

# Hyperledger – Fabric

Gallersdörfer, U., Holl, P., & Matthes, F. (2020). “Blockchain-based Systems Engineering”. Lecture Slides. TU Munich.

Chair of Software Engineering for Business Information Systems (sebis)  
Faculty of Informatics  
Technische Universität München  
[www.matthes.in.tum.de](http://www.matthes.in.tum.de)

## 1. Recap

- Motivation
- Exemplary use case

## 2. Fabric

- Transaction flow
- Channels
- Single channel networks
- Multi channel networks
- State database

## 3. Composer

- Motivation

# Motivation for consortium Blockchains

## The problem

Multiple parties that do not fully trust each other need to share information in a tamper proof and secure way.

## Challenges

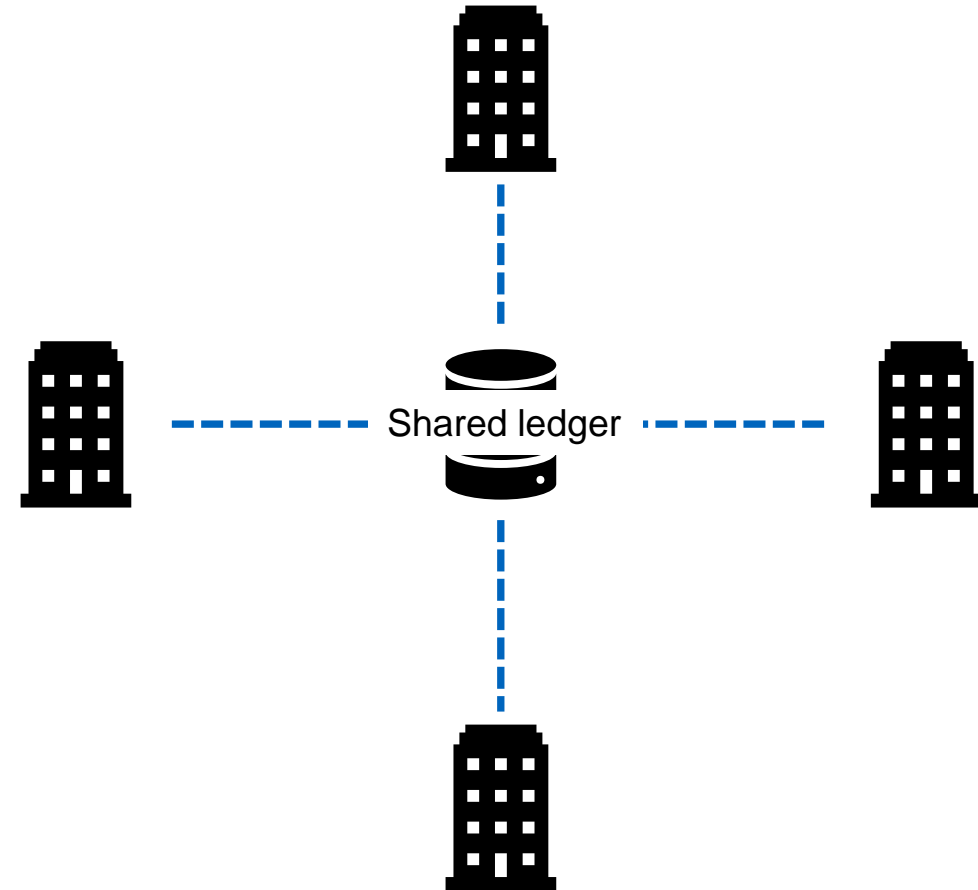
Who owns and operates the ledger?

Who pays for the shared data structure?

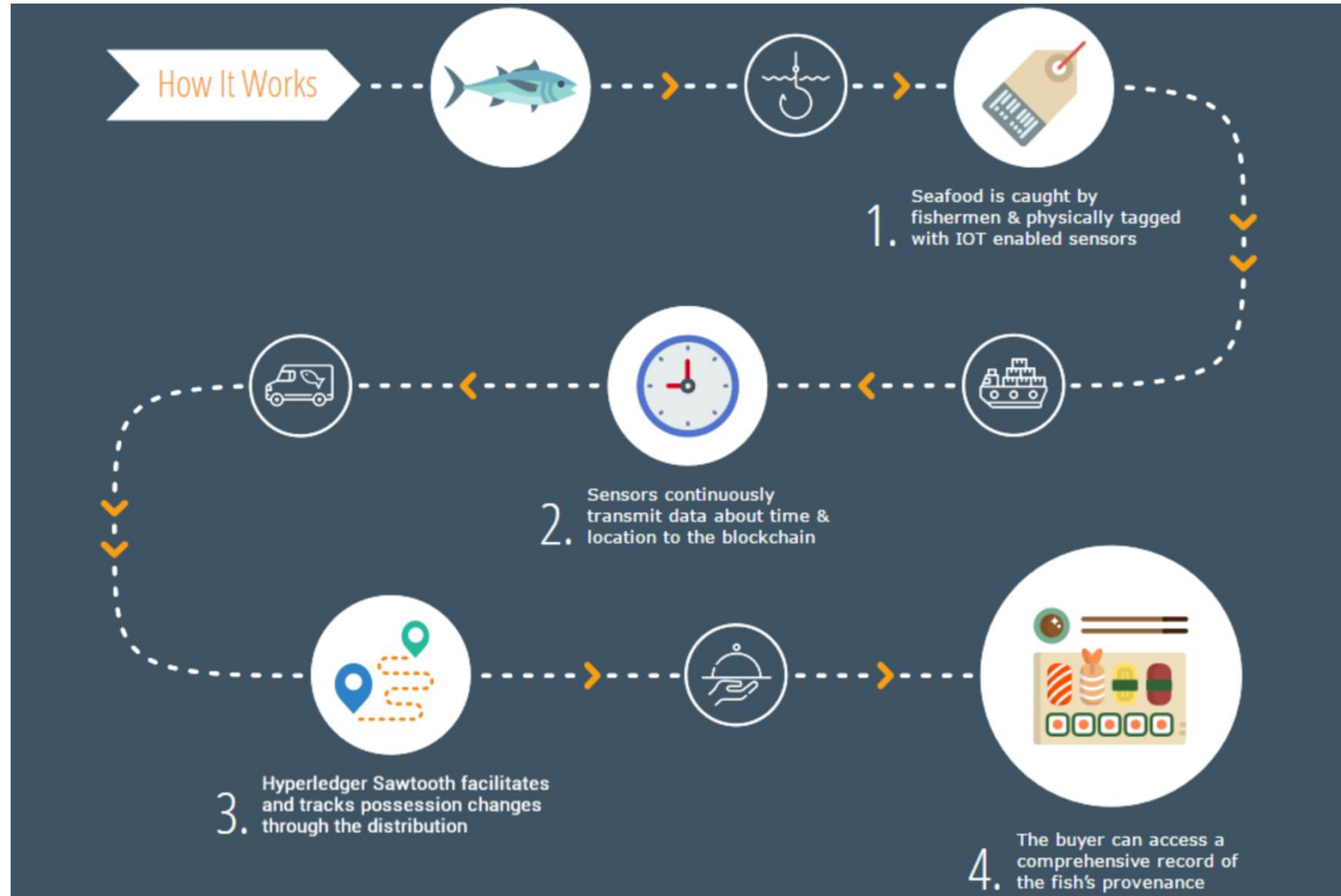
What are allowed operations on the shared data structure?

What is the process of adding new parties?

How is consensus achieved?



# Example: Supply chain of a fish market



Source: <https://www.hyperledger.org/projects/sawtooth/seafood-case-study>



Hyperledger Sawtooth is a modular platform for building, deploying, and running distributed ledgers. Hyperledger Sawtooth includes a novel consensus algorithm, Proof of Elapsed Time (PoET), which targets large distributed validator populations with minimal resource consumption.

» [LEARN MORE](#)



Hyperledger Iroha is a business blockchain framework designed to be simple and easy to incorporate into infrastructural projects requiring distributed ledger technology.

» [LEARN MORE](#)



Intended as a foundation for developing applications or solutions with a modular architecture, Hyperledger Fabric allows components, such as consensus and membership services, to be plug-and-play.

» [LEARN MORE](#)



Hyperledger Burrow is a permissionable smart contract machine. The first of its kind when released in December, 2014, Burrow provides a modular blockchain client with a permissioned smart contract interpreter built in part to the specification of the Ethereum Virtual Machine (EVM).

» [LEARN MORE](#)



Hyperledger Indy is a distributed ledger, purpose-built for decentralized identity. It provides tools, libraries, and reusable components for creating and using independent digital identities rooted on blockchains or other distributed ledgers for interoperability.

» [LEARN MORE](#)

Source: <https://www.hyperledger.org/projects>



## 1. Recap

- Motivation
- Exemplary use case

## 2. Fabric

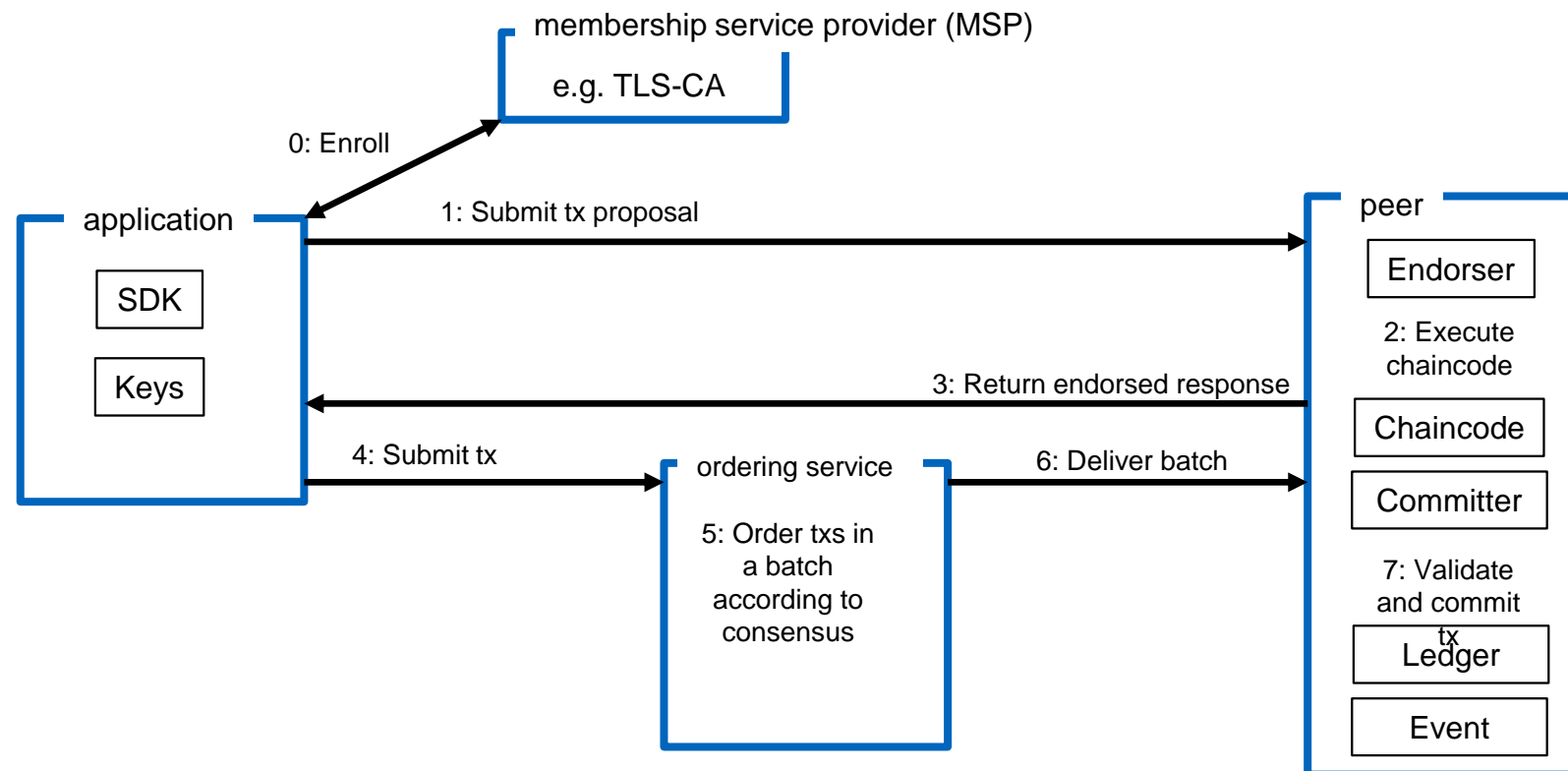
- Transaction flow
- Channels
- Single channel networks
- Multi channel networks
- State database

## 3. Composer

- Motivation

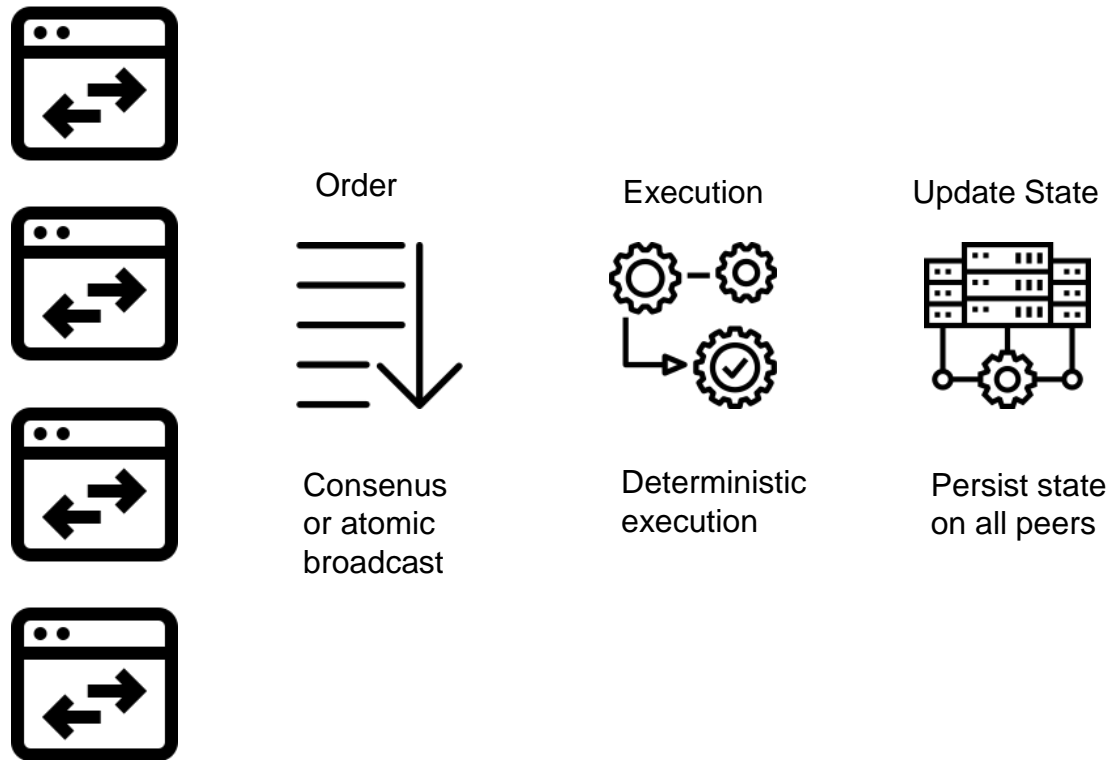
# Big picture of a transaction

In Fabric, a transaction initiates a seven step process from **simulation** of the executed chaincode (endorsement) to the **generation** of a read/write set which is then broadcasted to the ordering service and finally included in a new block (**committing**).



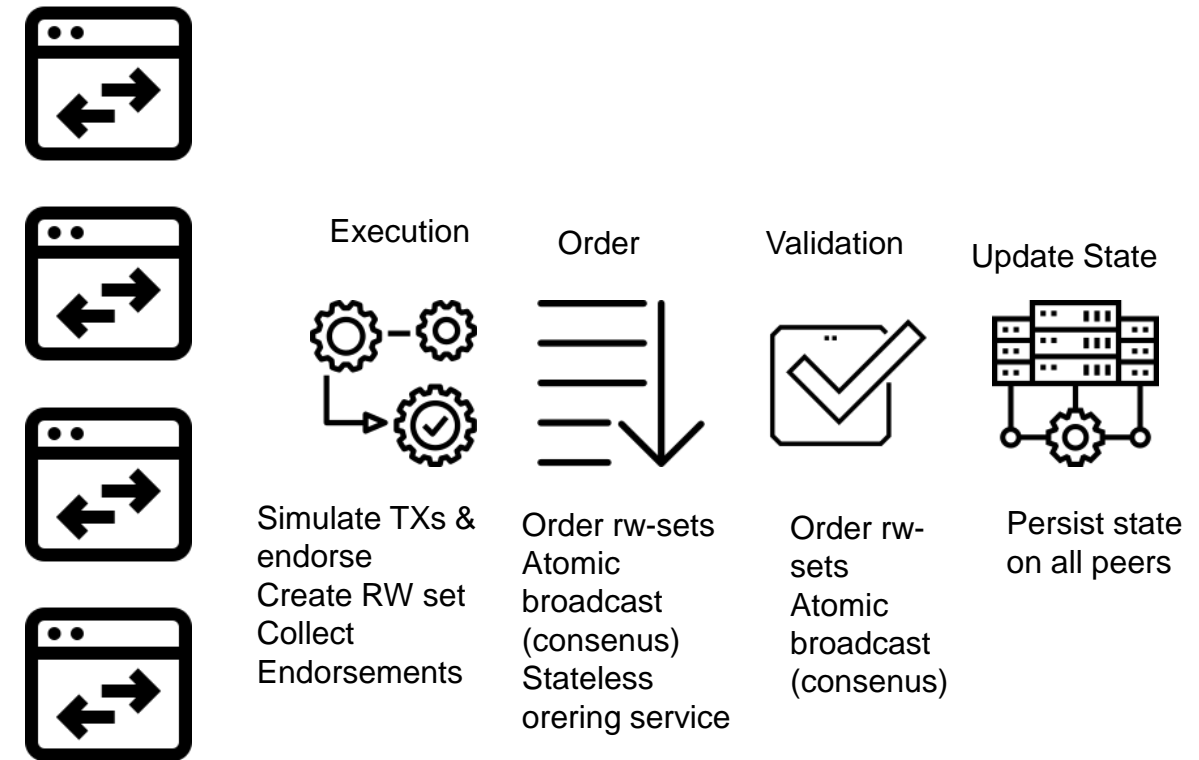
# Software Architecture Order-Execution vs. Execution-Order-Validate

## Order-Execution



## Execution-Order-Validate

Parallel execution of unrelated operations





## Sequential execution on all peers

Executing the transactions sequentially on all peers **limits** the effective **throughput** that can be achieved by the Blockchain. In particular, since the throughput is inversely proportional to the execution latency, this may become a performance bottleneck for all but the simplest smart contracts.

Only endorsing peers execute Proposed TXs

## Non-deterministic code

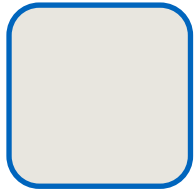
Agreement on the same state is an essential property of current Blockchain Systems. A single non-deterministic operation could have a devastating effect. This issue is addressed by programming Blockchains in domain-specific languages like Solidity in Ethereum. Those languages are usually not as expressive and require additional learning by the programmer.

Applications written in general purpose PL need to follow endorsement policies. If RW-sets differ → Rejection by endorsing peers

## Confidentiality of execution

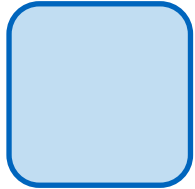
It often suffices to propagate the same state to all peers instead of running the same code everywhere. Because everyone has to execute everything in the Order-Execution cycle, inherent confidentiality is not provided. Confidentiality can only be achieved through additional data encryption leading to considerable overhead.

Only a few trusted endorsing peers execute the TXs against the chaincode, the rest just updates the state (RW set)



## **Committing Peer**

Maintains the state of the Blockchain and commits transactions. It verifies endorsements and validate the results of a transaction.



## **Endorsing Peer**

An endorsing peer is always also a committing peer. The difference is that an endorsing peer additionally takes transaction proposals and executes them to create endorsements.

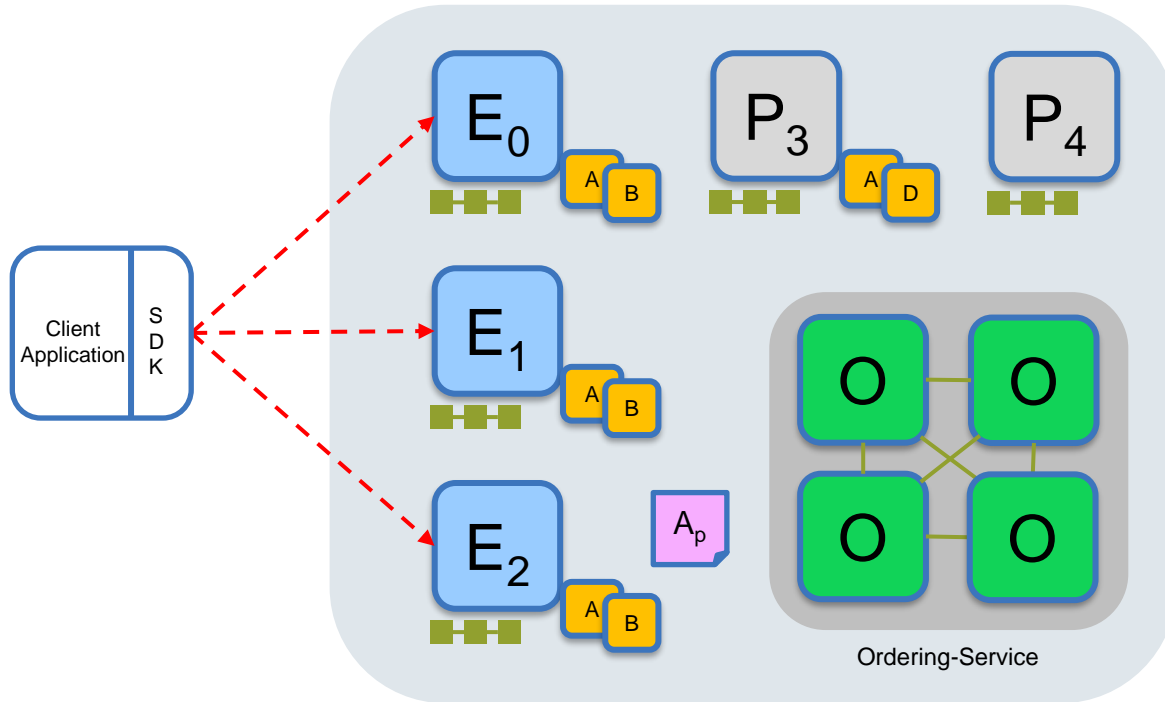


## **Ordering service node**

Applications must submit transactions to an ordering service node. The node then collects transactions and orders them sequentially. The transactions are then put into a new block and delivered to all peers of a specific channel. Does neither hold ledger nor chaincode.

# Transaction flow: Step 1 / 7

Application submits transaction proposal

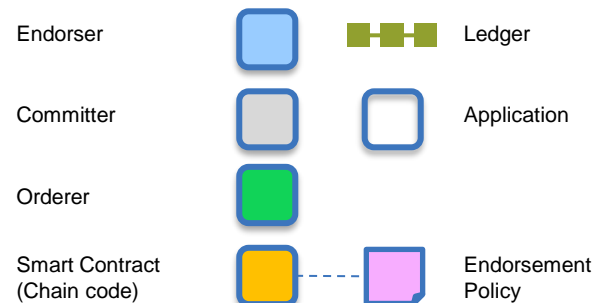


Application creates a transaction proposal for **chaincode A** and sends it to all peers that are part of the endorsement policy  $A_p$

Endorsement Policy  $A_p$

- $E_0$ ,  $E_1$  and  $E_2$  must sign the transaction
- ( $P_3$ ,  $P_4$  are not part of the policy)

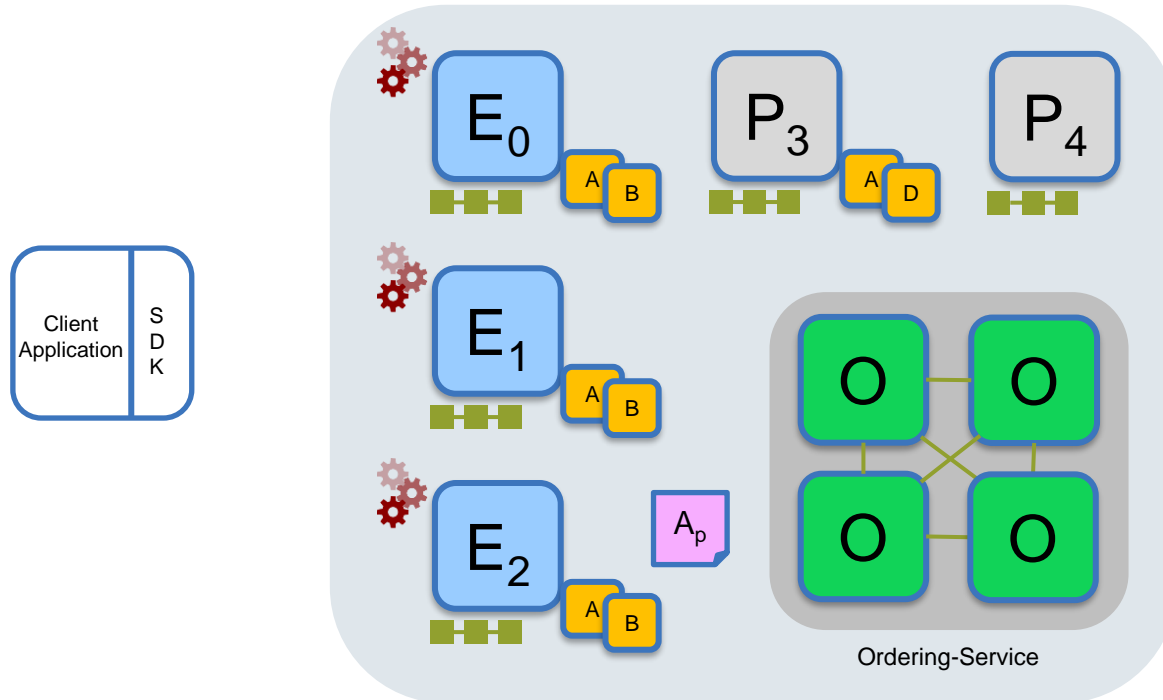
Since only the peers  $E_0$ ,  $E_1$  and  $E_2$  are part of the endorsement policy  $A_p$ , it is not required to send the transaction proposal to  $P_3$  and  $P_4$



Source: <https://www.hyperledger.org/resources/universities#educators>

# Transaction flow: Step 2 / 7

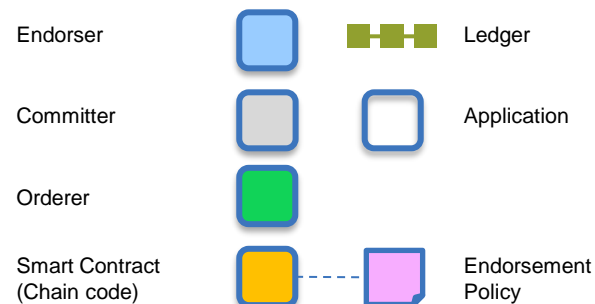
Endorsing peers execute the transaction proposal



$E_0$ ,  $E_1$  and  $E_2$  will each simulate the execution of the *proposed* transaction from the application.

None of these executions will update the ledger.

The **simulation will be used to capture** the read and write operations on the ledger. After the transaction is executed, each peer will have a generated **read/write set** (RW set)

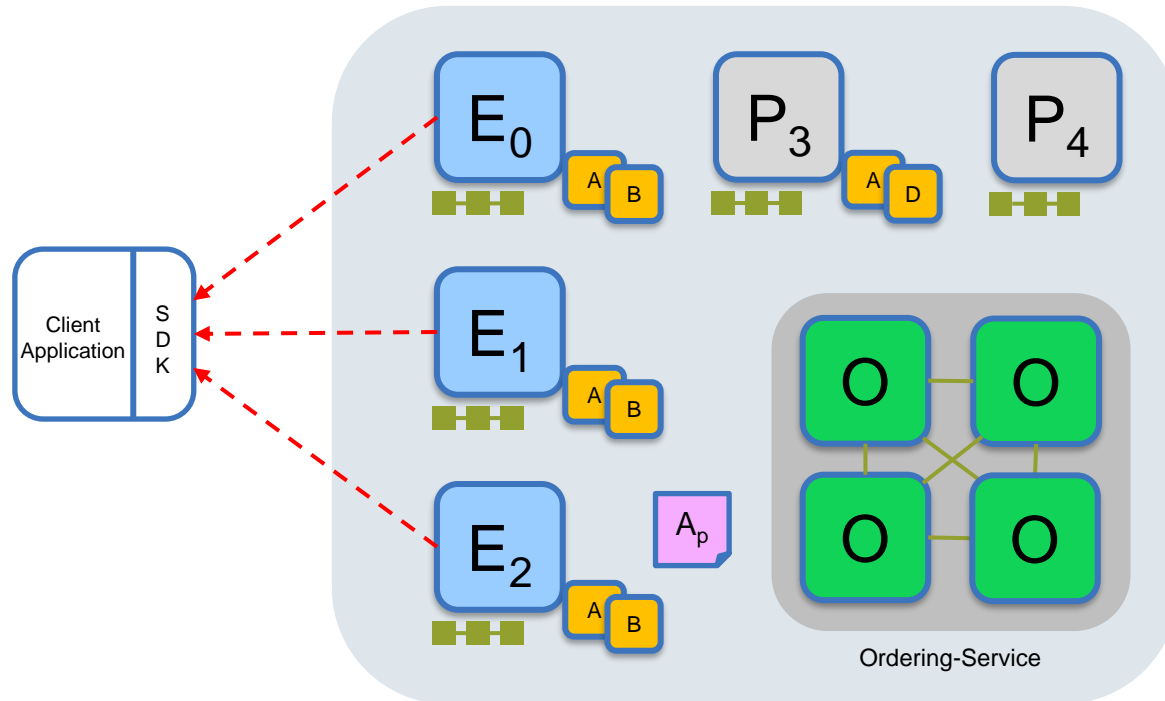


```
<TxReadWriteSet>
<NsReadWriteSet name="chaincode1">
  <ReadSet>
    <read key="K1", version="1">
    <read key="K2", version="1">
  </ReadSet>
  <WriteSet>
    <write key="K1", value="V1">
    <write key="K3", value="V2">
    <write key="K4", isDelete="true">
  </WriteSet>
</NsReadWriteSet>
</TxReadWriteSet>
```

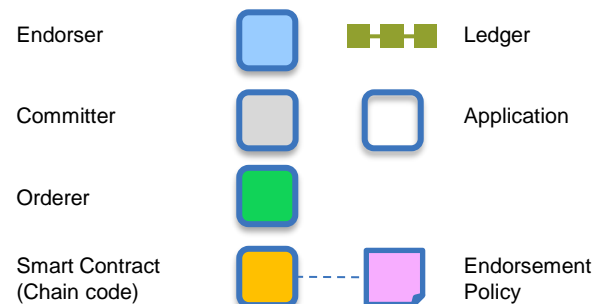
Source: <https://www.hyperledger.org/resources/universities#educators>

# Transaction flow: Step 3 / 7

Read/write set is signed and returned to the application



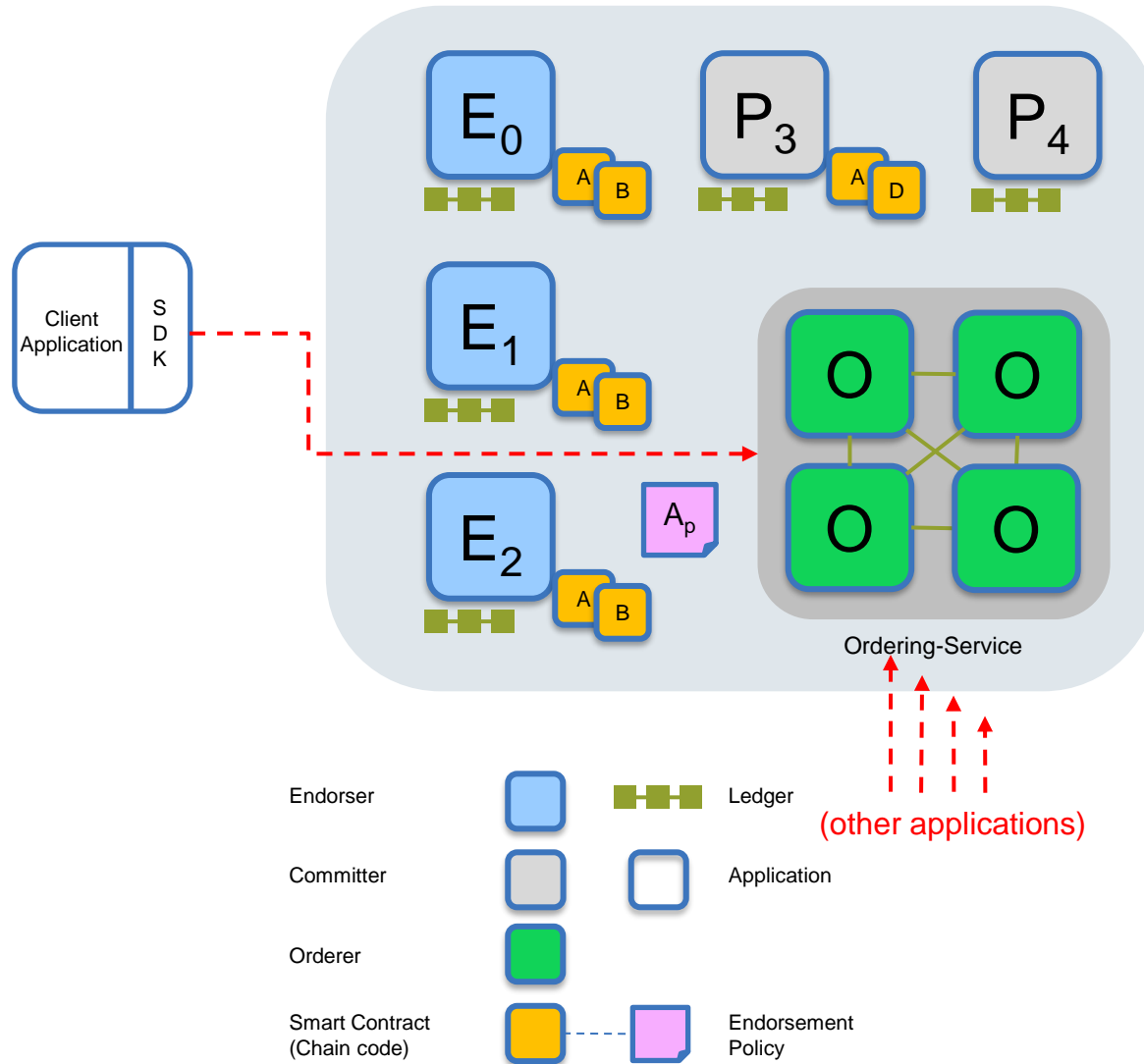
E<sub>0</sub>, E<sub>1</sub> and E<sub>2</sub> will each sign their generated read/write set and return it to the application that invoked the transaction.



Source: <https://www.hyperledger.org/resources/universities#educators>

# Transaction flow: Step 4 / 7

Singed responses are sent to the ordering service



The application submits the signed responses from  $E_0$ ,  $E_1$  and  $E_2$  to the ordering service.

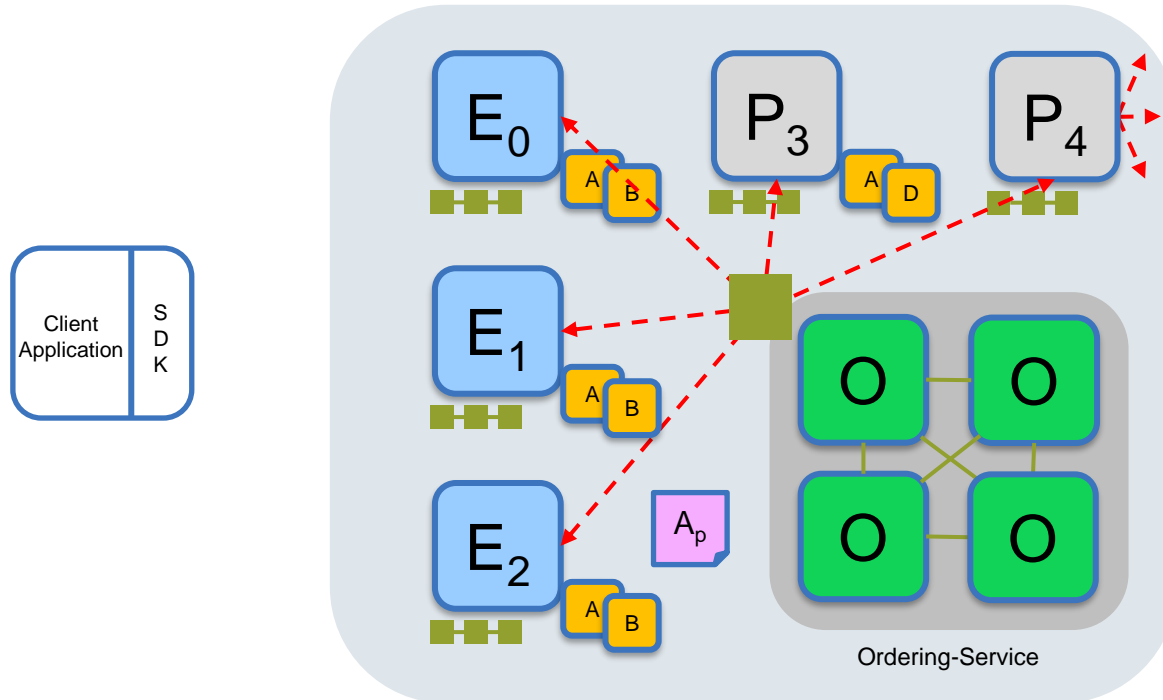
The ordering service is responsible to order all transactions from all applications in the network. The service tries to **serialize** the incoming transactions.

Source: <https://www.hyperledger.org/resources/universities#educators>



# Transaction flow: Step 5 / 7

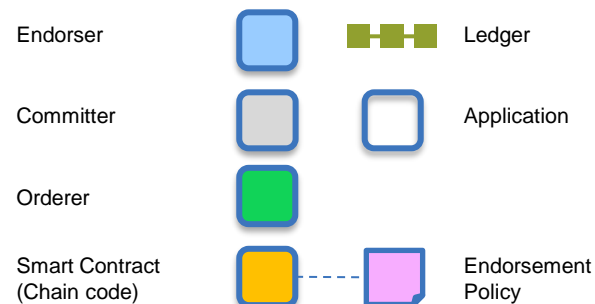
Ordering service distributes new block to all committing peer



The ordering service creates a new block based on the incoming transactions. The block will then be broadcasted to all committing peers in the channel.

Currently, the ordering service supports three different ordering algorithms:

- SOLO (single node, development)
- Kafka (blocks map to topics)
- SBFT (simple byzantine fault tolerance, tolerates faulty peers, future)



Source: <https://www.hyperledger.org/resources/universities#educators>

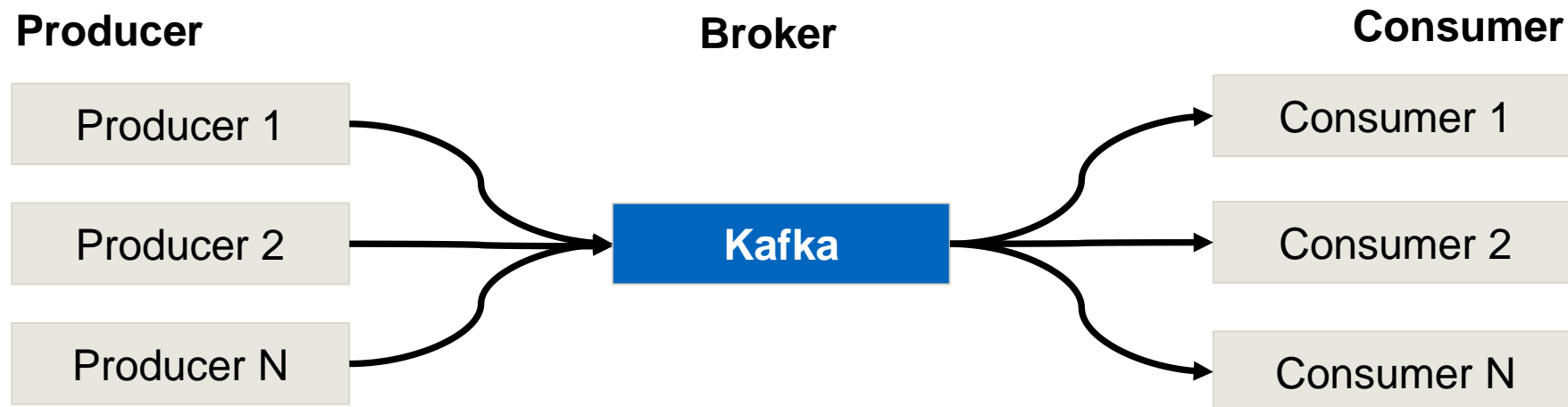
Currently, the most used ordering service implementation uses Apache Kafka.

Kafka is a stand-alone distributed streaming platform that is built for data engineering purposes.

Official resources describe the key capabilities of Kafka as follows:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

Kafka acts as a broker for data streams.

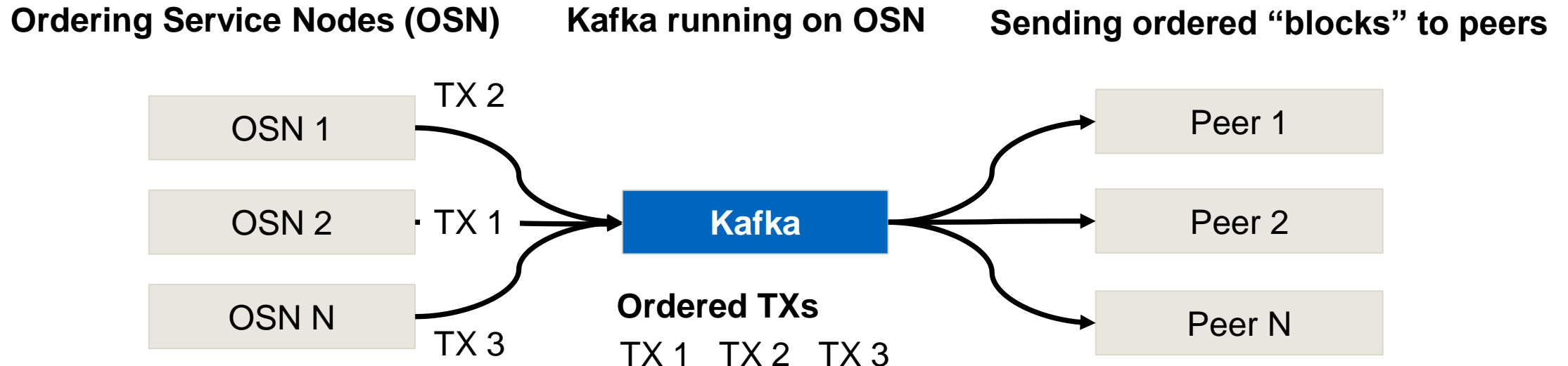


A consumer subscribes to messages that are published by one or more producers. Messages sent by the producers can be grouped and categorized into so-called topics.

Depending on the number and size of the messages that belong to a topic, Kafka partitions the topic and guarantees that all messages that belong to the topic are sequentially ordered. The messages are temporally ordered by the time they reach the Kafka broker.

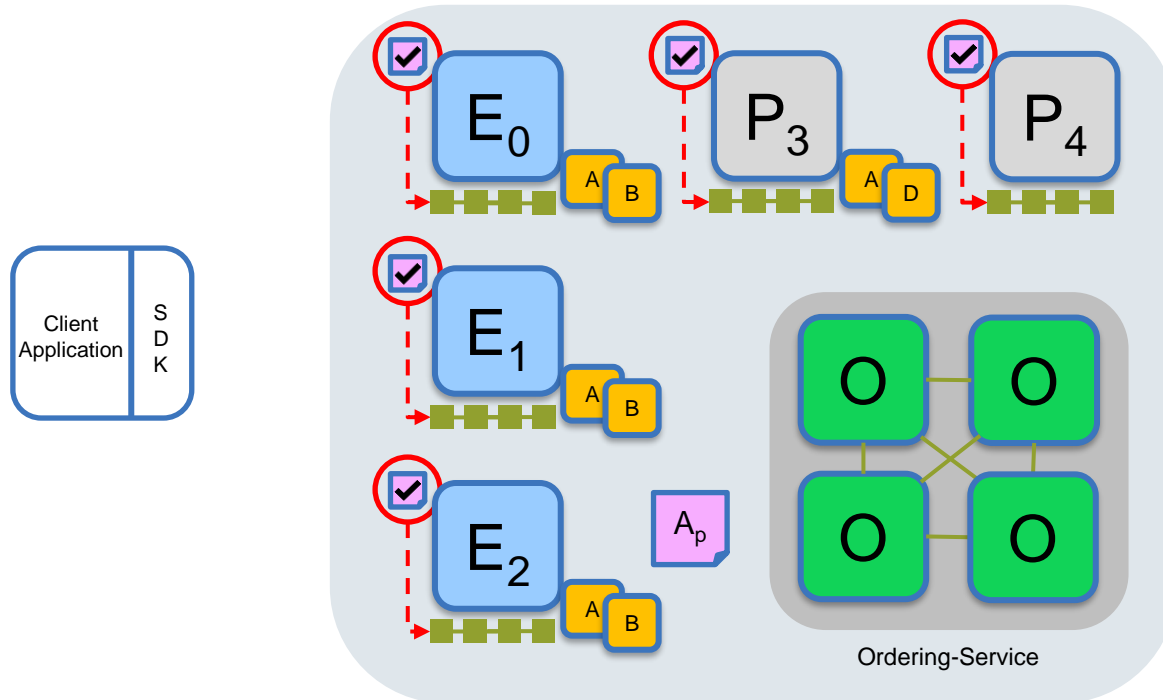
### Kafka in Fabric

Each chain maps to a certain topic in Kafka



# Transaction flow: Step 6 / 7

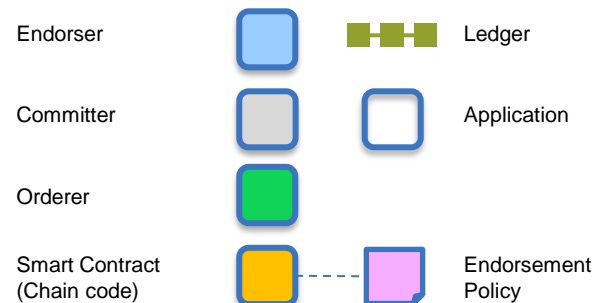
Committing peers validate the transaction



All committing peers in the channel validate the transaction (read/write set) according to the endorsement policy of the chaincode A.

If the transaction is valid, the **read** and write set is written to the ledger and added as a new block to the Blockchain.

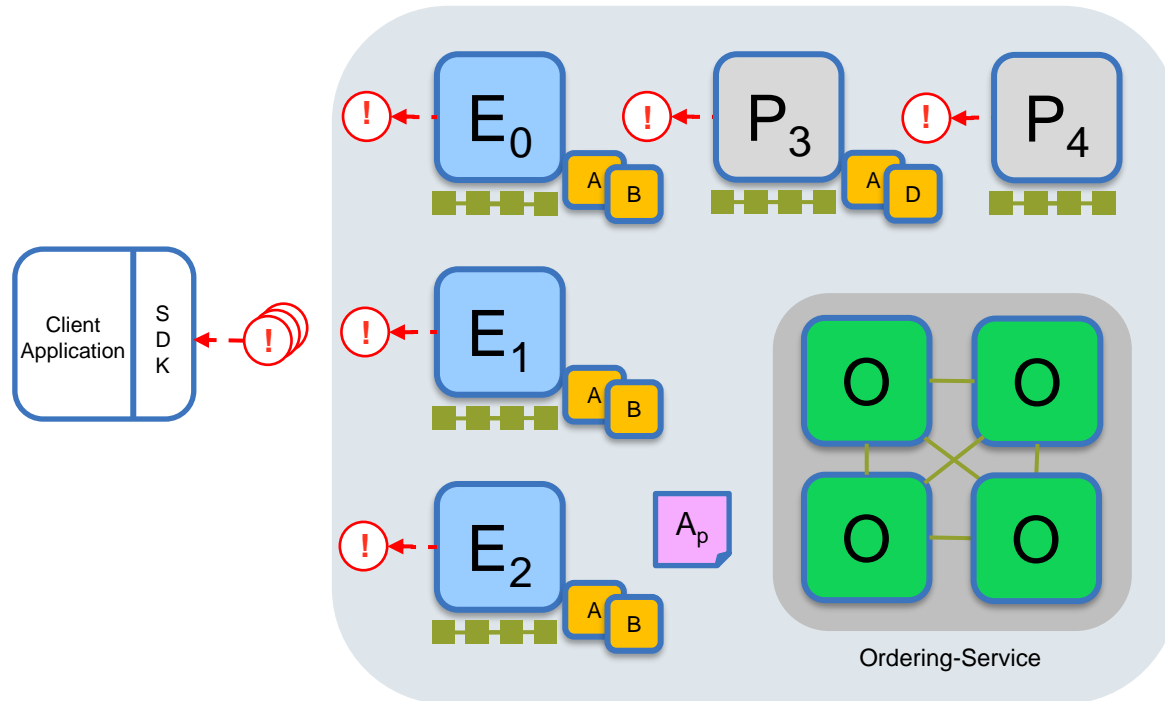
The databases used for caching are updated with the new state information accordingly.



Source: <https://www.hyperledger.org/resources/universities#educators>

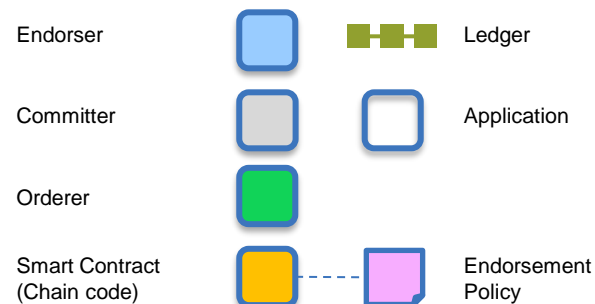
# Transaction flow: Step 7 / 7

## Notify application

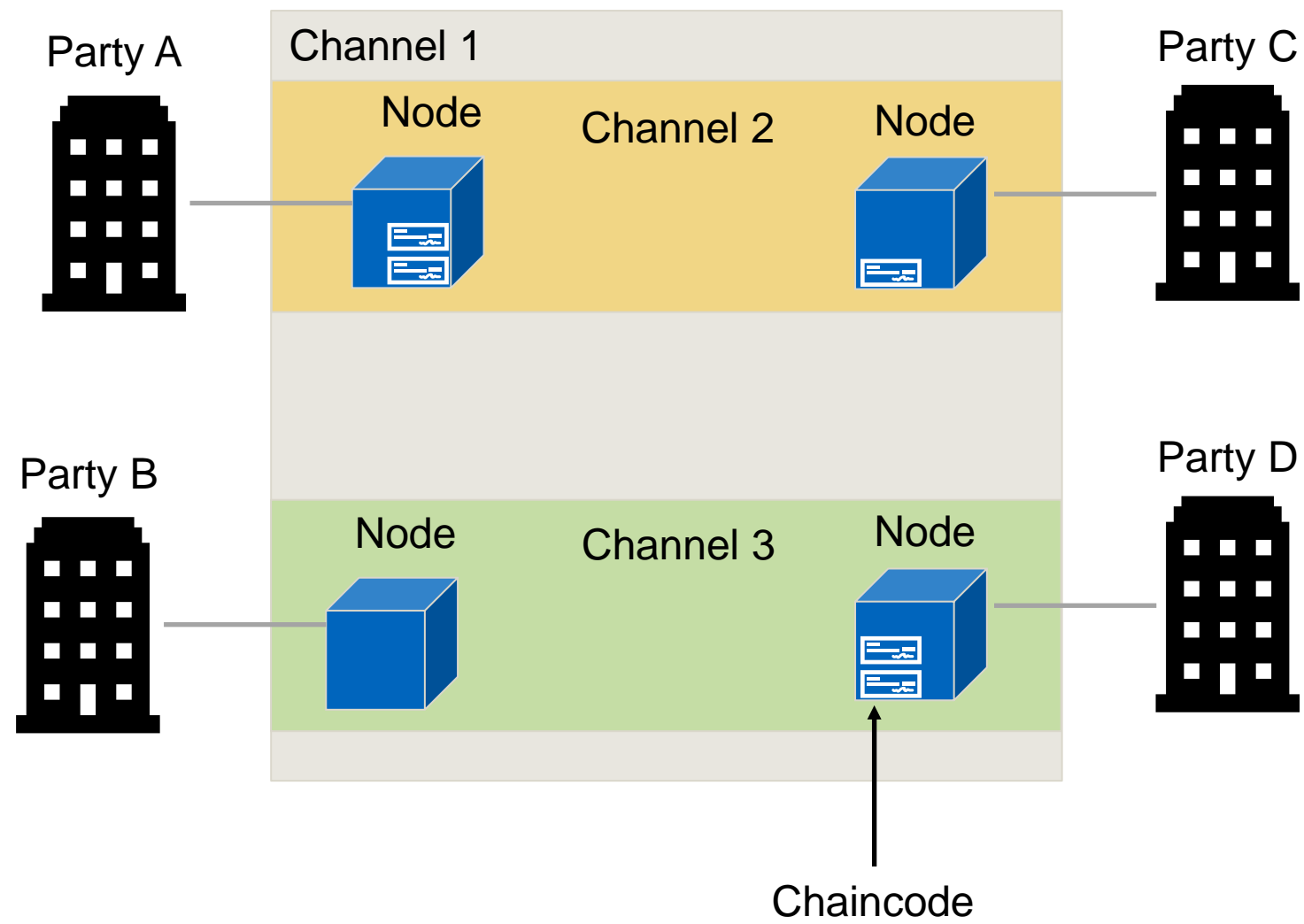


Applications can register to be notified by the peers once the transaction is done. The peers will emit an event that indicates if the transaction succeeded or failed.

Applications can also subscribe to state changes of the ledger, i.e. a connected peer will then notify them if new blocks are added to the chain.



Source: <https://www.hyperledger.org/resources/universities#educators>



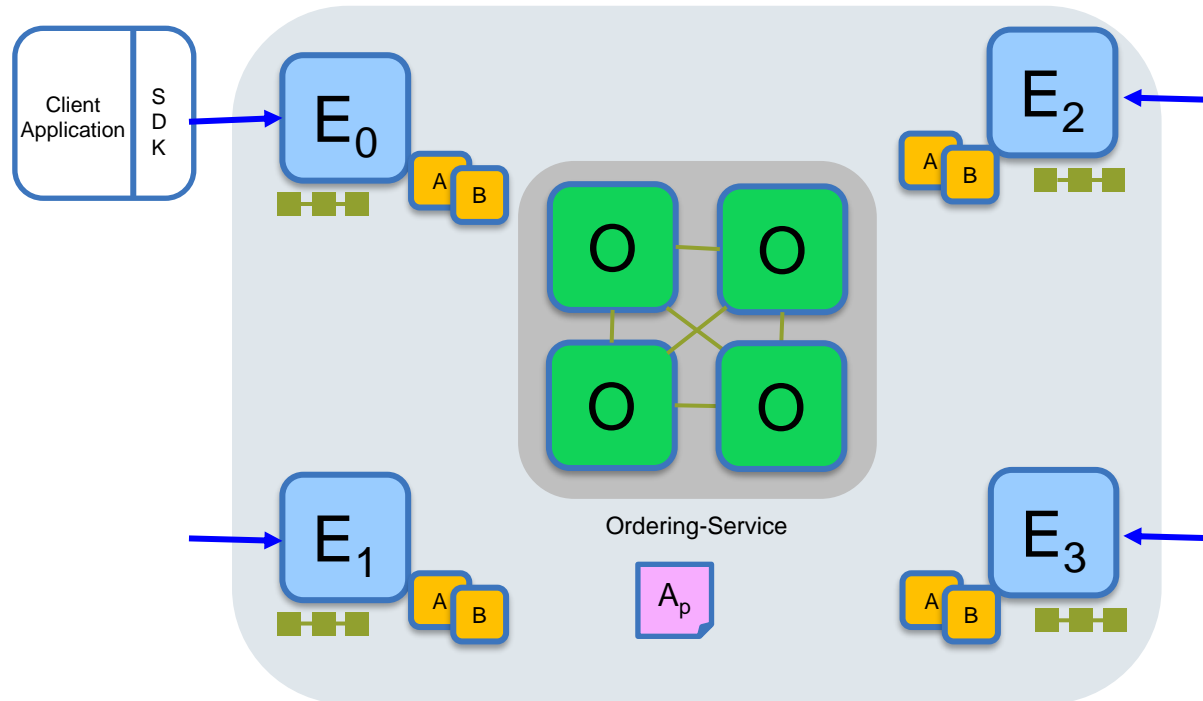


### **A channel is a separate Blockchain.**

This Blockchain is only managed by a subset of all available nodes as defined by the membership service provider.

- Separate channels isolate transactions on different ledgers
  - Consensus takes place within a channel by members of the channel
    - Other members on the network are not allowed to access the channel and will not see transactions on the channel
  - A chaincode may be deployed on multiple channels with each instance isolated within its channel
  - Peers can participate on multiple channels
- ➔ Concurrent execution for performance and scalability

# Single channel network

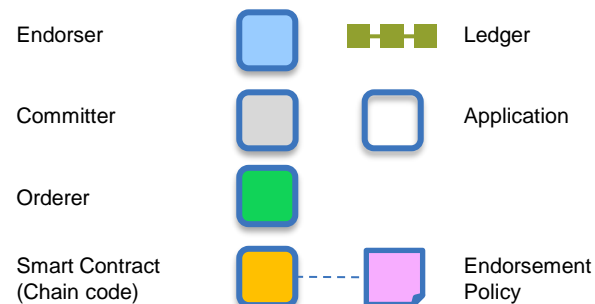


All peers connected to the same channel (**blue**)

All peers have the same chaincode and maintain the same ledger

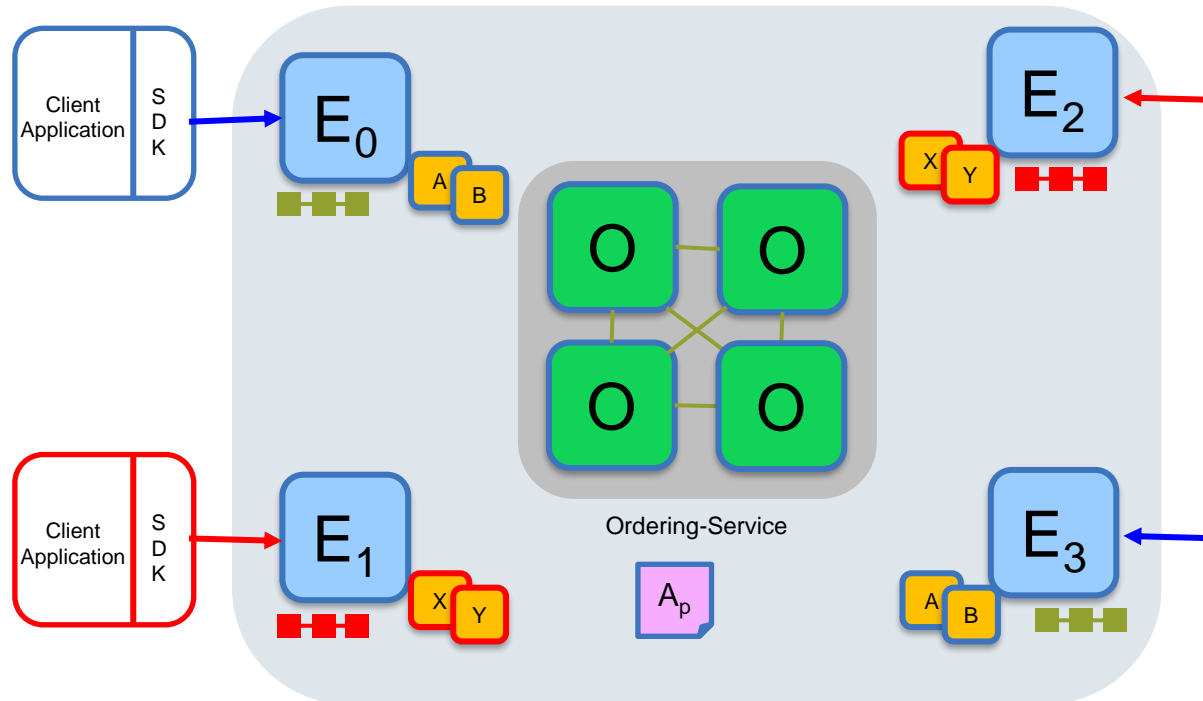
Endorsements by peers  $E_0$ ,  $E_1$ ,  $E_2$ , and  $E_3$

A single channel network is similar to traditional, public Blockchain network where the whole world state is shared with all participating nodes.



Source: <https://www.hyperledger.org/resources/universities#educators>

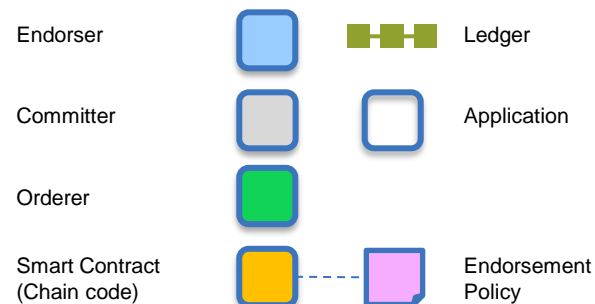
# Multi channel network



Peers  $E_1$  and  $E_2$  connect to the **red** channel for **X** and **Y** chaincodes.

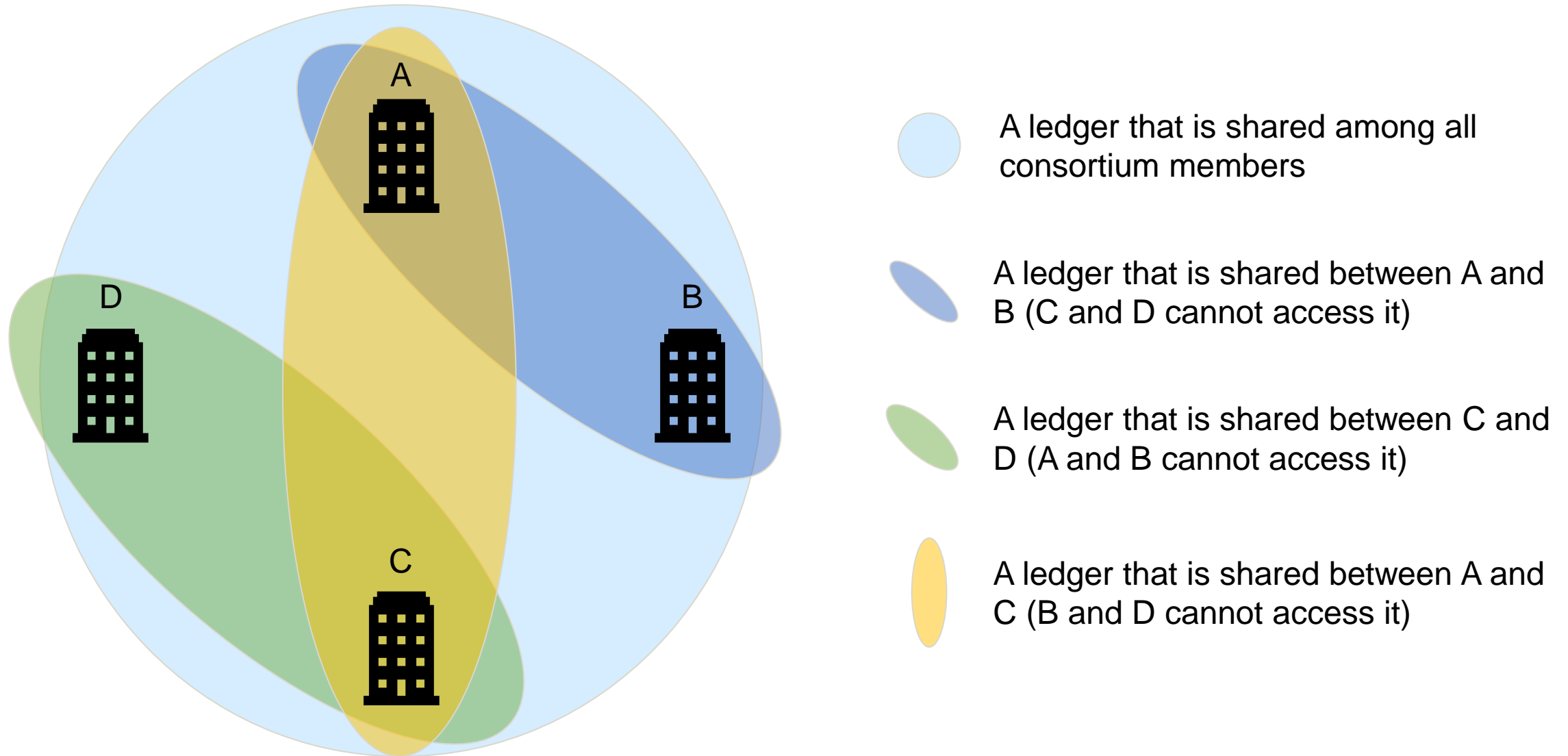
Peers  $E_0$  and  $E_3$  connect to the **blue** channel for **A** and **B** chaincodes.

Fabric support multi channel networks. Each peer only shares the ledger with nodes that are in the same channel. Smart contracts also operate on a channel basis and are not globally available.



Source: <https://www.hyperledger.org/resources/universities#educators>

# Example of a multi channel network

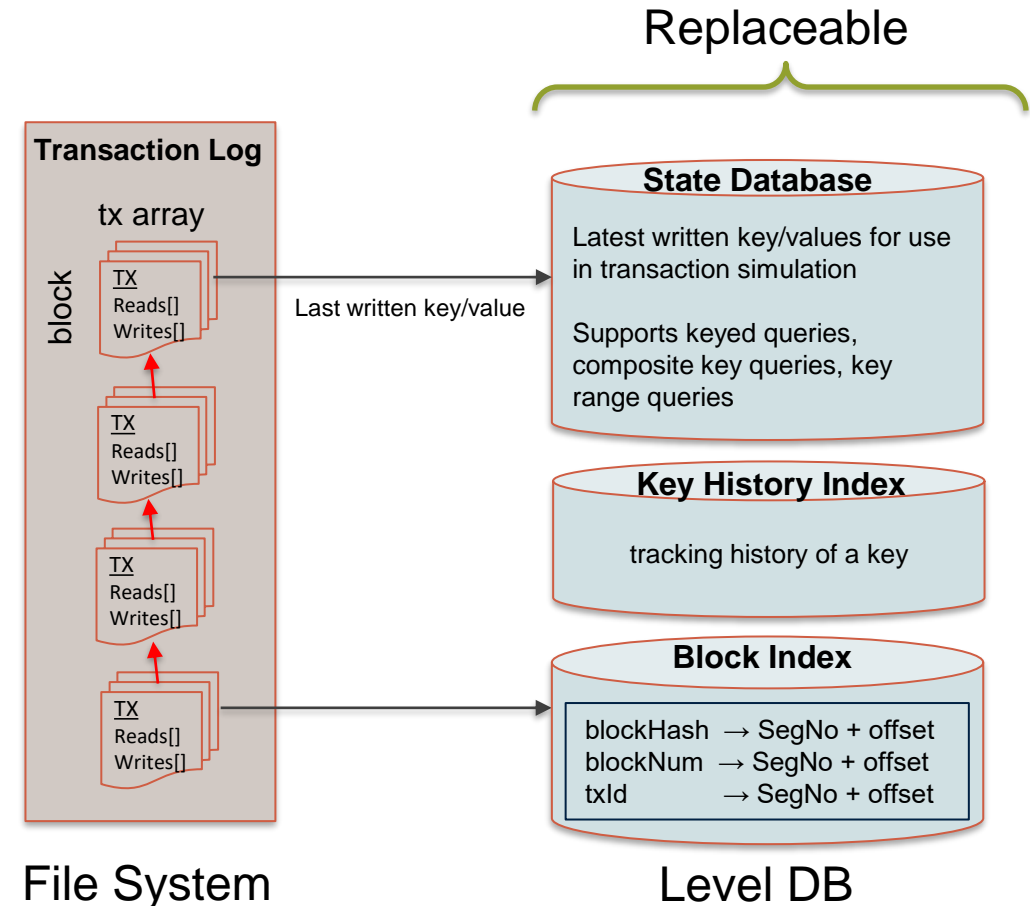


A block in Fabric consists of transactions that define reads and writes on the shared state data.

The Blockchain is saved on the file system, just like in Ethereum and most other Blockchain systems.

Reading from the Blockchain can be slow when the Blockchain on the file system has to be scanned.

Fabric uses a database that contains the current state of the ledger. This provides peers a very efficient data structure to retrieve ledger information.



Source: <https://www.hyperledger.org/resources/universities#educators>

All **peers** implement a **default state database** based on **LevelDB**. It is a key-value store developed and maintained by Google.

## Features according to Google include

- Keys and values are arbitrary byte arrays.
- Data is stored sorted by key.
- Callers can provide a custom comparison function to override the sort order.
- The basic operations are Put(key,value), Get(key), Delete(key).
- Multiple changes can be made in one atomic batch.
- Users can create a transient snapshot to get a consistent view of data.
- Forward and backward iteration is supported over the data.
- Data is automatically compressed using the Snappy compression library.
- External activity (file system operations etc.) is relayed through a virtual interface so users can customize the operating system interactions.

Source: <https://github.com/google/leveldb>



# State database CouchDB

Fabric does not enforce LevelDB as state database for the ledger.

Currently, another supported option is CouchDB which is a sophisticated document store.



A reason for using CouchDB instead of LevelDB are the broader options to store and query data.

Specially, when chaincode assets are modeled as JSON objects, CouchDB allows to query the assets directly by their properties and values.

However, **using CouchDB** (or any other custom store) requires **additional set up and security efforts**. Compared to LevelDB, **CouchDB is not part of the peer itself**, i.e. the CouchDB process will run as a separate process alongside the Fabric peer node process.

## 1. Recap

- Motivation
- Exemplary use case

## 2. Fabric

- Transaction flow
- Channels
- Single channel networks
- Multi channel networks
- State database

## 3. Composer

- Motivation

### Hyperledger Caliper

Hyperledger Caliper is a blockchain benchmark tool, which allows users to measure the performance of a specific blockchain implementation with a set of predefined use cases.

» [LEARN MORE](#)



Hyperledger Cello aims to bring the on-demand "as-a-service" deployment model to the blockchain ecosystem to reduce the effort required for creating, managing and terminating blockchains.

» [LEARN MORE](#)



Hyperledger Composer is a collaboration tool for building blockchain business networks, accelerating the development of smart contracts and their deployment across a distributed ledger.

» [LEARN MORE](#)



Hyperledger Explorer can view, invoke, deploy or query blocks, transactions and associated data, network information, chain codes and transaction families, as well as any other relevant information stored in the ledger.

» [LEARN MORE](#)



Hyperledger Quilt offers interoperability between ledger systems by implementing ILP, which is primarily a payments protocol and is designed to transfer value across distributed ledgers and non-distributed ledgers.

» [LEARN MORE](#)

Source: <https://www.hyperledger.org/projects>

# Motivation for Hyperledger Composer

Developing blockchain-based solutions using Fabric is usually a complex process. Besides programming the actual business logic, a lot of management and policy overhead comes on top.

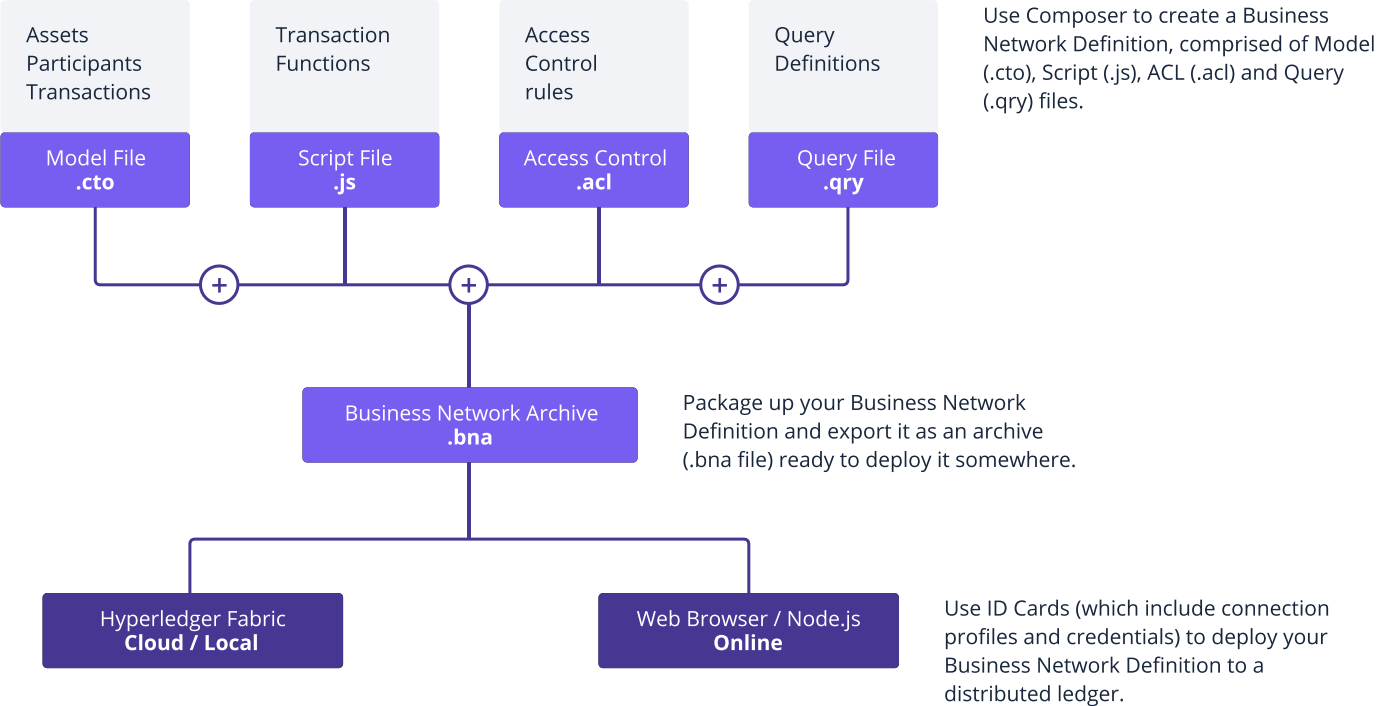
## Efforts to set up a Fabric network including business logic

- Modelling assets, participants and transactions
- Defining an access control policy
  - Who is allowed to participate?
  - How are participants authenticated / authorized?
  - What is the process to add new participants?
- Defining smart contracts and their functionality
- How can participants and applications interact with the ledger?
  - Which queries are allowed?
  - How is consensus achieved?

Composer is a toolset to develop and deploy Blockchain solutions.

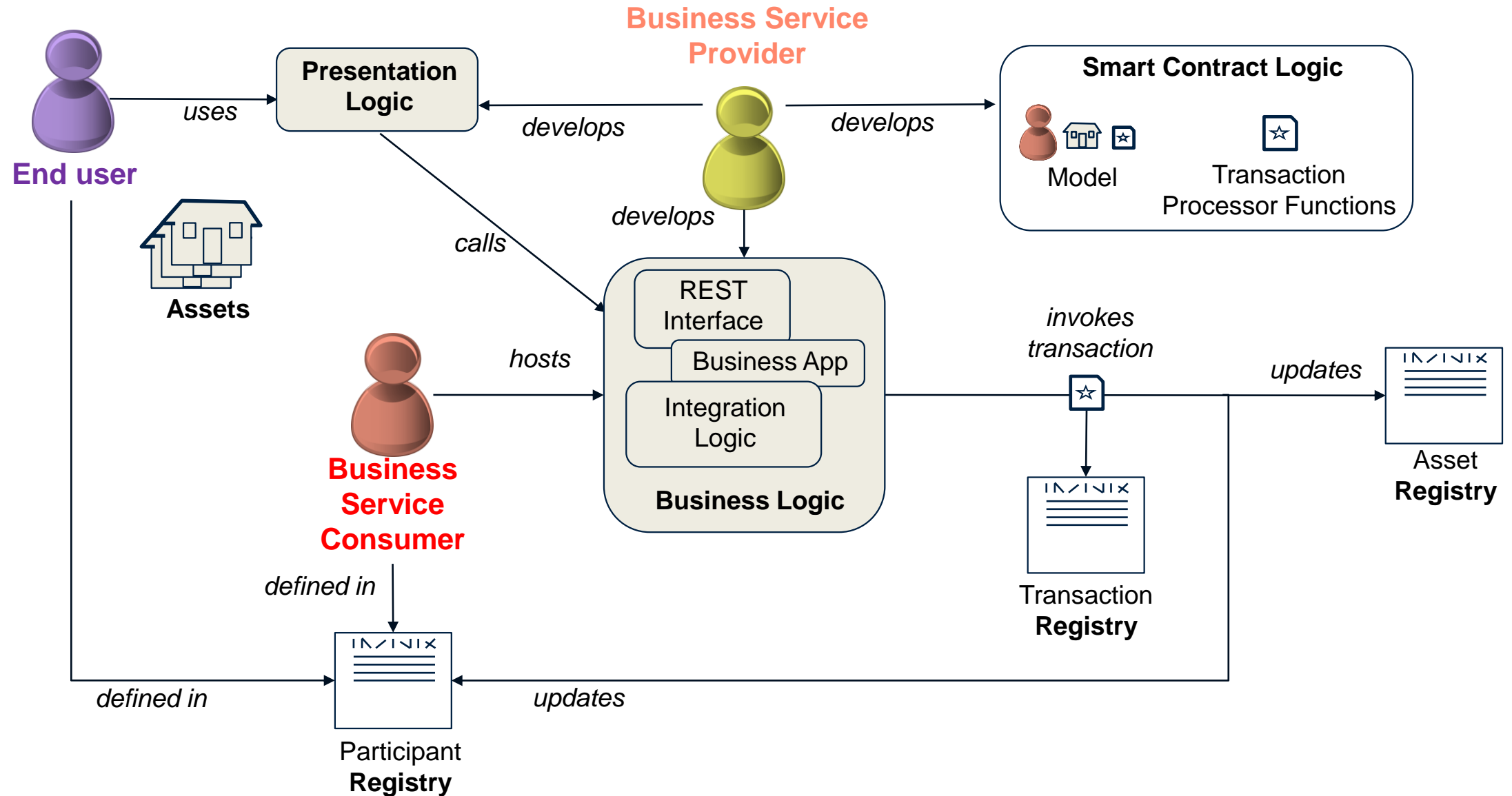
The tool provides a user interface to model a Blockchain use case in an enterprise consortium environment.

The following diagram illustrates the individual parts that can be defined using Composer.



Source: <https://hyperledger.github.io/composer/v0.16/introduction/introduction.html>

# Key concepts for the business service provider



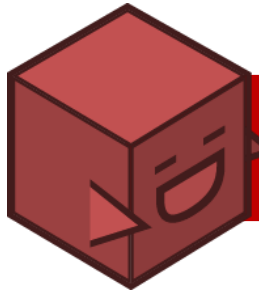


# The business service provider develops three components



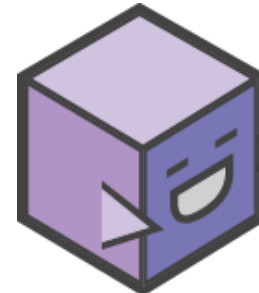
## Smart Contracts

- Implements the **logic** deployed to the Blockchain
  - Models describe assets, participants & transactions
  - Transaction processors provide the JavaScript implementation of transactions
  - ACLs define privacy rules
  - May also define events and registry queries



## Business Logic

- **Services** that interact with the registries
  - Create, delete, update, query and invoke smart contracts
  - Implemented inside business applications, integration logic and REST services
- Hosted by the Business Application Consumer



## Presentation Logic

- Provides the **front-end** for the end-user
  - May be several of these applications
  - Interacts with business logic via standard interfaces (e.g. REST)
- Composer can generate the REST interface from model and a sample application

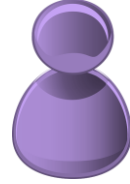
# Assets, participants and transactions



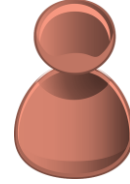
Vehicle



Vehicle Listing



Member



Auctioneer



Place Offer  
Close Bidding

```
asset Vehicle identified by vin {  
  o String vin  
  --> Member owner  
}
```

```
asset VehicleListing identified by listingId {  
  o String listingId  
  o Double reservePrice  
  o String description  
  o ListingState state  
  o Offer[] offers optional  
  --> Vehicle vehicle  
}
```

```
abstract participant User identified by email {  
  o String email  
  o String firstName  
  o String lastName  
}  
  
participant Member extends User {  
  o Double balance  
}  
  
participant Auctioneer extends User {  
}
```

```
transaction Offer {  
  o Double bidPrice  
  --> VehicleListing listing  
  --> Member member  
}  
  
transaction CloseBidding {  
  --> VehicleListing listing  
}
```

Transaction  
Processors

```
/**  
 * Close the bidding for a vehicle listing and choose the  
 * highest bid that is  
 * @param {org.acme.vehicle.auction.VehicleListing} listing  
 * @transaction  
 */  
function closeBidding(  
  var listing = clos  
  if (listing.state
```

```
/**  
 * Make an Offer for a VehicleListing  
 * @param {org.acme.vehicle.auction.Offer} offer - the offer  
 * @transaction  
 */  
function makeOffer(offer) {  
  var listing = offer.listing;  
  if (listing.state !== 'FOR_SALE') {
```

```
rule EverybodyCanReadEverything {  
  description: "Allow all participants read access to all resources"  
  participant: "org.acme.sample.SampleParticipant"  
  operation: READ  
  resource: "org.acme.sample.*"  
  action: ALLOW  
}
```

```
rule OwnerHasFullAccessToTheirAssets {  
  description: "Allow all participants full access to their assets"  
  participant(p): "org.acme.sample.SampleParticipant"  
  operation: ALL  
  resource(r): "org.acme.sample.SampleAsset"  
  condition: (r.owner.getIdentifier() == p.getIdentifier())  
  action: ALLOW  
}
```

```
rule SystemACL {  
  description: "System ACL to permit all access"  
  participant: "org.hyperledger.composer.system.Participant"  
  operation: ALL  
  resource: "org.hyperledger.composer.system.*"  
  action: ALLOW  
}
```

- It is possible to restrict which resources can be read and modified by which participants
  - Rules are defined in an `.acl/` file and deployed with the rest of the model
  - Transaction processors can also look up the current user and implement rules programmatically
- ACL rules can be simple (e.g. everybody can read all resources) or more complex (e.g. only the owner of an asset can do everything to it)
- Application supplies credentials (*userid/secret*) of the participant when connecting to the Fabric network
  - This also applies to Playground!
  - Remember to grant System ACL all access if necessary

- Events allow applications to take action when a transaction occurs
  - Events are defined in models
  - Events are emitted by transaction processor scripts
  - Events are caught by business applications
- Caught events include transaction ID and other relevant information
- Queries allow applications to perform complex registry searches
  - They can be statically defined in a separate *.qry* file or generated dynamically by the application
  - They are invoked in the application using `buildQuery()` or `query()`
  - Queries require the Blockchain to be backed by CouchDB

```
event SampleEvent {  
  --> SampleAsset asset  
  o String oldValue  
  o String newValue  
}
```

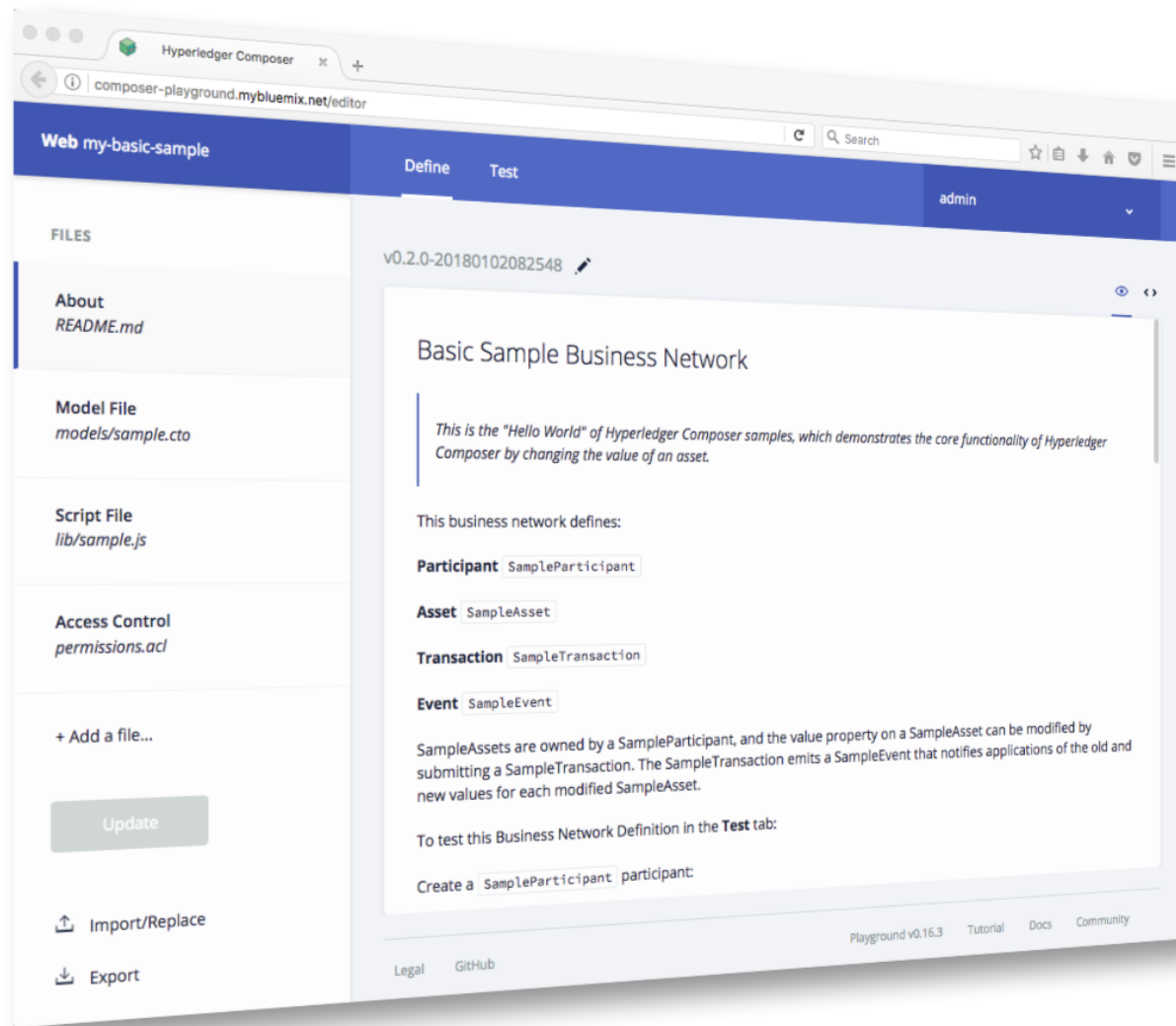
```
// Emit an event for the modified asset.  
var event = getFactory().newEvent('org.acme.sample', 'SampleEvent');  
event.asset = tx.asset;  
event.oldValue = oldValue;  
event.newValue = tx.newValue;  
emit(event);
```

```
businessNetworkConnection.on('SampleEvent', (event) => {  
  console.log(event);  
})
```

```
query selectCommoditiesWithHighQuantity {  
  description: "Select commodities based on quantity"  
  statement:  
    |  
    | SELECT org.acme.trading.Commodity  
    | WHERE (quantity > 60)  
}
```

```
return query('selectCommoditiesWithHighQuantity', {})
```

# Smart Contract development: Composer playground



- Web tool for defining and testing Hyperledger Composer models and scripts
- Designed for the application developer
  - Define assets, participants and transactions
  - Implement transaction processor scripts
  - Test by populating registries and invoking transactions
- Deploy to instances of Hyperledger Fabric V1, or simulate completely within browser

# Creating the business and end-user applications

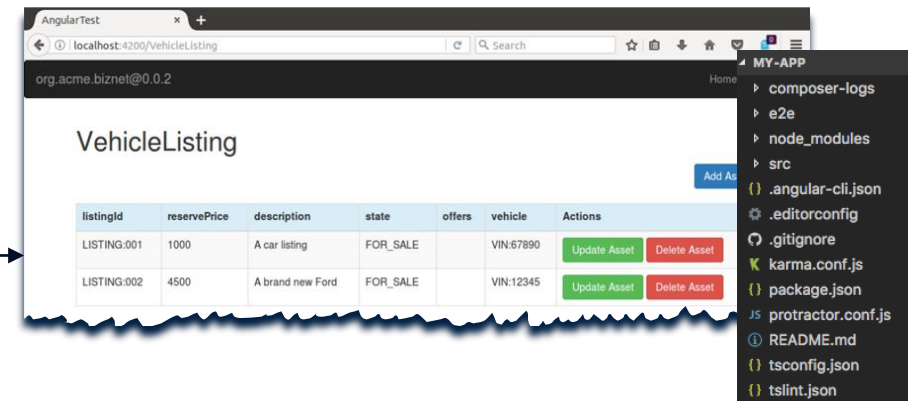
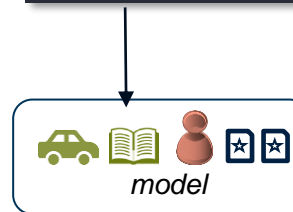
- JavaScript business applications require() the NPM “composer-client” module
  - This provides the API to access assets, participants and transactions
  - RESTful API (via Loopback) can also be generated... see later
- Command-line tool available to generate end-user command-line or Angular2 applications from model
  - Also helps with the generation of unit tests to help ensure quality code

```
const BusinessNetworkConnection = require('composer-client')
    .BusinessNetworkConnection;

this.bizNetworkConnection = new BusinessNetworkConnection();
this.titlesRegistry = this.bizNetworkConnection.getAssetRegistry
    ('net.biz.digitalPropertyNetwork.LandTitle')

let landTitle1 = factory.newResource
    ('net.biz.digitalPropertyNetwork', 'LandTitle', 'LID:1148');
landTitle1.owner = ownerRelation;
landTitle1.information = 'A nice house in the country';
this.titlesRegistry.add(landTitle1);
```

```
yo hyperledger-composer
```



VehicleListing

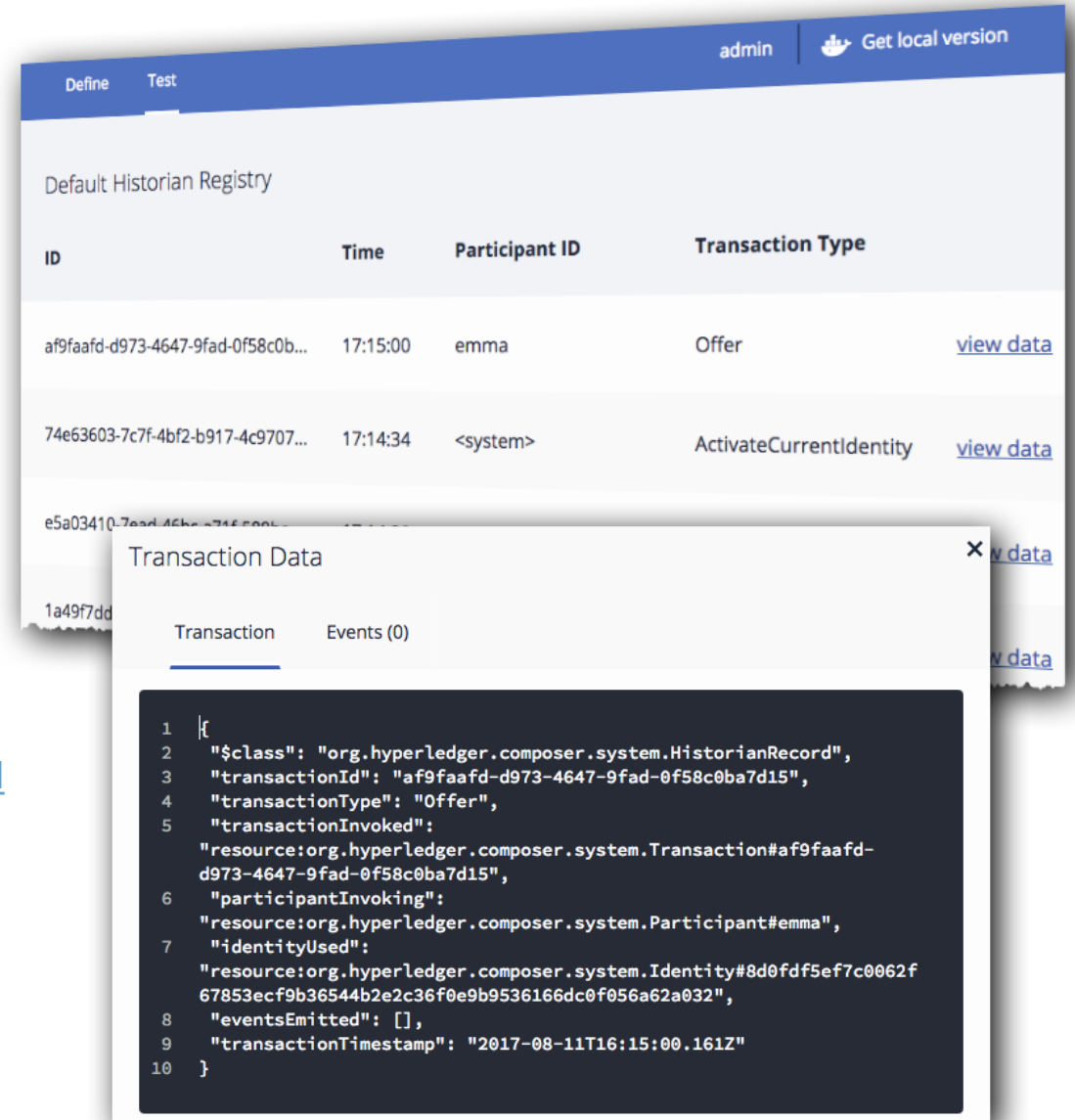
listingId	reservePrice	description	state	offers	vehicle	Actions
LISTING:001	1000	A car listing	FOR_SALE		VIN:67890	<button>Update Asset</button> <button>Delete Asset</button>
LISTING:002	4500	A brand new Ford	FOR_SALE		VIN:12345	<button>Update Asset</button> <button>Delete Asset</button>

MY-APP

- composer-logs
- e2e
- node\_modules
- src
- .angular-cli.json
- .editorconfig
- .gitignore
- karma.conf.js
- package.json
- protractor.conf.js
- README.md
- tsconfig.json
- tslint.json

- Playground Historian allows you to view all transactions
  - See what occurred and when
- Diagnostics framework allows for application level trace
  - Uses the Winston Node.js logging framework
  - Application logging using *DEBUG* env var
  - Composer Logs sent to *stdout* and *./logs/trace\_<processid>.trc*
- Fabric chaincode tracing also possible (see later)
- More information online:

<https://hyperledger.github.io/composer/latest/problems/diagnostics.html>



The screenshot displays the 'Default Historian Registry' in the Hyperledger Composer Playground. It features a table with columns for ID, Time, Participant ID, and Transaction Type. Two transactions are visible: an 'Offer' transaction by 'emma' and an 'ActivateCurrentIdentity' transaction by '<system>'. A 'Transaction Data' modal is open, showing the JSON details for the 'Offer' transaction.

ID	Time	Participant ID	Transaction Type
af9faafd-d973-4647-9fad-0f58c0b...	17:15:00	emma	Offer
74e63603-7c7f-4bf2-b917-4c9707...	17:14:34	<system>	ActivateCurrentIdentity

```
1 {  
2   "$class": "org.hyperledger.composer.system.HistorianRecord",  
3   "transactionId": "af9faafd-d973-4647-9fad-0f58c0ba7d15",  
4   "transactionType": "Offer",  
5   "transactionInvoked":  
6     "resource:org.hyperledger.composer.system.Transaction#af9faafd-d973-4647-9fad-0f58c0ba7d15",  
7   "participantInvoking":  
8     "resource:org.hyperledger.composer.system.Participant#emma",  
9   "identityUsed":  
10    "resource:org.hyperledger.composer.system.Identity#8d0fdf5ef7c0062f67853ecf9b36544b2e2c36f0e9b9536166dc0f056a62a032",  
11   "eventsEmitted": [],  
12   "transactionTimestamp": "2017-08-11T16:15:00.161Z"  
13 }
```



# IBM deprecates Hyperledger Composer

At this time, IBM has decided to reduce the investment it makes towards developing Composer, in order to focus the team on directly delivering improvements into Fabric. At present, the team is working on two big features, due to be released as part of Fabric v1.3, that will improve the underlying chaincode and application APIs in Fabric. We believe that these features will start to provide the underpinnings of an improved development experience for Fabric users, and we are looking to continue to deliver improvements in Fabric v1.4 and onwards - for example, you may see modelling, REST APIs, code generators, IDE extensions, and online playgrounds start to appear in Fabric over time.

The IBM team will continue to update Composer to maintain compatibility with the latest Fabric v1.x releases, and we will fix any high priority bugs - but certainly for the time being, we will not be looking at delivering any major new features into Composer.

We would also like to make it clear that we are still more than happy to engage with potential new contributors, and help you get up and running with delivering the features and bug fixes that you deem important into Composer yourself. If you're interested, we'd also love to have you come and take a look at what we're doing in Fabric. Our goal is as always to enhance the experience for developers who are developing blockchain solutions , and as developers - you're best placed to give us feedback on what we're doing.

Finally, Dan (now at Clause.io) has also proposed an exciting plan regarding the future of the modelling language, which involves splitting the code out so it can be more easily consumed and used by a much wider range of blockchain projects. We'll be looking to publish a sample that shows the modelling language being used in conjunction with smart contracts in Fabric v1.3, and hope to extend the code to support additional programming languages (Go, Java, etc).

Many thanks,

SimonUnless stated otherwise above:

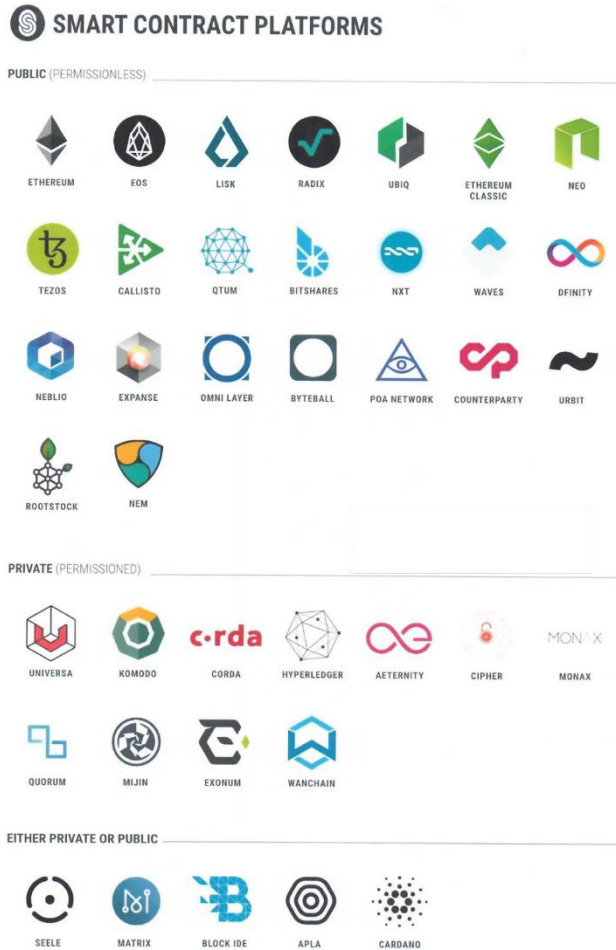
IBM United Kingdom Limited - Registered in England and Wales with number 741598.

Registered office: PO Box 41, North Harbour, Portsmouth, Hampshire PO6 3AU

Source: <https://lists.hyperledger.org/g/composer/message/125>



# Outlook on Smart Contract platforms and languages



- Currently, more than 23 public and permissionless smart contract platforms exist
  - Ethereum, Neo and EOS are the largest
- More than 11 private smart contract platforms exist

Source: <https://hackernoon.com/comparison-of-smart-contract-platforms-2796e34673b7>