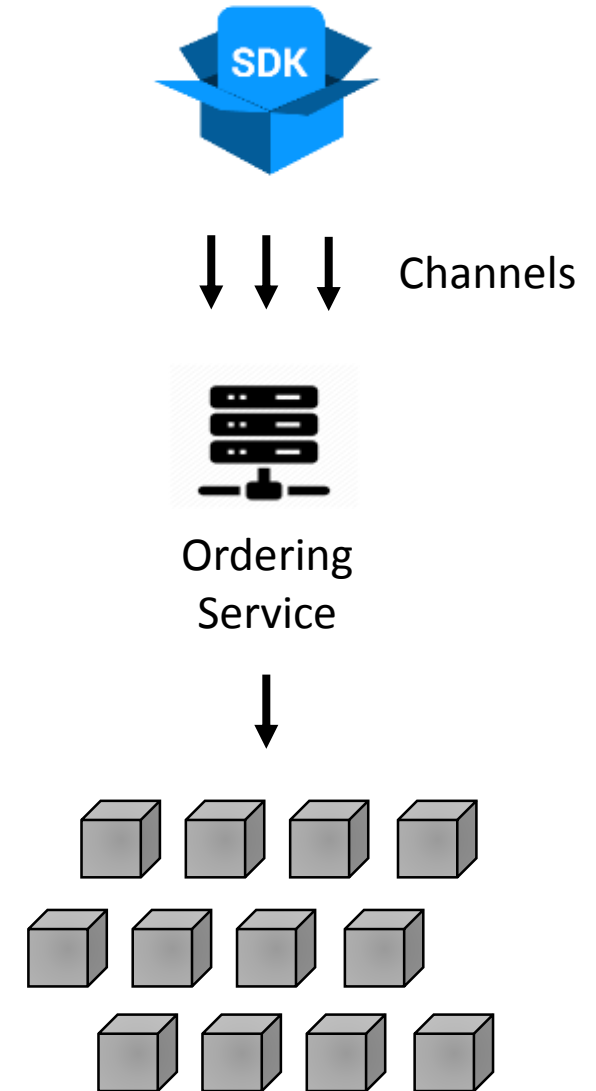
Proposal Responses are inspected

1. App verifies the endorsing peer signatures.
2. Checks if the Endorsement policy is fulfilled (ie. Customer and the Merchant Peer both endorse the proposal).



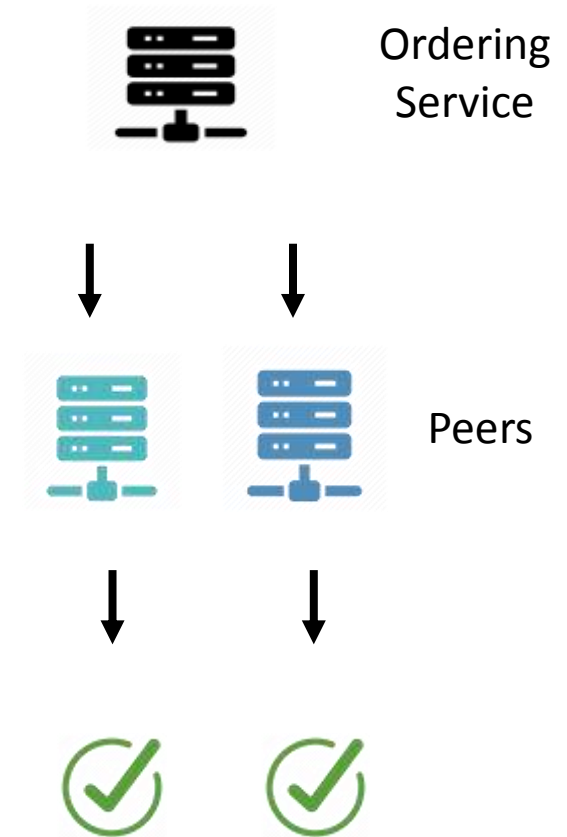
Convert Endorsement to transaction

1. The App “broadcasts” the transaction proposal and response within a “transaction message” to the Ordering Service
2. The transaction will contain the read/write sets. The endorsement signatures and Channel ID.
3. The ordering service doesn't need to inspect the content of the transaction.
4. It receives transactions from all the channels from the network and orders them chronologically by the Channel and
5. Creates blocks of transactions per channel.



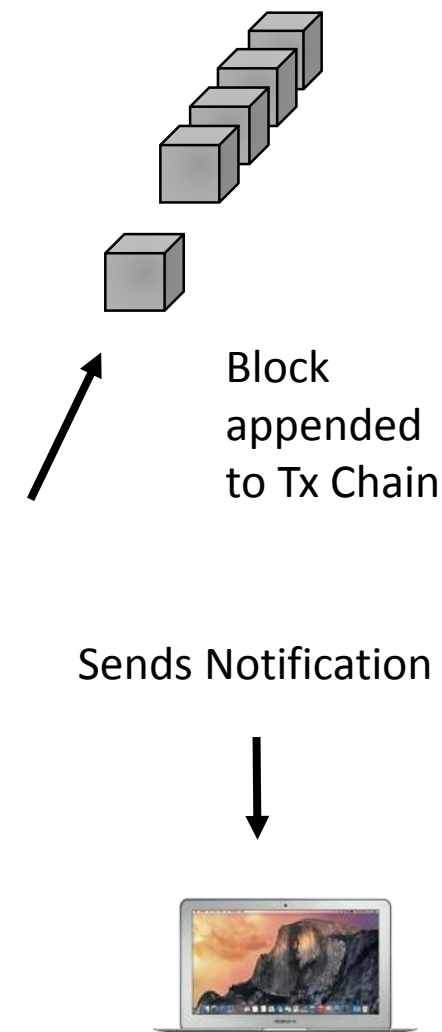
Transaction Validated and Committed

1. The blocks of transactions are delivered to all Peers in the network based on the Channel ID.
2. Validate the transaction based on Endorsement policy.
3. Ensure that no change in the Ledger state based on the Read-set variables since the read-set was generated by the transaction execution.
4. Transactions in the block are tagged as being Valid or Invalid.

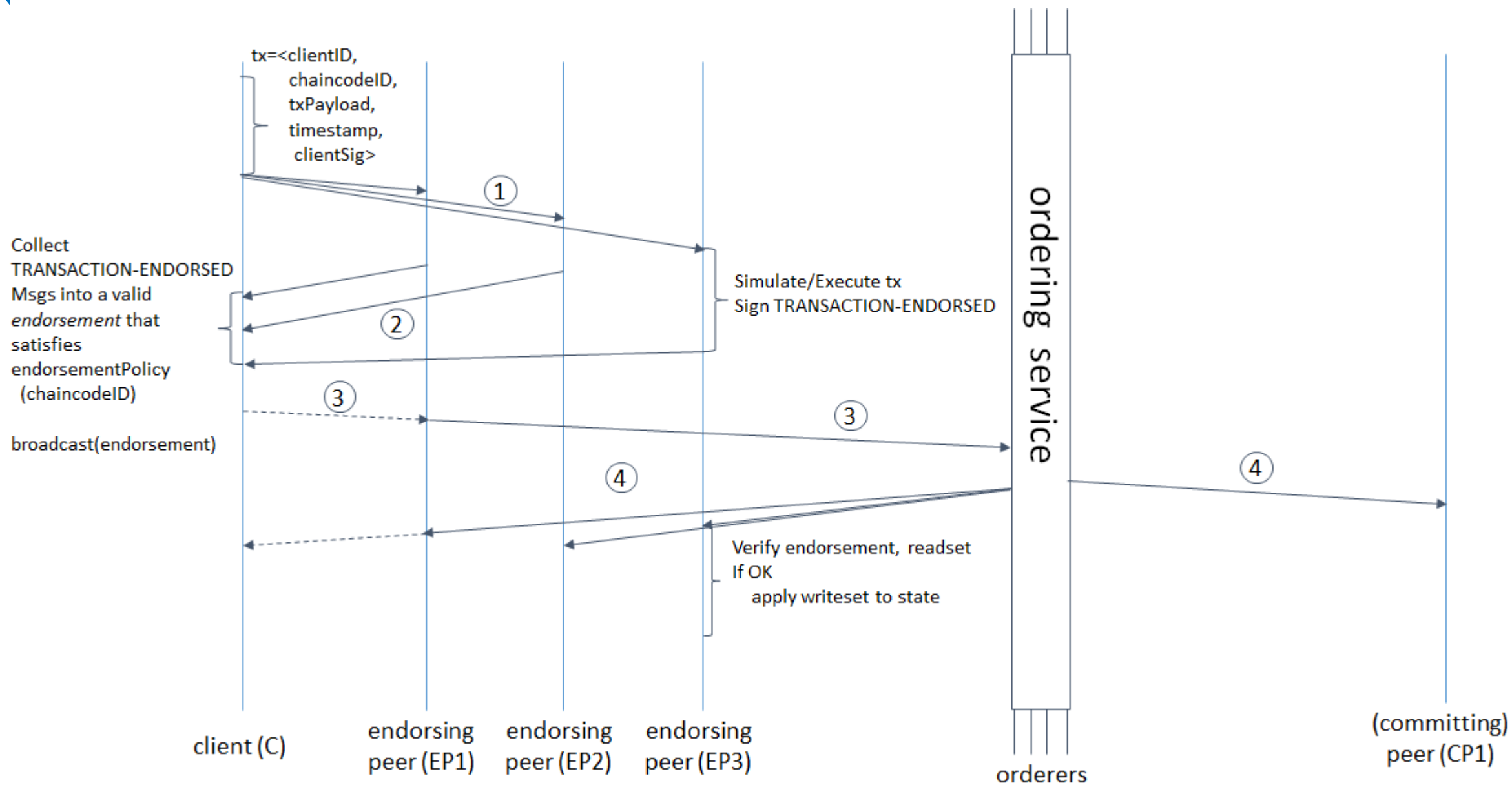


Ledger Updated

1. Each Peer appends the Block to the Channels chain.
2. For Each VALID transactions the write-set is committed to the World State Database.
3. An Event is emitted to Notify the Client Application.
4. Transactions in the block are tagged as being Valid or Invalid.



Transaction Proposal to Commit



HyperLedger Fabric v1.1 release – Mar 20, 2018

- **Node.js chaincode support**- developers can now author chaincode using the most popular framework for the world's most popular programming language
- Channel based event service – to enable clients to subscribe to block and block transaction events on a per-channel basis.
- Ability to package CouchDB indexes with chaincode, to improve performance
- Ability to generate certificate revocation lists (CRLs)
- Ability to dynamically update client identities and affiliations
- Node.js SDK connection profiles to simplify connections to Fabric nodes
- Mutual Transport Layer Security (TLS) between Fabric nodes, and between clients and nodes
- **Ability to encrypt ledger data for confidentiality using the chaincode encryption library**
- **Attribute-based Access Control in chaincode**
- Chaincode APIs to retrieve client identity for access control decisions
- Performance improvements for transaction throughput and response time

Source: <https://www.hyperledger.org/blog/2018/03/20/hyperledger-fabric-v1-1-released>



Conclusion

Ethereum Vs. HyperLedger Fabric Vs. R3 Corda



Characteristics	Ethereum	HyperLedger Fabric	R3 Corda
Description of the Platform	Generic Blockchain Platform	Modular Blockchain Platform	Specialized distributed ledger platform for Financial Industry
Release History	July 2015	v0.6 Sept 2016 , v1.0 July 2017, v1.1 Mar 2018	v-m0.0 May 2016 , v1.0 Oct 2017, v3.0 Mar 2018
Crypto Currency	Ether / Tokens (Usage, Work) via Smart Contract	None Currency and Tokens via Chaincode	None
Governance	Ethereum Developers Enterprise Ethereum Alliance	Linux Foundation IBM	R3
Consensus	Mining based on Proof of Work (POW) – All participants need to agree. Ledger Level	Selective Endorsement. Consensus can be even within a channel with select parties instead of everyone. Transaction Level	Specific understanding of Consensus. (Validity, Uniqueness) Transaction Level
Network	Permissionless, Public or Private	Permissioned, Private	Permissioned, Private
State	Account Data	Key-value Database Transaction Log, World State	Vault contains States Historic & Current State
Smart Contracts	Solidity	Chaincode (GoLang, Node.JS)	Smart Contract (Kotlin, Java)
Development Languages	GoLang, C++, Python	Java, Node.JS, Python (Post 1.0)	Java, Kotlin

Summary – Benefits of Blockchain Technologies



Saves Time

Immutable transaction across parties done at the same time.



Increases Trust

Through shared process and unified Systems of Record. For end consumers it's a System of Proof.



Reduces Risk

Tampering of data, fraud and cyber crime is avoided.

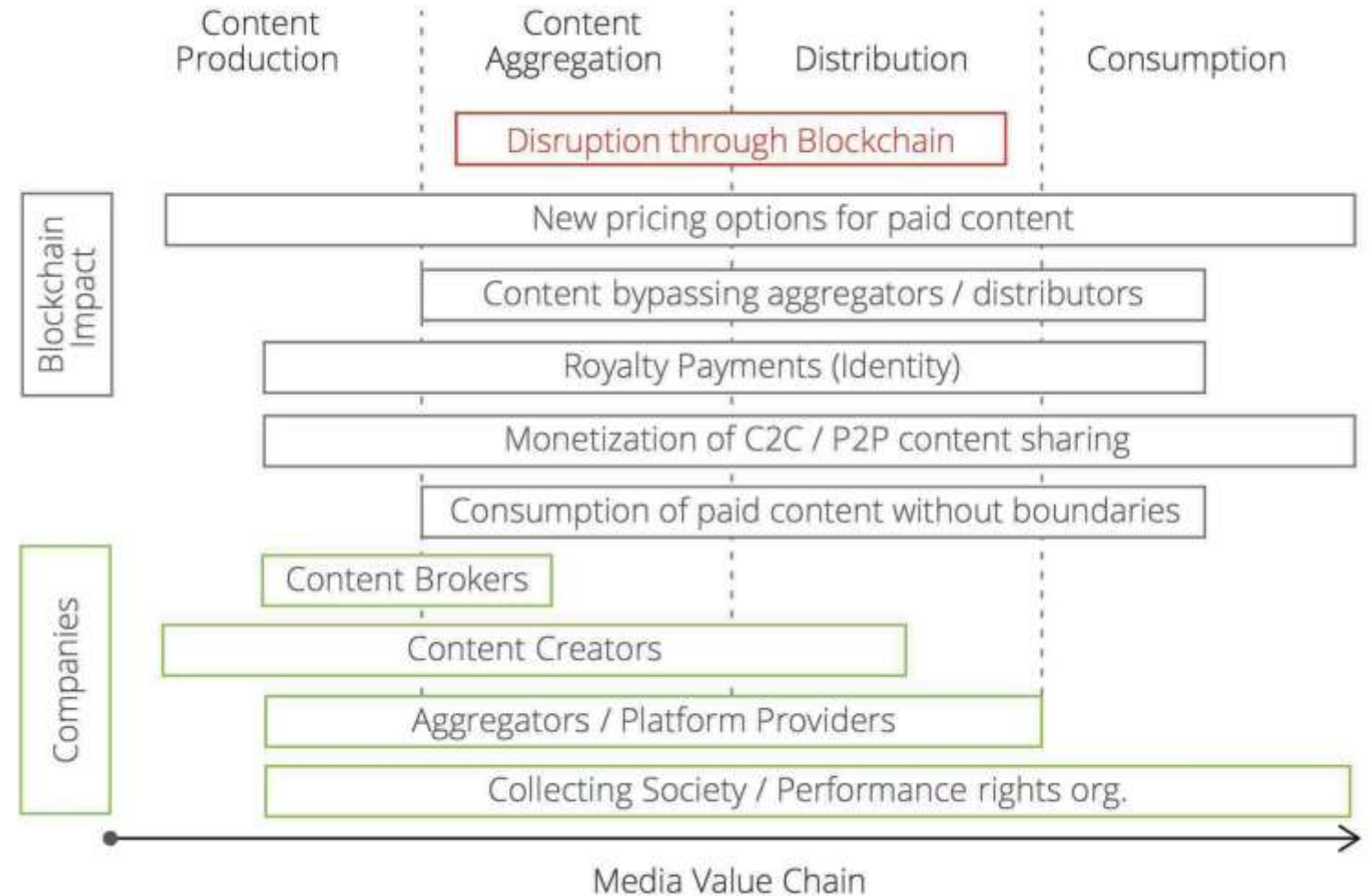


Remove Cost

Overheads on maintaining and synchronizing silos.

Disrupting the Media Value Chain

Introduction of Blockchain will result in **deprecation** of critical roles/operations played by some of the **Business Entities**.

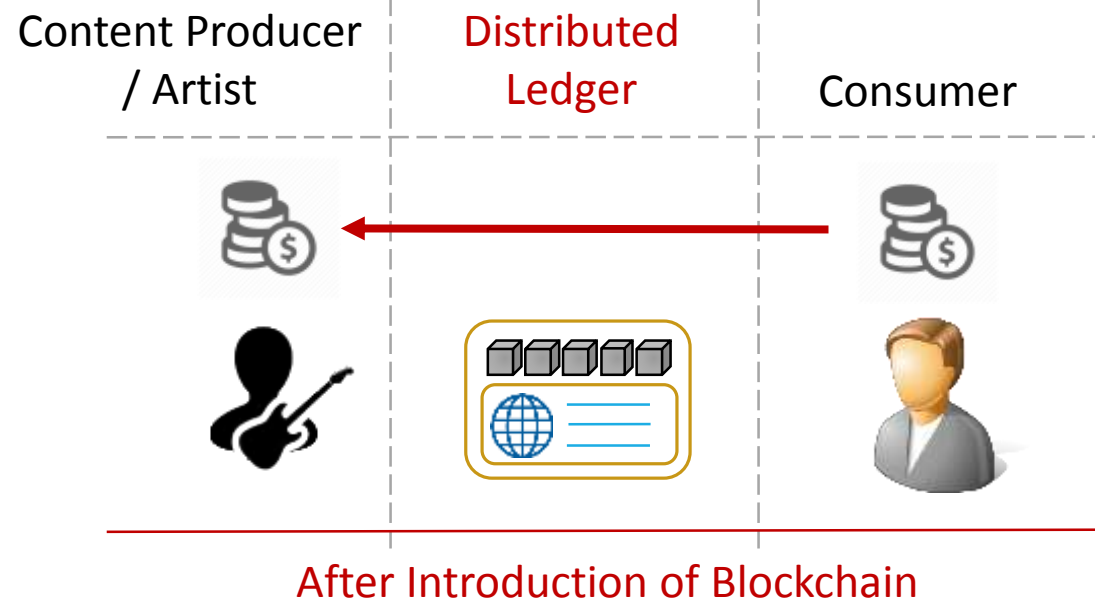
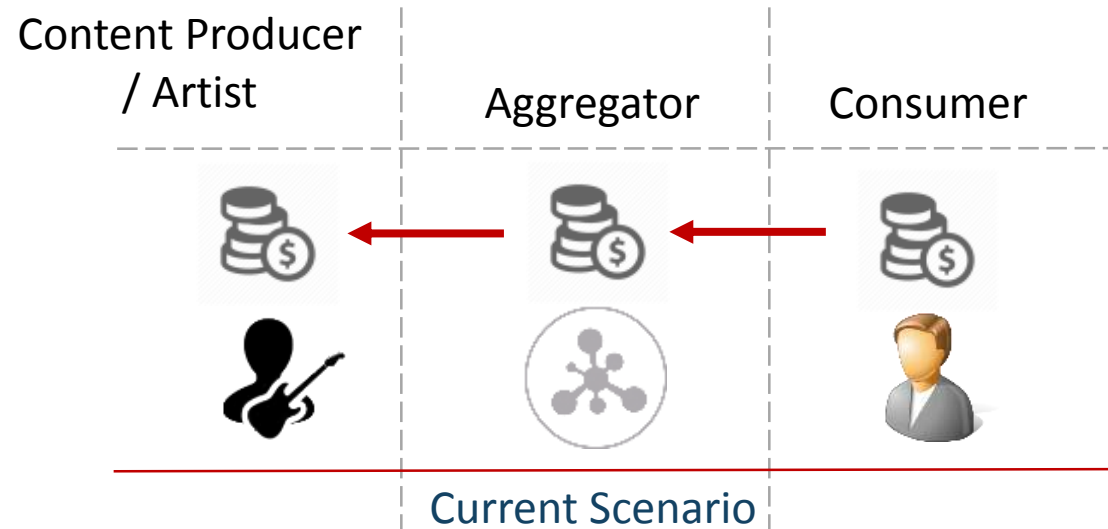


Blockchain's primary relevance in the media value chain

(c) Deloitte : Blockchain @ Media
A new Game Changer for the Media Industry

Content Bypassing the Aggregators

Source: Deloitte : [Blockchain @ Media](#)



Challenges

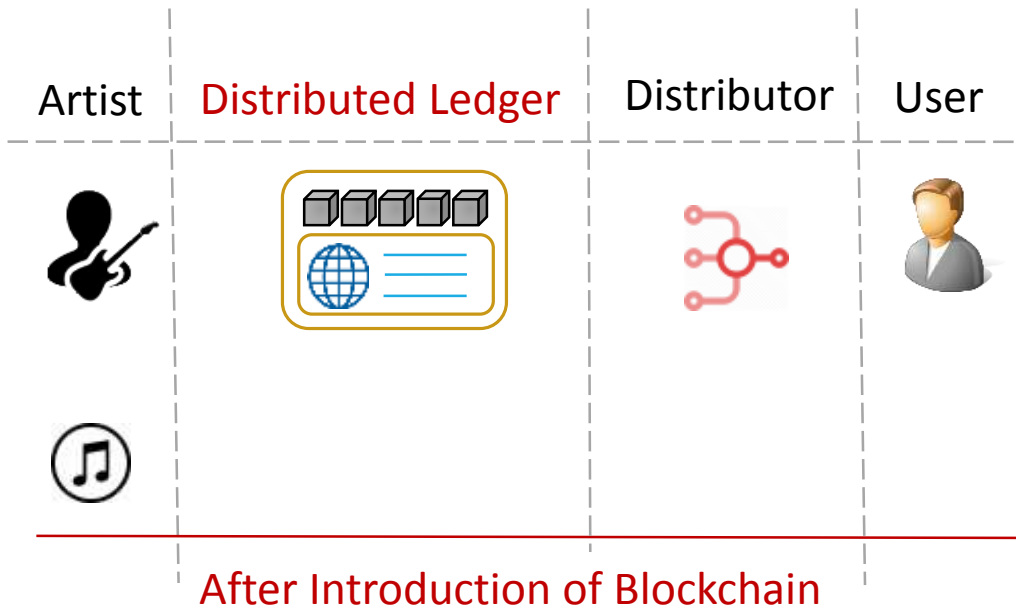
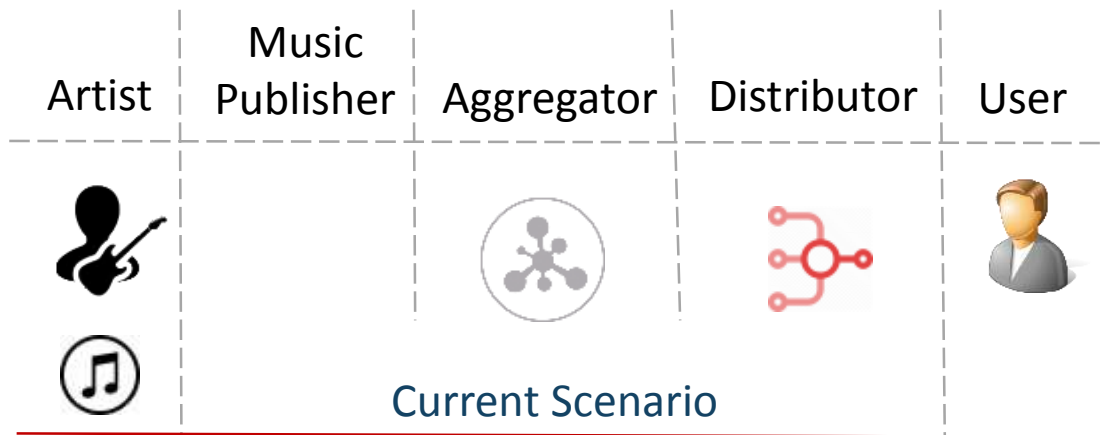
- Content Aggregators and Advertising networks to loose their dominant position.
- Blockchain has the potential to make DRM systems obsolete or at least to reduce it's complexity

Benefits

- Direct Customer Artist relationship.
- Easy to measure marketing and performance.

Distribution of Royalty Payments

Source: Deloitte : [Blockchain @ Media](#)



Challenges

- Handling large amount of Data
- All the parties agreeing to common Blockchain Platform.

Benefits

- Near Real-time distribution of Royalty payments based on Smart Contracts
- Every usage and consumption is tracked in Blockchain

References

Blockchain Video Tutorials

- Hyperledger Fabric Explainer: <https://www.youtube.com/watch?v=js3Zjxbo8TM>
- Blockchain Technology : <https://www.youtube.com/watch?v=qOVAbKKSH10>
- Smart Contracts : <https://www.youtube.com/watch?v=ZE2HxTmxfrl>
- Hyperledger Fabric v1.1 Node.JS Chaincode : <https://www.youtube.com/watch?v=dzwR0dwzXNs>
- Hyperledger Fabric v1.0 : <https://www.youtube.com/watch?v=6nGlptzBZis>
- Hyperledger Fabric for Beginners : <https://www.youtube.com/watch?v=Y177TCUc4g0>
- Fabric Composer : <https://www.youtube.com/watch?v=fdFUsrv5iw>
- Hyperledger Composer : https://www.youtube.com/watch?v=iRIm4uY_9pA
- How does it works: Blockchain : <https://www.youtube.com/watch?v=ID9KAnkZUjU>
- How does Blockchain work: <https://www.youtube.com/watch?v=LZEH0lZY2To>
- 19 Industries The Blockchain will disrupt : <https://www.youtube.com/watch?v=G3psxs3gyf8>
- What is the difference between Bitcoin & Blockchain? : <https://www.youtube.com/watch?v=MKwa-BqnJDg>
- [Bitcoin: What Bill Gates, Buffet, Elon Mush & Richard Branson has to say about Bitcoin?](#)



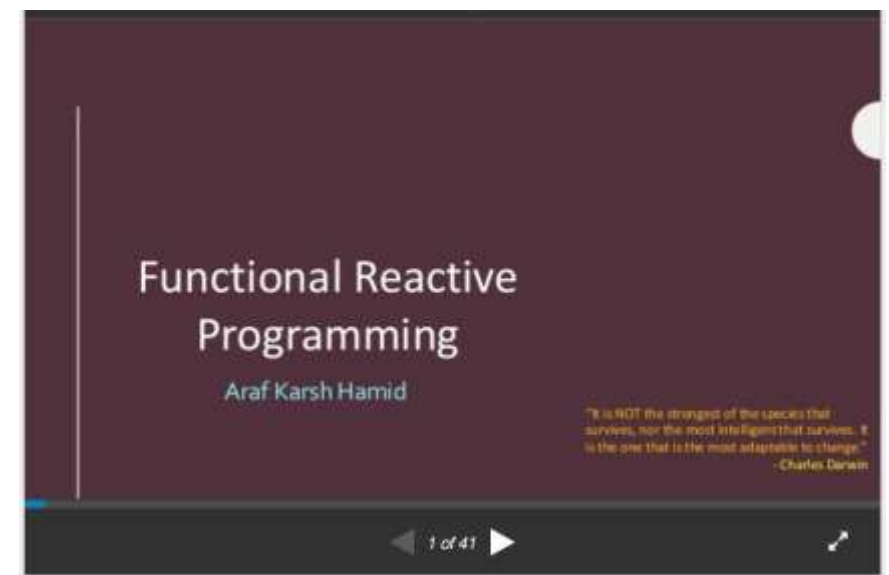
References

1. [HyperLedger Architecture Volume 1](#)
2. Creating Tokens in Hyperledger Fabric : <https://github.com/Kunstmaan/hyperledger-fabric-kuma-token-example/>
3. Karame, Androulaki, & Capkun. (2012). Double spending fast payments in bitcoin. *ACM Conference on Computer and communications security*, (pp. 906-917).
4. King, N. (2012). PPCoin: Peer to peer cryptocurrency with proof of stake.
5. Kwon. (2014). *Tendermint: Consensus without Mining*. Cornell Edu.
6. Mazieres, D. (2015). *The Stellar Consensus Protocol: Federated Model for Internet level consensus*. Stellar Development Foundation.
7. Narayanan, B. F. (2016). *Bitcoin and cryptocurrency technologies*. Princeton: Princeton University Press.
8. O'Dwyer, & Malone. (2014). Bitcoin mining and its energy footprint. *Irish Signals and Systems Conference*, pp. 280-285
9. Byzantine Fault Tolerance : https://en.wikipedia.org/wiki/Byzantine_fault_tolerance
10. Building Blockchain Apps for Node.JS developers with HyperLedger Composer – Simon Stone IBM

My Slide share Presentations

1. Enterprise Software Architecture – <http://www.slideshare.net/arafkarsh/software-architecture-styles-64537120>
2. Functional Reactive Programming – <https://www.slideshare.net/arafkarsh/functional-reactive-programming-64780160>
3. Microservices – Part 1: <https://www.slideshare.net/arafkarsh/micro-services-architecture-80571009>
4. Event Storming and Saga Design Pattern – Part 2 : <https://www.slideshare.net/arafkarsh/event-storming-and-saga>
5. Docker and Linux Containers : <https://www.slideshare.net/arafkarsh/docker-container-linux-container>
6. Function Point Analysis : <http://www.slideshare.net/arafkarsh/functional-reactive-programming-64780160>

Thank you



Araf Karsh Hamid : Co-Founder / CTO : araf.karsh@metamagic.in
USA: +1 (201) 616 0780 India: +91.999.545.8627 Skype / LinkedIn / Twitter / Slideshare : arafkarsh
<http://www.slideshare.net/arafkarsh> | <https://www.linkedin.com/in/arafkarsh/>

